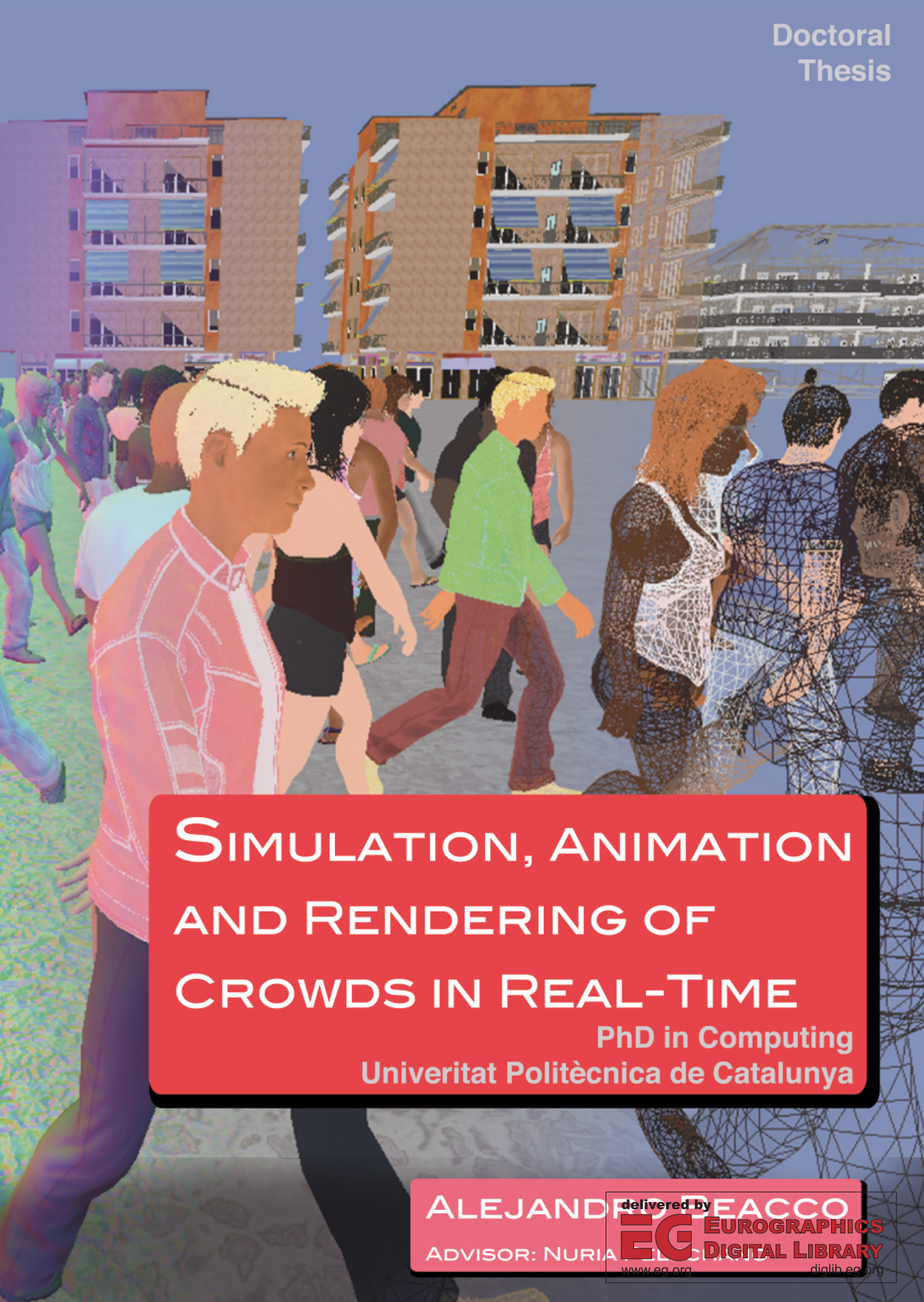
Doctoral Thesis



Simulation, Animation and Rendering of Crowds in Real-Time

PhD in Computing
Univeritat Politècnica de Catalunya

Alejandro Beacco
Advisor: Nuria Pelechano

Universitat Politècnica de Catalunya
(UPC)

Doctoral Thesis

Simulation, Animation and Rendering of Crowds in Real-Time

Author:
Alejandro Beacco

Supervisor:
Dr. Nuria Pelechano

A thesis submitted in fulfilment of the requirements
for the degree of Doctor of Philosophy

in the

Research Center for Visualization, Virtual Reality and Graphics Interaction (ViRVIG)
Departament de Ciències de la Computació (CS)

October 16, 2014

Declaration of Authorship

I, Alejandro Beacco, declare that this thesis titled, 'Simulation, Animation and Rendering of Crowds in Real-Time' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

PROXIMO

– I was the best because the crowd loved me. Win the crowd, win your freedom.

MAXIMUS

– I will win the crowd. I will give them something they have never seen before.

Gladiator (2000) movie script by D. Franzoni, revised by J. Logan.

Abstract

Simulation, Animation and Rendering of Crowds in Real-Time

by Alejandro Beacco

Nowadays crowd simulation is becoming more important in computer applications such as building evacuation planning, training, videogames, etc., presenting hundreds or thousands of agents navigating in virtual environments. Some of these applications need to run in real time in order to offer complete interaction with the user. Simulated crowds should seem natural and give a good looking impression to the user. The goal should be to produce both the best motion and animation, while minimizing the awkwardness of movements and eliminating or hiding visual artifacts. Achieving simulation, animation and rendering of crowds in real-time becomes thus a major challenge. Although each of these areas has been studied individually and improvements have been made in the literature, its integration in one real-time system is not straight forward. In the process of integrating animation, simulation and rendering of real time crowds, we need to assume some trade-offs between accuracy and quality of results.

The main goal of this thesis is to work on those three aspects of a real-time crowd visualization (simulation, animation and rendering) seeking for possible speed-ups and optimizations allowing us to further increase the number of agents in the simulation, to then integrate them in a real-time system, with the maximum number possible of high quality and natural looking animated agents. In order to accomplish our goal we present new techniques to achieve improvements in each one of these areas: In crowd simulation we work on a multi-domain planning approach and on planning using footsteps instead of just root velocities and positions; in animation we focus on a framework to eliminate foot sliding artifacts and on synthesizing motions of characters to follow footsteps; in rendering we provide novel techniques based on per joint impostors. Finally we present a novel framework to progressively integrate different methods for crowd simulation, animation and rendering. The framework offers level-of-detail for each of these areas, so that as new methods are integrated they can be combined efficiently to improve performance.

Acknowledgements

This PhD. Thesis would not have been possible without the great supervision of my advisor, Dr. Nuria Pelechano, to whom I ought being where I am now. She has been the perfect guide to a work she envisioned from the first moment, and she has been able to push me in those moments when you think your work is stuck. She has almost as much credit as I have in all the work I have done.

I also have to thank Dr. Carlos Andújar for his guidance and collaboration in all my work on rendering, as well as Dr. Bernhard Spanlang for the same reasons and for assisting me in the use of his animation library. Also, I need to thank Dr. Mubbasir Kapadia and Professor Norman I. Badler, for supervising me during my stay at the University of Pennsylvania, which has lead us to more than one collaboration.

And of course I also want to give thanks to my girlfriend, Teresa, who has supported me and my work all these years; to my parents, my sister, and the rest of my family for the same reason, and because they have always been there even when they did not really know what I was doing at a PhD.; and to all my friends and colleagues who have accompanied me during this journey. To my friends from university days: Albert, Carlos, Ignasi, Marc, Óscar, Víctor, Leti, and Gloria. To my friends from work, colleagues or excolleagues: Adri, Carles, Dani, Eva, Ferrán, Genís, Héctor, Imanol, Isaac, Jesús D., Jesús O., Jesús R., Jordi M., Jordi S., Jose, Lázaro, M. Àngels, Marc, Marcos, María, Miguel-Ángel, Oriol, Óscar, Pedro, Prithiviraj, Ramón, Roger, Sergi, Víctor and Xavi. And to other friends I have made or maintained during the PhD. like: Alba, Atia, Enric, Eva, Gerard, Henar, Javi, Jorge, Mariela, Miguel, Nico, Sergio, Sofía, and Xavi.

I also want to give special thanks to other Doctors and Professors who have helped me or guided me at some point in all my years of research: Isabel Navazo, Pere Brunet, Àlvar Vinàcua, Pere-Pau Vázquex, Marta Fairén, Toni Chica, Toni Susín, Alex Rios and Gustavo Patowe.

Finally, this thesis has also been possible thanks to the grant FPUAP2009-2195 (Spanish Ministry of Education). The different works presented in this document have also been partially funded by the Spanish Ministry of Science and Innovation under Grant TIN2010-20590-C02-01.

Contents

Declaration of Authorship i

Abstract v

Acknowledgements vii

Contents ix

1 Introduction 1
- 1.1 Motivations . . . 2
- 1.2 Problems . . . 4
- 1.3 Goals . . . 6
- 1.4 Contributions . . . 7
- 1.5 Document Organization . . . 10

2 Concepts 11
- 2.1 Simulation . . . 12
  - 2.1.1 Crowd Model . . . 12
  - 2.1.2 Crowd Simulation . . . 13
- 2.2 Agent Complexity . . . 15
  - 2.2.1 Representation . . . 15
  - 2.2.2 Static Parameters . . . 16
  - 2.2.3 Variable Data . . . 16

2.3 Control Granularity . . . 16
2.3.1 Reactive and Local Motion . . . 17
2.3.2 Planning and Global Motion . . . 17
2.3.2.1 Navigation Meshes . . . 18
2.3.2.2 Planner . . . 18
2.3.2.3 Prediction . . . 19
2.3.3 Behavior . . . 20
2.3.4 Complete Control . . . 20
2.4 Environment Complexity . . . 21
2.4.1 Static . . . 21
2.4.2 Dynamic . . . 21
2.5 Time Discretization . . . 22
2.6 Animation . . . 23
2.6.1 Frames and Keyframes . . . 24
2.6.2 Blending Between Keyframes . . . 24
2.6.3 Character Animation . . . 24
2.6.3.1 Skeleton . . . 25
2.6.3.2 Pose . . . 25
2.6.3.3 Blending Between Poses . . . 26
2.6.4 Animation Clip . . . 26
2.6.4.1 Transitions . . . 27
2.6.4.2 Cycling Clip . . . 27
2.6.4.3 In Place Animation . . . 27
2.7 Character Control . . . 28
2.7.1 Crowd Animation Synthesis . . . 28
2.7.2 Animation Preprocess . . . 29
2.7.2.1 Angle of Movement Vs. Orientation Angle . . . 29
2.8 Animation Quality . . . 30
2.9 Rendering . . . 31
2.9.1 Mesh Deformation . . . 31
2.9.2 Level of Detail . . . 31
2.9.3 Impostor . . . 33
2.10 Visual Quality . . . 33
2.10.1 Uncanny Valley . . . 33
2.10.2 Variety . . . 35
2.11 Real-Time Crowds . . . 36
2.11.1 Scale . . . 36
2.11.2 Performance . . . 36
2.11.3 Integration, Global Coherence and Consistency . . . 37
2.12 Problem Statement . . . 38

3 State of the Art **41**
3.1 State of the Art on Crowd Simulation . . . 42
3.1.1 Macroscopic Models . . . 42

3.1.2 Microscopic Models and Local Movement 43
3.1.2.1 Rule-Based Models 43
3.1.2.2 Social Forces Models 44
3.1.2.3 Velocity-Based Models 45
3.1.2.4 Continuum Dynamics Models 47
3.1.2.5 Cellular Automata Models 47
3.1.2.6 Footstep-Driven Approaches 48
3.1.2.7 Animation-Dependent Planners 49
3.1.3 Data-Driven Techniques 51
3.1.3.1 Parameter Extraction 51
3.1.3.2 Example-Based Simulation 52
3.1.4 Global Planning 53
3.1.4.1 A* and Weighted A* 53
3.1.4.2 Incremental Planners 53
3.1.4.3 Anytime Algorithms 54
3.1.4.4 Anytime Dynamic A* 54
3.1.5 Multi-Domain and Hierarchical Planning 55
3.1.6 Crowd Simulation Conclusions 56
3.2 State of the Art on Crowd Animation 58
3.2.1 Motion Synthesis 58
3.2.1.1 Procedural Techniques 58
3.2.1.2 Physics-Based Techniques 58
3.2.1.3 Example-Based Techniques 59
3.2.2 Synthesizing Crowd Motions From Root Trajectories . . . 62
3.2.3 Footstep-Driven Animation Systems 66
3.2.4 Crowd Animation Conclusions 71
3.3 State of the Art on Crowd Rendering 73
3.3.1 Character Animation and Skinning 73
3.3.1.1 Skeletal Animation 74
3.3.1.2 Animation Blending 74
3.3.1.3 Non-Skeletal Animation 76
3.3.1.4 Individuality 77
3.3.2 Point-Based Techniques 77
3.3.3 Image-Based Techniques 79
3.3.3.1 Dynamic Impostors 80
3.3.3.2 Pre-Generated Impostors 81
3.3.4 Geopostors 82
3.3.5 Layered Impostors 83
3.3.6 Polypostors 85
3.3.7 Per-Joint Impostors 85
3.3.8 Culling Techniques 86
3.3.9 Level-Of-Detail (LOD) 88
3.3.9.1 Generation 89
3.3.9.2 Tessellation 90

3.3.9.3 LOD Selection . . . 91
3.3.10 Hardware Improvements . . . 92
3.3.10.1 Instancing and Pseudo Instancing . . . 93
3.3.10.2 Palette Skinning . . . 94
3.3.10.3 Dynamic Caching . . . 94
3.3.11 Increasing Realism . . . 95
3.3.12 Comparison . . . 97
3.3.13 Crowd Rendering Conclusions . . . 98
3.4 Existing Tools for Crowds . . . 102
3.4.1 Commercial Solutions . . . 102
3.4.2 Libraries . . . 102
3.4.3 Engines . . . 103
3.4.4 Research Platforms . . . 104
3.4.5 Conclusions on Existing Tools for Crowds . . . 105
Publications and More . . . 107

4 Contributions to Crowd Simulation 109
4.1 Planning in Multiple Domains . . . 110
Role in this work . . . 111
4.1.1 Overview . . . 111
4.1.2 Planning Domains . . . 112
4.1.3 Problem Decomposition and Multi-Domain Planning . . . 118
4.1.4 Relationship Between Domains . . . 119
4.1.4.1 Domain Mapping . . . 120
4.1.4.2 Mapping Successive Waypoints to Independent Planning Tasks. . . . 120
4.1.4.3 Tunnels . . . 121
4.1.5 Results . . . 123
4.1.5.1 Comparative Evaluation of Domain Relationships 123
4.1.5.2 Performance . . . 125
4.1.5.3 Scenarios . . . 127
4.2 Planning Using Footsteps . . . 130
4.2.1 Overview . . . 130
4.2.1.1 Events Monitor . . . 132
4.2.2 Preprocess . . . 132
4.2.2.1 Locomotion Modes . . . 133
4.2.2.2 Footsteps Extraction . . . 134
4.2.2.3 Clip Annotation . . . 134
4.2.3 Planning Footstep Trajectories . . . 136
4.2.3.1 High Level Path Planning . . . 136
4.2.3.2 Problem Definition . . . 137
4.2.3.3 Real-Time Planning Algorithm . . . 138
4.2.3.4 Pruning Rules . . . 139
4.2.3.5 Collision Prediction . . . 140

4.2.4 Animation Engine . . . 143
4.2.5 Results . . . 143
4.3 Conclusions on Crowd Simulation . . . 146
Publications . . . 148

5 Contributions to Crowd Animation 149
5.1 Reflecting the Root Motion . . . 150
5.1.1 Framework . . . 150
5.1.2 Animation Planning Mediator . . . 153
5.1.2.1 Animation Clip Selection . . . 153
5.1.2.2 Blending Factors . . . 156
5.1.2.3 Calculation of Root Displacement . . . 157
5.1.2.4 Updating the Skeletal State . . . 159
5.1.2.5 The Algorithm . . . 160
5.1.3 Results . . . 161
5.2 Synthesizing Motion Following Footsteps . . . 164
5.2.1 Framework Overview . . . 165
5.2.2 Footstep-Based Locomotion . . . 166
Motion Clip Analysis . . . 166
Footstep and Root Trajectories . . . 168
Online Selection . . . 169
Interpolation . . . 169
Inverse Kinematics . . . 171
5.2.3 Incorporating Root Movement Fidelity . . . 172
5.2.4 Results . . . 176
Foot Placement Accuracy . . . 178
Performance . . . 178
5.3 Conclusions on Crowd Animation . . . 180
Publications . . . 182

6 Contributions to Crowd Rendering 183
6.1 Relief Per-Joint Impostors . . . 184
Relief Mapping . . . 186
6.1.1 Our Approach . . . 187
Overview . . . 187
Construction . . . 188
LOD for Relief Impostors . . . 191
Real-Time Rendering . . . 192
6.1.2 Results . . . 194
Impostor Creation . . . 194
Image Quality . . . 196
Mesh vs Impostor Rendering . . . 198
Choosing the Fastest Representation . . . 204
Crowd Rendering Performance . . . 204

User Study . . . 207
6.2 Flat Per-Joint Impostors . . . 209
6.2.1 Overview . . . 210
6.2.2 Preprocessing . . . 212
Bone Hierarchy Simplification . . . 213
Choosing View Samples From S^2 . . . 213
Impostor Generation . . . 214
6.2.3 Real-Time Rendering . . . 219
CPU Stage . . . 219
Vertex Shader . . . 220
Geometry Shader . . . 220
Fragment Shader . . . 220
6.2.4 Results . . . 222
Implementation details . . . 222
Illustrative Results . . . 223
Performance . . . 230
Discussion . . . 232
6.3 Conclusions on Crowd Rendering . . . 233
Publications . . . 235

7 Crowd Framework **237**
7.1 Introduction . . . 238
Motivation and Current State . . . 238
7.2 Overview . . . 239
7.2.1 Simulation . . . 240
Pathfinding . . . 240
Agent Controller . . . 241
Crowd Simulation . . . 242
7.2.2 Rendering . . . 243
Scene Render . . . 243
Character Representations . . . 245
7.2.3 Animation . . . 247
Preprocessing Animations . . . 247
Animation Controller . . . 247
Instancing and Palette Skinning . . . 248
7.2.4 Integration . . . 249
7.3 Features . . . 250
7.4 Results and Discussion . . . 252
7.5 Conclusions on Crowd Framework . . . 254
Publications . . . 254

8 Future Work and Conclusions **255**
8.1 Future Work . . . 256
8.2 Conclusions . . . 258

Bibliography **261**

Dedicated to my parents, to my sister Clara,
and to my girlfriend Teresa

1 . Introduction

This chapter introduces the motivations and some of the main problems of this thesis. We also present our goals, and list all the contributions to the area with their related publications. Following this chapter, the reader can find a chapter where all the main concepts related to the topics covered by this thesis are explained in detail. The reader may want to skip this chapter if he is already familiar with these topics.

1.1 . Motivations

In our everyday lives we encounter a lot of people. They conform an essential part of our cities, our societies, our environments and definitely our lives. We see people going to work, waiting for a train or bus, meeting friends, working and performing a huge number of activities. People can create a rich tapestry of activities during the day, one of which we might not be conscious about. But if suddenly we would not see so much people or no one (this could obviously happen at some time) we would immediately notice the difference, the absence of it (Figure 1.2). Precisely this aspect, this diversity of characters, activities and movements, is what a lot of computer graphic simulations, presenting 3D environments inhabited by animated virtual humans [Pelechano et al., 2008], lack of.

Figure 1.1: A street in a normal day in Chengdu, China.

Simulating and visualizing people's activities can be done for different purposes. There are a lot animation and simulation in computer applications where you need modeling virtual crowds of autonomous agents. Some of these applications include planning, education, entertainment, training, and human factor analysis for building evacuations. Other applications include simulations of huge scenarios where masses of people congregate, flow and disperse, like transport centers, sport events, or concerts. A lot of crowd simulations only include basic movement behaviors, possibly along with some stochastic actions.

FIGURE 1.2: We notice that something is wrong when seeing an empty London street (photo extracted from the *28 Days Later* motion picture).

From all these computer graphic applications simulating crowds we distinguish the ones that are in real-time from the ones that are not. In the movies industry it is easy to watch some scenes with high quality virtual crowds, masses, armies, etc., like in *The Lord Of The Rings* (Figure 1.3); these are precomputed simulations for which the visualization rendering process can require a lot of hours. For real-time applications, like videogames or the ones for a virtual reality system, where interactivity is crucial, the speed of computation becomes fundamental. We thus require not only navigation algorithms for one agent in a huge virtual environment while avoiding obstacles and other agents; we also need efficient algorithms for rendering high complex scenes with animated characters represented by completely jointed 3D figures or another equivalent representation.

FIGURE 1.3: The Lord Of The Ring movies used the MASSIVE software to render army crowds.

1.2 . Problems

Trying to achieve a high level of realism, each one of these areas can become a bottle-neck for a real-time simulation. Therefore, it is necessary to have a trade-off between accuracy and speed of computation. Simulating human motion accurately, while satisfying physical constraints and maintaining its temporal restrictions, is not an easy task. Although there are currently a lot of techniques developed in order to synthesize motions for one agent [Treuille et al., 2007], these are not easily extensible for large numbers of agents simulated in real-time. Moreover, depending on how the agent is simulated the set of parameters and constraints to animate its character can go from just a velocity vector to a complete set of footprints to follow.

Figure 1.4: Grand Theft Auto V (2013) for Playstation 3 and XBox 360

For example, in videogames, since two generations ago the sandbox genre has become very popular. This kind of game usually has the user controlling his avatar in an open city or region, with no predefined paths. In this usually large environment, the user decides almost always what to do and where to go. But to be realistic the 3D environment must be inhabited by virtual characters. So, if we take one videogame of this kind, such as the last hit *Grand Theft Auto V* (Figure 1.4) from Rockstar Games or Assassin's Creed IV Black Flag (Figure 1.5) from UbiSoft, we are able to notice some things that still are far from perfect.

FIGURE 1.5: Assassin's Creed IV Black Flag (2013) for PC, Playstation 3, XBox 360, WiiU, Playstation 4 and XBox One

First of all, it is often noticeable that there are not as many characters in the environment as one would expect in certain situations (street markets with just a hand full of animated characters). And in the cases where big crowds are involved in the game, these are mostly standing in the same place with no particular inner interaction, moving and reacting like a unique entity. So the first desirable goal would be to have more people inhabiting real time virtual worlds and that the particular agents of a crowd exhibit individual interactions and goals.

Secondly, when staring at the virtual crowd for a certain amount of time, we would notice, repetitions and lack of variety in the models, animations and motions of the agents. The cloning impression of this is negative. Individuality again should be a target.

Third, if we observe the visual quality of the non-controlled characters, they will always have less detail than the main characters; sometimes we can even notice the differences between levels of detail. Effects like popping are very common, as well as limiting the maximum viewing distance with some fog trick. Improving the performance while maintaining the visual impression over the user is therefore a major challenge.

Fourth, we will have disturbing artifacts such as the foot-sliding effect, where the feet seem to slide on the floor. This is often due to a mismatch between the

simulation and the animation. Avoiding this and obtaining smooth, continuous and natural motions of the characters should be fulfilled to not break the illusion of the player.

Finally, we would sometimes find situations that are not controlled by the current simulator and that may produce some unnatural behaviors. All these problems have an impact in the overall impression and immersion of the user, which in the case of videogames or virtual reality applications, becomes critical.

1.3 . Goals

The main goal of this thesis is to find and propose solutions to some of the problems occurring in real-time crowd simulations, represented with 3D animated characters, while trying to improve their overall realism. The work carried out during this thesis has been focused in achieving that goal with the maximum possible efficiency, while obtaining realistic results from the point of view of the simulation, animation, and high visual quality. These techniques have been developed, with the final goal in mind of making possible to efficiently integrate all of them in the same system.

In order to achieve this main goal, we have aimed at the following specific research goals::

1. Simulation: to speed-up the simulation, and improve the natural behavior of the agents and their interactions, by developing different and novel granularities of planning control, such as planning at a footstep level, and the capacity to use more than one at the same time.

2. Animation: to avoid animation artifacts and mismatch problems with the simulation, by developing new animation controllers synthesizing motion to accurately follow the different outputs of the simulators, while respecting constraints. These controllers need to be efficient enough to work for a large amount of characters.

3. Rendering: to avoid the rendering computation bottleneck and being able to support a higher number of agents in real-time during the simulation, by developing novel and efficient image-based techniques for animated characters.

As we can see, virtual crowd visualization is a complete field consisting of an agglomeration of parts with its own problems that need to be solved. Obviously, each one of these parts could constitute a single thesis on its own. But my PhD research has focused on having all of them working at the same time in a real-time system. Therefore, these elements are not explored in all its extension, but they are researched in our concrete scenario of a real-time application.

1.4 . Contributions

The contributions of this thesis are a set of novel techniques which have in common that they are meant to work for large groups of agents in real-time, and a novel framework with an architecture that allows embedding all these elements together:

Contributions to Crowd Simulation:

- **A)** A framework that decomposes a planning problem, of navigating in complex and dynamic virtual environments, into multiple heterogeneous problems of differing complexities. Related publication:
  - **1.** M. Kapadia, **A. Beacco**, F. Garcia, V. Reddy, N. Pelechano and N.I. Badler. *Multi-Domain Real-time Planning in Dynamic Environments*. ACM SIGGRAPH /EUROGRAPHICS Symposium on Computer Animation 2013 (SCA 2013), Anaheim, CA, U.S.A., 2013
- **B)** A planner that given any set of animation clips outputs a sequence of footsteps to follow from an initial position to a goal guaranteeing obstacle

avoidance and correct spatio-temporal foot placement. Related publication:

- **2. A. Beacco**, N. Pelechano and M. Kapadia. *Dynamic Footsteps Planning for Multiple Characters*. EUROGRAPHICS Spanish Conference of Computer Graphics 2013 (EGse CEIG 2013), Madrid, Spain., 2013

Contributions to Crowd Animation:

- **C)** A technique focused on eliminating artifacts that are common in this kind of visualization, such as the well-known foot-sliding effect. Related publications:
  - **3.** N. Pelechano, B. Spanlang and **A. Beacco**. *A framework for rendering, simulation and animation of crowds*. EUROGRAPHICS Spanish Conference of Computer Graphics (EGse CEIG 2009), Donostia (San Sebastian), Spain. 9-11 September 2009.
  - **4. A. Beacco**, B. Spanlang, and N. Pelechano. *Efficient elimination of foot sliding for crowds*. In Posters Proceedings, The ACM SIGGRAPH/EUROGRAPHICS Symposium on Computer Animation (SCA 2010), pages 19-20, Madrid, Spain 2010
  - **5.** N. Pelechano, B. Spanlang, and **A. Beacco**. *Avatar locomotion in crowd simulation*. In International Conference on Computer Animation and Social Agents (CASA 2011), Chengdu, China, 2011
- **D)** A new controller synthesizing motion that satisfies accurate foot placement constraints.
  - **6. A. Beacco**, N. Pelechano, M. Kapadia, N.I. Badler. *Footstep Parameterized Motion Blending using Barycentric Coordinates*. Submitted to Computer and Graphics. *Currently under review*.

Contributions to Crowd Rendering:

- **E)** A new image-based representation of the agents based on a novel per-joint impostors approach, using relief mapping. Related publications:

  - **7. A. Beacco**, B. Spanlang, C. Andujar, and N. Pelechano. *Output-sensitive rendering of detailed animated characters for crowd simulation*. In CEIG Spanish Conference on Computer Graphic, 2010
  - **8. A. Beacco**, B. Spanlang, C. Andujar, and N. Pelechano. *A flexible approach for output-sensitive rendering of animated characters*. Computer Graphics Forum,
    30(8):2328 - 2340, 2011

- **F)** Another new image-based representation of the agents based on an improved version of the per-joint approach, but using classic flat impostors. Related publications:
  - **9. A. Beacco**, C. Andujar, N. Pelechano, and B. Spanlang. *Efficient rendering of animated characters through optimized per-joint impostors*. Computer Animation and Virtual Worlds, 23(1): 33 - 47, 2012
  - **10. A. Beacco**, C. Andujar, N. Pelechano and B. Spanlang. *Crowd Rendering with per joint impostors*. Poster in the 24th EUROGRAPHICS Symposium on Rendering (EGSR 2013), Zaragoza, Spain, 2013.

Contribution to the Integration of Simulation, Animation and Rendering of Crowds in Real-Time:

- **G)** A new prototyping testbed for crowds that lets the researcher focus on one of these areas at a time without loosing sight of the others. Related publication:
  - **11. A. Beacco** and N. Pelechano. *CAVAST: The Crowd Animation, Visualization, and Simulation Testbed*. EUROGRAPHICS Spanish Conference of Computer Graphics (EGse CEIG 2014), Zaragoza, Spain. 2-4 July 2014.

At the time of writing this document, two journal publications have been submitted and are now under revision: a publication covering our work on synthesizing motion accurately following footsteps (D-6), presented in 5.2; and a survey on real-time rendering of crowds, including most of the related work presented in 3.3. Furthermore, we plan for one more journal submissions with an extension of our multi-domain simulation work (A) presented in chapter 4.1.

Notice how our work in rendering has yielded two JCR-indexed journal publications (numbers 8 and 9). Also, a short stay of 4 months at the *Human Modeling and Simulation Lab*, of the *University of Pennsylvania*, in Philadelphia, yielded 2 publications (numbers 1 and 2), the first one at the *Symposium in Computer Animation*, as well as the submitted work on footstep motion covered in 5.2.

1.5 . Document Organization

The present document is organized as follows. The next chapter introduces us to the concepts of the crowd visualization taking simulation, animation and rendering into account. The third chapter has a complete state of the art on crowd simulation, animation and rendering. The following chapters present our contributions in all these areas. Our last contribution chapter introduces a framework for a novel prototyping development tool for crowds, allowing to embed all these parts together, and allowing researchers to quickly build their new projects on top of it. Finally we show our conclusions and expose our future work in the last chapter.

2 . Concepts

This chapters is dedicated to introduce the most relevant concepts appearing in this thesis. It presents in more detail the different dimensions of the main problem we want to attack, and provides an overview how all the different elements of the research on crowds link together, bringing to first plane crowd simulation, animation, rendering, and some of the difficulties when trying to integrate them all.

2.1 . Simulation

The first concept to address is **simulation**. The definition according to Wikipedia is the following: *"Simulation is the imitation of the operation of a real-world process or system overt time. The act of simulating something first requires that a model be developed; this model represents the key characteristics or behaviors of the selected physical or abstract system or process. The model represents the system itself, whereas the simulation represents the operation of the system over time. ... "* This means, for the purposes of crowd research, that the goal is to imitate the real-world process of crowds of people inhabiting an environment, and for that a model representing both the crowd and the environment is needed.. The simulation will be in charge of operating the different actions and behaviors of the crowd in the environment over time.

2.1.1 . Crowd Model

The arising question is therefore what is the model for a crowd simulation. And also what is a crowd. In this research what we call a **crowd** is a group of people, animals or other entity, moving, interacting and inhabiting an environment. How large is the crowd depends on the scale of the problem addressed at each time. But let us say we try to cover cases that go from 1 agent to hundreds or event thousands of agents. We will resume this question about how big a crowd is later on. A crowd is therefore an aggregation of people, what we will call in our model **agents**. One agent is one of the individuals in the crowd. In the model agents can move and perform different actions, as well as have different interactions or reactions to the events of the simulation. Another key element of the model is the environment, which should be also modeled by a 3D mesh, or abstracted with some graph structure. Finally, the last element we need to think of in our model is time and its discretization. Figure 2.1 illustrates this.

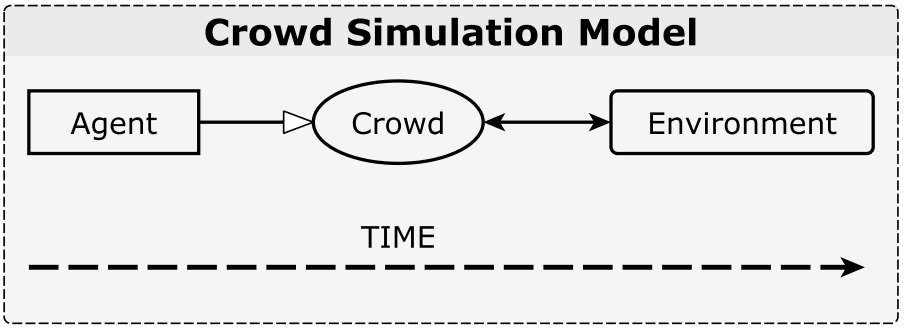


FIGURE 2.1: A crowd simulation model is composed of an aggregation of agents and the environment they inhabit. The simulation is then carried out over time.

2.1.2 . CROWD SIMULATION

Crowds can be of many different types and of a wide variety of sizes, from just a few tens, to thousands of people. Therefore, we could focus our study on a group of not too many individuals (thousands at most), or we could focus our study on a group of masses (even millions) as a whole. This lets us introduce the two main approaches on crowd simulation:

- **Microscopic models**: those models focusing on simulating local behavior of individual agents and their interaction with other agents in the crowd. Figure 2.2 shows an example scenario that should be modeled at microscopic level.
- **Macroscopic models**: those models simulating the group behavior, sometimes imitating other simulation models like fluids or particles. Figure 2.3 shows an example situation that should be modeled at a macroscopic level.

In this thesis we only consider microscopic models since the main goal is to have a real-time crowd simulation with animated characters rendered.

Figure 2.2: A crowd at a microscopic level: the famous scrambled crossing in Shibuya (Tokyo) stops vehicles in all directions to allow pedestrians to inundate the entire intersection.

Figure 2.3: A crowd at a macroscopic level: Thousands of African Muslims in Mecca (Saudi Arabia) for the annual pilgrimage known as the Hajj.

2.2 . Agent Complexity

Knowing what a crowd simulation consists of, we are ready to define one of the dimensions of our problem. That is the agent complexity, or how an agent is going to be represented by the simulation. On the one hand we need to choose a visual representation, or output, for an agent. On the other hand we need to determine what data is going to be stored for each agent instance.

2.2.1 . Representation

The simplest representation an agent can have is a point, just like a particle, representing the position of the agent. A better one is a disc, where the center is the position of the agents and the radius represents the area it occupies. A cylinder adds the height in a 3D world. We can attach an arrow to indicate what is the cylinder orientation. Another arrow can even model a vector whose direction and size represents the velocity of the agent. All these representations are basic and of a high-level of abstraction.

For humanoids, animals or characters having legs and feet, a more complex representation can be to add their footsteps. The resulting simulation will output the different footprints that the agent steps on. These footprints can be represented by a foot plant shape, a position and one orientation.

A higher complexity is achieved using an articulated skeleton or character to represent each agent. But this means we need to output the pose of that character at each time. Depending on the number of bones used to represent the agent, and also on how the poses are computed, the performance of the simulation should be highly affected. Finally, the more complex representation for an agent is too have a complete skinned mesh representing the agent and being simulated at all levels. That could mean for example that we could really detect collisions at mesh level, and therefore have physically accurate interactions. Again, the cost of such computations would strongly affect the simulation performance.

Figure 2.4 illustrates the axis of the agent complexity.

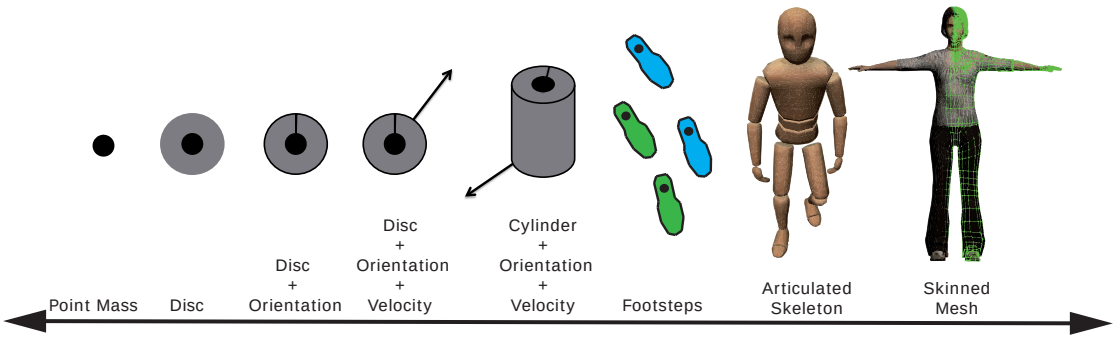


Figure 2.4: Axis of Agent Complexity

2.2.2 . Static Parameters

An agent can be accompanied of a series of static parameters defining its general and timeless condition. Think about parameters such as the radius, the height, the age, the maximum speed, etc.

2.2.3 . Variable Data

The data varying through time is the set of attributes that are affected by the simulation over time. The most basic is position and velocity (direction and speed), but an orientation should also be desired, provided that the representation is more complex than a cylinder. Depending on the simulation complexity we could have other aspects such as the energy, the fatigue, mental state, etc. that could affect the motion of the agent.

2.3 . Control Granularity

The second dimension of the crowd simulation problem is the control granularity, that is, how to control the movements of our different characters. The control granularity axis can go from computing only a reactive and local motion, to all possible behavior aspects, going through planning a global motion.

2.3.1 . Reactive and Local Motion

We call **steering** the behavior of an agent when simulating motion at local level. Basically that is to compute the velocity vector, with speed and direction of movement, that has the agent, and to modify it accordingly by reacting to the surrounding events such as possible collisions or pushes. The orientation of the agent is also important if we want to distinguish the direction of movement and the direction that the agent is facing. Adding orientation to the simulation might also imply simulate rotation, turns, pivoting actions and torques. The interactions that might happen at this level are purely reactive and may go from collision avoidance forces to pushes, forming queues, waiting, and even physical reactions.

An important distinction we should mention at this level is about how do we represent the surface or space where agents are moving: as a discrete local space or as a continuous space. A **discretization** of the local space, such as having a grid, reduces somehow the dimensionality of the problem as we can work on fixed units. We can easily know, for example, if a same unit of space is already occupied and collisions can be avoided. The problem of using a discrete space is that, depending on its granularity we do not allow agents to really get in contact. A **continuous** space might have a higher computational cost to be used, but it is more suitable to obtain effects such as pushes between agents as its movements are not limited (see Figure 2.5).

2.3.2 . Planning and Global Motion

The terms **planning** and **global motion** apply to a series of simulation algorithms that take place on a bigger scale than reactive and local motion. First of all as the term **planning** itself states, they plan a series of actions to be performed over time. These actions may come determined by the current state, a desired goal state, and the environment as well as other possible events. Therefore they usually take into consideration an abstract model of the environment such as a grid composed of cells or navigation meshes.

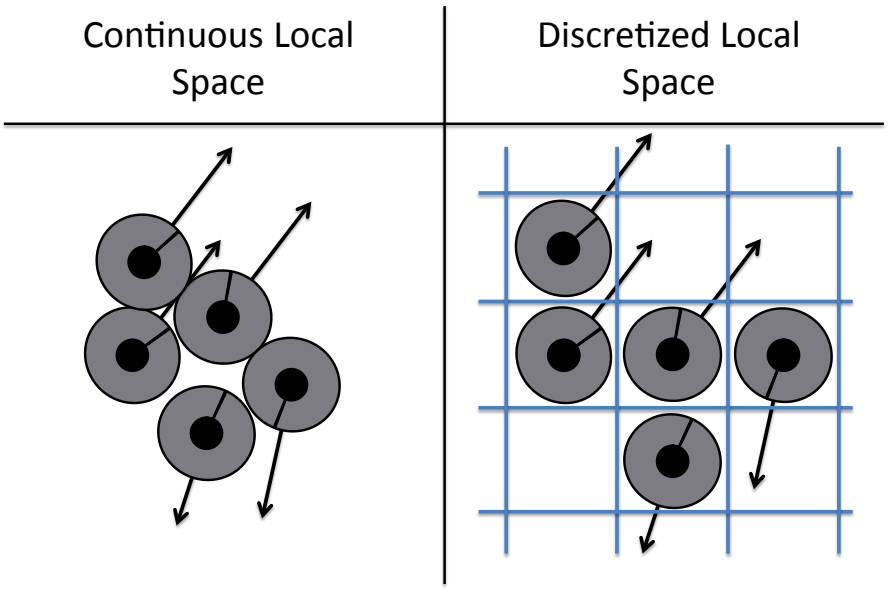


Figure 2.5: Continuous Vs. Discretized Local Space. With continuous space real contact and interactions like pushing behaviors can be possible

2.3.2 . Navigation Meshes

The agents of crowd simulations move around in a virtual environment, which is usually modeled in 3D. From the 3D mesh of the scene we can manually or automatically extract features such as rooms, doors, or moreover a navigation **graph** containing cells and portals, indicating the space where agents can walk over. The **navigation mesh** is this graph, and it is necessary to have it in order to efficiently compute plans to navigate from one place of the environment to another. The applied algorithms to plan are usually pathfinding algorithms, such as the well known A*. For example, our navigation mesh is composed of polygons, and an agent might need to go from its current position to another one. The planner will detect the polygons where those points lie into, and then use a pathfinding algorithm to output a sequence of polygons, or **waypoints**, conforming the plan or path of the agent (see Figure 2.6).

2.3.2 . Planner

A formal description of a **planner** requires that it works over a **graph** composed of **states** and **transitions**. The states can be the **nodes** of a navigation mesh or the composition of our articulated character, and the transitions are all the possible **actions** that the agent can perform. Depending on the granularity of

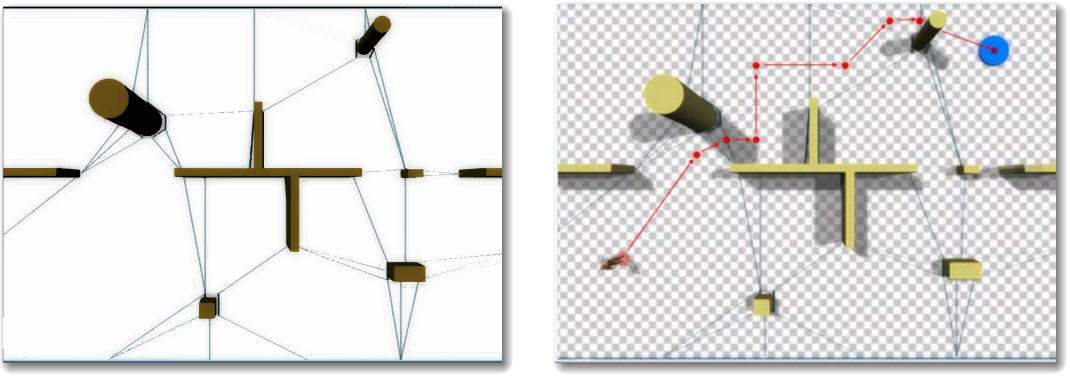


FIGURE 2.6: An example of a navigation mesh and a plan computed over it.

the planner these might go from moving to an adjacent cell to execute some action or playback an animation clip. But the algorithms are always essentially **pathfinding** algorithms trying to reach one state from another one by executing actions.

These algorithms usually work by and exploring the search space while expanding nodes, applying different transitions. Explored nodes are evaluated with a **cost function**, determining how much effort it takes to get to that specific state. An **heuristic** function is then needed to estimate the necessary cost to get to the final goal state. Heuristics can be as simple as an euclidian distance function, to any complex and high costly function.

2.3.2 . PREDICTION

When planning at a local level, only with a reactive behavior, an agent can predict the future position of an agent just by using its current velocity, and maybe other forces interacting with it. But it can not foresee abrupt changes in the direction, neither it can predict decelerations or accelerations. When planning at a global level we can compute the plans of all agents. Since a planner can have access to the other agent, it makes sense that it can access to their plans too. Therefore we can foresee and predict possible collisions between agents. Such a planner should be able to modify plans in order to avoid predicted collisions (see Figure 2.7). Interactions between agents can then emerge at a collaborative level, like wait for someone to come across, help another agent to perform an action, etc.

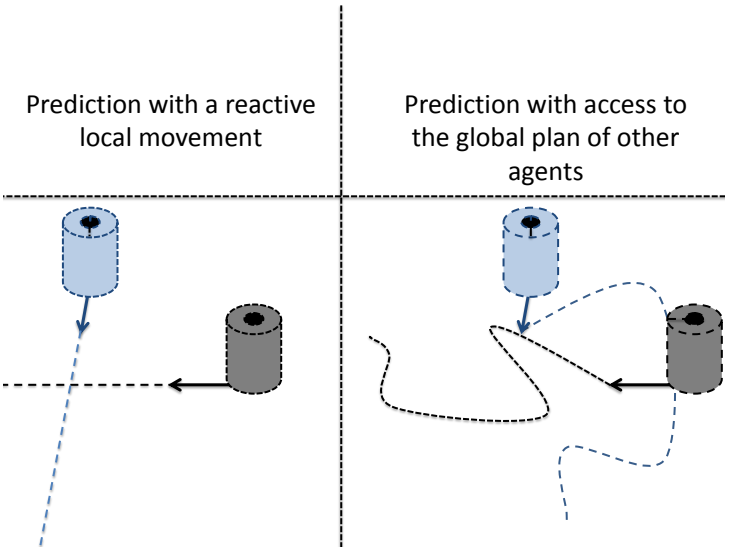


FIGURE 2.7: A collision prediction only by estimating the future position of an agent using its velocity vector, like in a purely reactive behavior (left). With global planning, agents can access to the real plan of other agents and make a more accurate collision prediction to modify their own plan (right).

2.3.3 . BEHAVIOR

Behavior refers to the decisions and the set of actions or reactions that agents can do or have in different situations. Simulating the behavior of a character is a higher level AI than just planning for a specific goal or reacting to something. It implies a mind state, a more complex goal and even strategies. Interactions between agents at this stage can be also more complex, with collaborative strategies for a common goal.

2.3.4 . COMPLETE CONTROL

A crowd simulation system can work with agents having just a local reactive motion, but they will lack the efficiency to find shortest paths to get a specific goal, and the exhibition of a a specific behavior. Depending on how the global planning is designed, including predictions and collision avoidance, a local motion might not be necessary. Although if we just plan waypoints to go through different cells, a local steering behavior is sufficient to navigate within those cells. A behavior simulation is needed when we want to give agents specific goals and particular states at a higher level. Therefore, although these different

levels of control granularity should be able to run a crowd simulation independently, they are extremely connected and should be integrated and run in a same system.

Figure 2.8: Axis of Control Granularity

2.4 . Environment Complexity

The environment is an essential part of the simulation. It limits and conditions the actions of the agents. This is therefore another dimension of our problem where the environment can be either static or dynamic.

2.4.1 . Static

A static environment can be preprocessed. As we have already explained, navigation meshes can be the output of this preprocess and contain all the information about the navigable surface. Static obstacles can be added to the scene, and since they are not moving their collisions can be easily avoided at any time.

2.4.2 . Dynamic

An environment can be dynamic when obstacles move in it. In fact, other agents can be considered as dynamic obstacles. But depending on how the simulation is carried out, their treatment would be different.

Deterministic obstacles are those whose plan or animation curve is known, being possible to predict their position at any given time. Collision prediction is therefore possible and we can plan with collision avoidance.

Undeterministic obstacles are unpredictable, and therefore its effects need to be computed and updated at every step. They can only be handled by reactive behaviors and physic reactions.

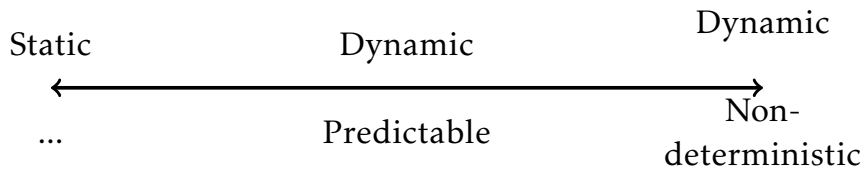


Figure 2.9: Axis of Environment Complexity

2.5 . Time Discretization

As the definition says, a crowd simulation represents the operation of the system over time. But how do we discretize time is essential for the output of the simulation. If, for example, our steering behavior works with velocity vectors, and our agents change their position multiplying their velocity by the elapsed amount of time in the simulation step, it is clear that the resulting position is strongly dependent on that elapsed amount of time.

Essentially, if the **time step** is too big the simulation will be less prone to changes and agents will be less agile, meaning that collisions will be more probable and inevitable. If the time step is too small, you may end up carrying out repetitive and unnecessary computations, as the state of the simulation might not have significant changes between consecutive time steps. A good choice for the time step duration is therefore required. In our case we can link this directly to the animation and real-time rendering requirements. That is, for a local steering, we will desire a time step corresponding to the time between two rendered frames, meaning that our simulations should be carried out for each frame.

In fact, for a real-time simulation, the different processes associated to the different controls of granularity, should be executed for every time step. But then, these different levels of control could have different time steps. We do not have the same requirements for a local steering behavior than for a global planning.

For example, computing a local steering motion can be done for every frame, but the global plan of an agent does not need to be replanned every frame. It may just need to be replanned in case that the environment has significant changes, or if many unpredicted agents or obstacles are now in its field of view. Therefore you could set the time step of the global planning to a higher value, like 3 seconds for example. At the other extreme, collision avoidance might need a time step higher than the frame rate of the application. Imagine an agent or an object moving at high speed. If the time step is too big, a collision between two consecutive steps may exist, but it will remain undetected (see Figure 2.10).

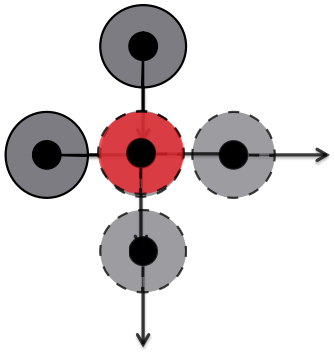


FIGURE 2.10: Two agents at two consecutive positions of a simulation. The time step for the collision check is too small and the collision is not detected.

2.6 . ANIMATION

Our second big concept to address is **animation**. According to Wikipedia: *"Animation is the process of creating a continuous motion and shape change illusion by means of the rapid display of a sequence of static images that minimally differ from each other. The illusion (as in motion pictures in general) is thought to rely on the phi phenomenon.* [...] *Images are displayed in a rapid succession, usually 24, 25, 30, or 60 frames per second."* In our case the meaning of this implies we need to show the motion of the agents at at least 24 frames per second. Therefore, as we have previously mentioned, the simulation time step should be set at maximum to 1/24 seconds.

2.6.1 . Frames and Keyframes

What we call a **frame** throughout this document is each one of the images displayed by the system in order to create the motion illusion. In a stored animation, a **keyframe** is a specific configuration or state for a particular instant of time. As in simulation, to store a motion time must be discretized. Animation data is therefore stored for a discrete set of instants, and a keyframe is all the animation data corresponding to one of those instants having it.

2.6.2 . Blending Between Keyframes

Note how the display rate does not need to necessarily match the sampling rate of an animation. That means, when playing and visualizing an animation, we might not have to display the same number of frames than the number of keyframes that our animation has. In both cases time is discretized, but at different time steps.

If we have more keyframes than frames we need to display, we can just choose the closer keyframe to the corresponding time of our current frame. If we have less keyframes than frames, which is more usual, we need a way to display inner frames between keyframes. That is usually done with some kind of interpolation technique, which is known as **blending** between keyframes.

2.6.3 . Character Animation

So far we have just talked about animation in general. In fact, if we are going to represent agents with just a point, a disc, a cylinder or any other representation, we do not need to worry about keyframes or any other element about stored animations. In these cases we only need to worry about translation, and maybe rotation, to move our agents. But if we choose an articulated 3D character to display our agents, we will want to animate them, and this is where character animation enters the scene.

Note how in this section we do not speak about mesh deformation, since we will include this in the rendering section. In fact, most of the related work in

character animation is done without taking into account any mesh or specific character. They usually abstract and illustrate their results using some standard skeleton or stick figure. Therefore the mesh deformation concepts will be explained in the corresponding rendering section.

2.6.3 . Skeleton

The most extended way of animating a character is to have it composed by an articulated **skeleton**. This skeleton is usually a hierarchy of **bones** or **joints** (see Figure 2.11), where the transformation applied to one of the joints is recursively applied to all of its children. Those transformations are usually encoded using matrices or quaternions.

From these bones or joints, we distinguish the most important one, which is the **root** of the hierarchy. In humanoid characters, or biped characters, the root is usually placed at the hips. The root is of a particular importance since it is usually the bone that will guide the global movement or motion of the whole skeleton, as all the other joints are hanging from it.

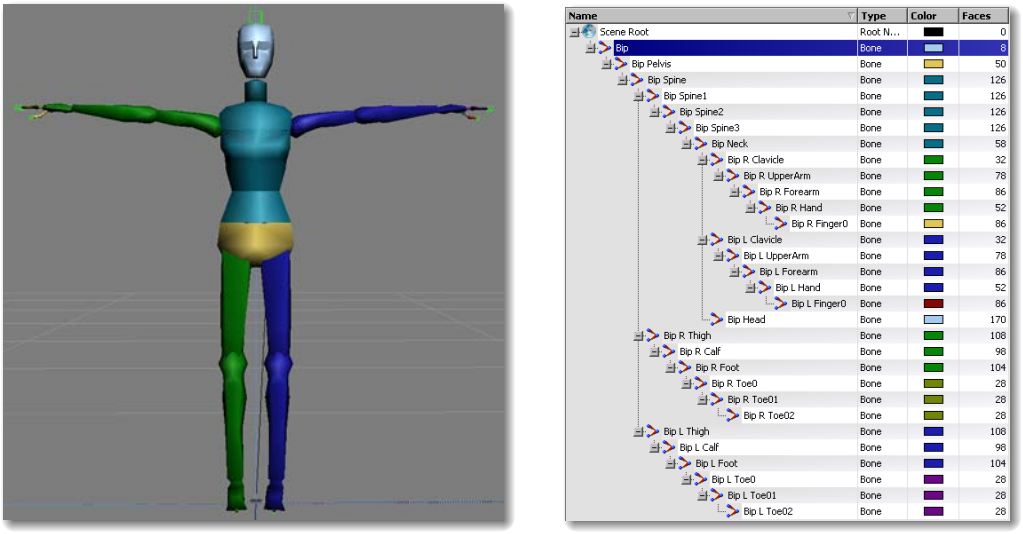


Figure 2.11: A biped skeleton (left) and its hierarchy of bones (right).

2.6.3 . Pose

A **pose** is a complete configuration of the skeleton, that is the whole set of transformations for all the skeleton joints (see Figure 2.12). In an animation

each keyframe might encode a pose, although it is possible that for compression purposes, a keyframe only stores the joint transformations that have changed from the previous keyframe to the current one.

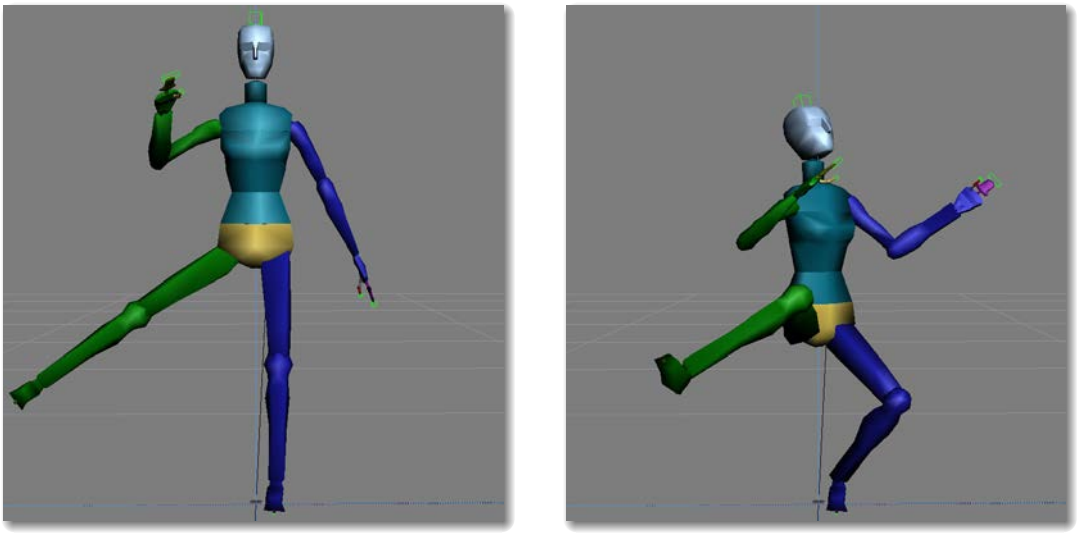


Figure 2.12: Two different poses of the same skeleton.

2.6.3 . Blending Between Poses

In order to create new poses and new animations it is possible to mix different poses. We can easily interpolate two poses by giving different weights to their corresponding transformations and adding them together, although the sum of weights should be one. This operation might be repeated with other poses from other animations, so we can have a completely new pose generated from existing ones. This is what we call **blending** between poses.

2.6.4 . Animation Clip

An **animation clip** is a set of pairs $\langle keyframe, time \rangle$ each one with a pose of the skeleton of the character. Animators work using animation software to manually create them. This is a hard and tedious work usually performed by an **artist**. Another possibility is to obtain animation clips using motion capture techniques. **Motion capture** systems capture the real poses of actors at high rates to obtain highly realistic and natural motions.

2.6.4 . Transitions

Sometimes we will want to play one animation after another. Although an animation clip presents continuity between its own keyframes, the final pose of one animation clip does not have to be the same as the starting pose of the another clip. Moreover, both clips could have different root positions. Therefore continuity between both clips is no trivial.

A smooth **transition** can be generated between both clips to ensure continuity between them. A transition can be a new animation clip explicitly created for a pair of clips. But a transition can also be achieved by blending for a short amount of time (a second or less) the two particular poses that need a smooth transition. Although this can be a good solution when the poses are quite similar, and requires no extra effort in designing new clips, this can introduce problems when the poses are very different.

The major problem occurs at the level of the feet. If a foot is supposed to remain still on the ground, and we perform a simple blending between two poses without taking this into account, the foot might slide on the floor to reach a different foot position. This is an unpleasant effect known as **foot sliding** or foot skating. Thus additional efforts and techniques should be applied in order to avoid such undesired effect.

2.6.4 . Cycling Clip

We call a **cycling clip** an animation clip whose end pose is equal to its initial pose, thus making it possible to easily produce an play continuous loop of the same animation clip without noticing any discontinuity. Cycling clips are very useful to perform character control, which we will explain in the next section.

2.6.4 . In Place Animation

We call an **in place** animation the one where the root has been removed of its forward translation. That is, the character remains at the same Z coordinate during the whole animation clip. This kind of clips can be useful for a character control where we want to have full control of the translation of the character's root.

2.7 . Character Control

By **character control** we mean the methods and techniques used to control the motion and actions of a virtual character by animating it. To do so we can use a database of animation clips, we can use inverse kinematics (IK) controllers, or even physically based procedural methods. The goal is to produce an animation to represent and reflect a desired input or behavior, such as a motion trajectory.

2.7.1 . Crowd Animation Synthesis

To synthesize the animation of a crowd means to control characters to reflect the movement of the agents in the crowd simulation. Therefore the input of the crowd animation synthesis will be the output of the current crowd simulation.

As we have previously said, the most basic crowd simulation will output for every agent a position. If no other information is given the animation system will need to compute and maintain at least a velocity vector. But it is most likely that the crowd simulation already gives a velocity vector and an orientation. The output that must produce the crowd animation synthesizer is a continuous animation that reflects the motion of the agent without artifacts. The problem in this case becomes how to synthesize an animation from just a root trajectory. If we have an animation database we can use to blend the available clips and obtain a motion that accurately follows the root trajectory. The output in this case would be the blending weights for each clip to synthesize the current pose of the character. If we do not have an animation database, we might need to use some other methods to animate our character, such as IK or procedural methods. In the case of having time foots prints, or a footstep trajectory, the same applies in order to obtain our desired motion. The difference is that we will have more constraints when synthesizing the motion.

2.7.2 . Animation Preprocess

When working with a database of animation clips, we need a way of knowing what are the properties of each one when applied to a character. For example, an animation clip has an inherent data such as the duration of the clip and the poses of the character, but we would like to know at what average speed the character is moving on during the clip. Fortunately animation clips can be analyzed in a preprocess phase in order to extract such data. Usually the most relevant data would be the root velocity, the orientation of the character, the turning velocity, a classification of the type of motion (walking motion, running, jump, etc.). In the footsteps case we might also want to extract the footsteps information of the clip.

2.7.2 . Angle of Movement Vs. Orientation Angle

It is important to remark, since these concepts are often repeated in this thesis, that there is an important difference between the **angle of movement** and the **orientation** of an agent. The **orientation** can be expressed as the angle between one axis of the world coordinate system (usually the $\vec{X}$) and the direction vector that the agent is facing. Since the moment our character has more than a disc to represent it, this is necessary to render it correctly. On the contrary, the **angle of movement** is the angle between the orientation angle and the velocity vector that the agent is following. This is the common way to express it, meaning that an angle of movement equal to 0^o is equivalent to move forward, and an angle of movement equal to 90^o is a side stepping motion to the left. Figure 2.13 illustrates this difference.

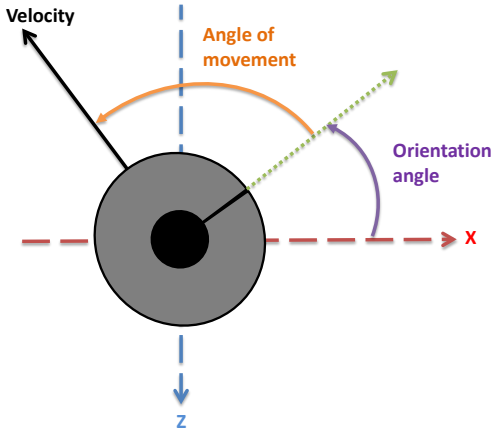


FIGURE 2.13: The orientation is the angle that the agent is facing, while the angle of movement is the velocity angle with respect to the orientation.

2.8 . ANIMATION QUALITY

Having introduced most of the animation concepts, we can therefore see that in our problem of visualizing real-time crowds simulations with animated characters, the achieved animation quality represents another dimension of our problem. It is not the same to use handmade animations than motion captured ones. It is not the same to fulfill physical constraints such as footsteps than to ignore them. It is better to have seamless and continuous transitions than to present abrupt changes and discontinuities in the motion of a character. And also, an animation will not have the same quality depending on the granularity of the skeleton used. A different number of joints, can have different visual qualities but also different performances. We must wonder if we need in a crowd simulation to model the fingers or toes of the characters, or even if we need to have complex facial animation.

2.9 . Rendering

Although the term **rendering** is most related to achieving a highly realistic image, in the case of crowd rendering it is commonly equivalent to **visualization**. This visualization can be done by simply rendering a point per agent, a disc, a cylinder or any static 3D model. But ultimately we want to visualize the output of a crowd simulation by rendering animated 3D characters. An agent will be therefore represented by one of these characters, also known as an **avatar**.

2.9.1 . Mesh Deformation

As we have mentioned previously, the most extended animation approach is **skeletal animation**. This requires to have our animations created for a specific skeleton. We therefore need a method to transfer those animations from the skeleton to our 3D mesh.

The first step we do to animate a mesh of a 3D character is to create a skeleton and fit it into the 3D mesh. This is known as **fitting**. Secondly we need to attach our mesh to the skeleton. This is done by deciding which bones of the skeleton will have an influence over which vertices of the mesh, and assigning a transformation weight for each influence. This process is called **rigging** (see Figure 2.14). **Skinning** is the technique which transforms each vertex of the mesh. This is done every frame by adding, with the corresponding weights of the rigging, the transformations of the influencing bones from the animation.

2.9.2 . Level of Detail

When a character model is rendered at a far distance it will cover a small size in pixels on the screen. This means we will see it with less precision and that it can be replaced with a lower quality version without noticing it. This is the principle of **level of detail (LOD)** (see Figure 2.15). In the case of crowd rendering, using LOD presents some problems or challenges, such as how to generate good low resolution meshes that can be correctly animated, when to switch from one

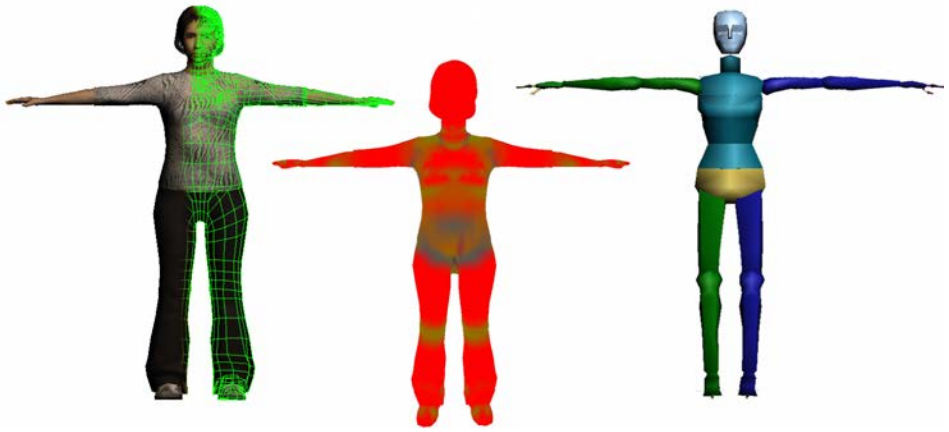


Figure 2.14: Rigging: a character is composed by a 3D mesh representing the skin (left) and a skeleton (right). The rigging process assigns each vertex of the mesh to one or more bones of the skeleton with a weight (center). Vertices influenced exclusively by one bone are represented in red, and other colors indicate vertices that are influenced by several bones.

to the other (at what distances), and how to do it without producing popping effects.

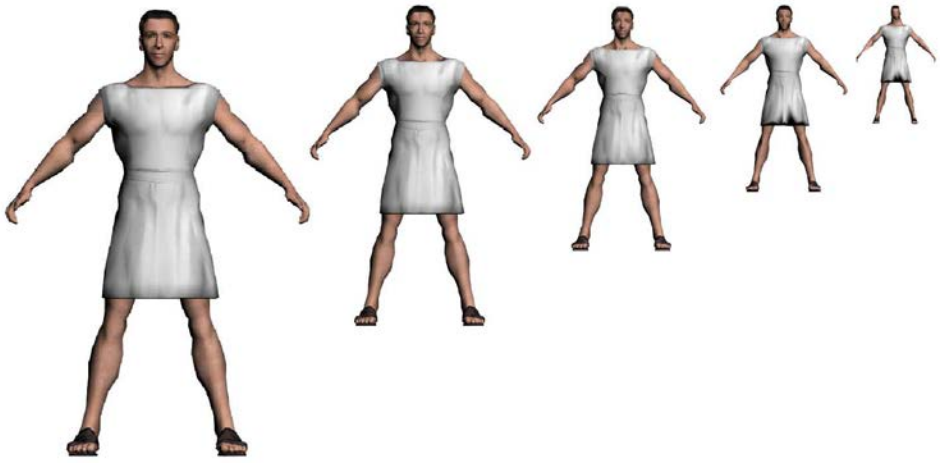


Figure 2.15: Five models of the same avatar with decreasing number of polygons as they are placed further away.

As we will see in this thesis, a well known technique for LOD is to use **impostors**, that is replace the 3D animated mesh by another kind of structure to fool the user. These structures can be image-based, like texture quads, or even point-based. Finally, systems are known as **hybrid** systems when they combine the use of geometry mesh for close up agents, and the use of impostor for far away characters.

2.9.3 . Impostor

Impostors are useful to fool the viewer when characters are far away from the camera and we want to avoid the computational cost of rendering complex geometry. Impostors are usually simple quads textured with an image of the rendered object, but they can be more complex in their construction, although their rendering should remain more efficient than the original geometry.

2.10 . Visual Quality

Having introduced some of the rendering concepts, we have to add to our global problem the visual quality dimensionality. 3D characters are modeled everyday with more and more polygons, and reducing their number while maintaining its quality is not straight forward. Moreover, scaling the crowd and rendering thousands of high quality models can reach the performance limits even of modern GPU cards. If models have complex shapes and are animated the problems becomes harder to resolve. Finally, applying special effects such as lighting, shadows, cloth simulation or hair, increases the necessary resources needed to render these crowd scenes. There is therefore a new axis were the visual quality goes from a simple rough rendering to a completely realistic rendering, using light shading, shadowing, and even cloth and hair simulation rendering.

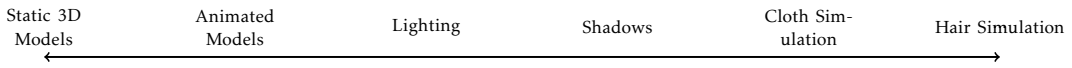


Figure 2.16: Axis of the visual quality

2.10.1 . Uncanny Valley

The uncanny valley, from the field of human aesthetics, corresponds to the hypothesis that humans respond negatively and with repulsion to the view of

robots or virtual characters with human features looking and moving almost, but not exactly, like natural human beings. There is a graph of the comfort level that human viewers have, as a function of the familiarity or acceptability we have versus the human likeness of the virtual character. The "valley" refers to a region of this function close to the maximum human likeness [Mori et al., 2012] (see Figure 2.17).

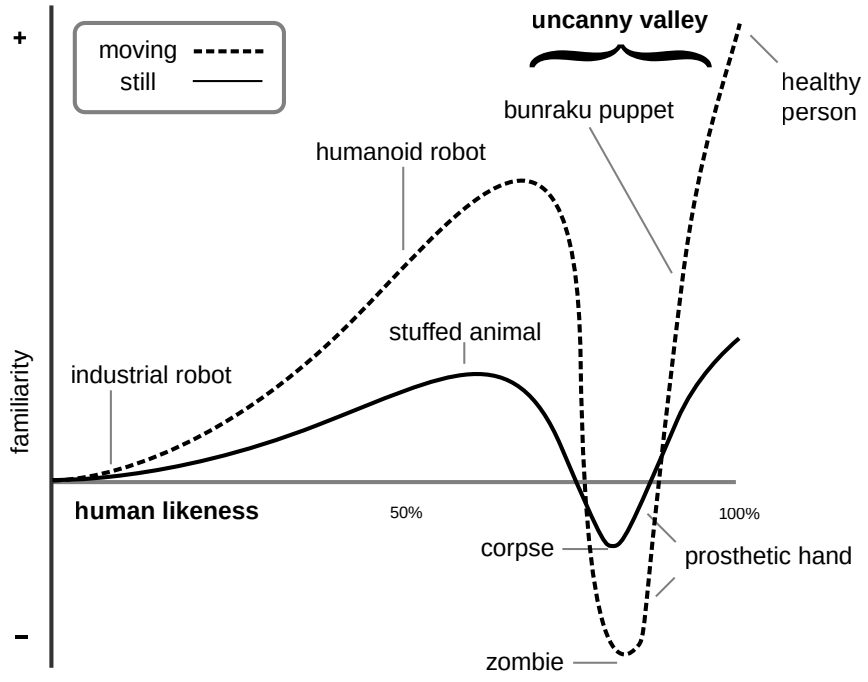


FIGURE 2.17: Hypothesized emotional response of human subjects is plotted against anthropomorphism of a robot, following Mori's statements. The uncanny valley is the region of negative emotional response towards robots that seem "almost human". Movement amplifies the emotional response. [Mori et al., 2012]

This phenomenon is well known in robotics, and moreover in computer animation. Important computer animation movies with high realistic characters, such as *Final Fantasy: The Spirits Within* from Square Pictures, or *Beowulf* from Warner Bros Pictures (see left of Figure 2.18 had a bad response from public. Recent films try to avoid this by keeping a more cartoonist style, which has been proven to have a more pleasant response from the audience, like in *The Adventures of Tintin* from Amblin Entertainment (see right of Figure 2.18).

Addressing the problem of crowds in real-time we are still quite far from the uncanny valley problem. These high level of realism are by now only reached

FIGURE 2.18: **Left:***Beowulf* from Warner Bros Pictures, had a bad response from public falling into the uncanny valley. **Right:** *The Adventures of Tintin* from Amblin Entertaintment, adopts a more cartoonist style which had a better response from the audience.

by the movie industry, whereas the real-time virtual characters used in videogames are not as detailed, since it would greatly affect performance. Models, shading and rendering techniques are improved every year, getting us closer to the quality of the computer animation pictures of some years ago. Eventually it will be possible that our simulations, due to the rendering quality, or maybe the animation and simulation, start falling into the uncanny valley of unpleasant response.

2.10.2 . VARIETY

Another important aspect of crowds is the individuality of each agent. Whereas it is the behavior agent, the animation or the appearance of the character, each individual should be different and have particular characteristics. Ideally, in our crowd simulations, there should be no clones, or a minimum amount of them, as they are easily detectable for the viewer. This introduces a new axis for our problem, which is the axis of variety (see Figure 2.19). The main problem with variety is that adding more variety usually comes with a higher consumption of memory resources.

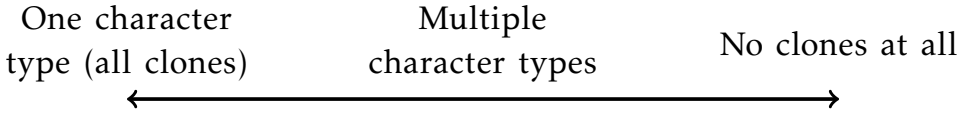


FIGURE 2.19: Variety axis

2.11 . Real-Time Crowds

All the concepts we have seen until now cover simulation, animation and rendering of one or more agents, but we have not talked about their integration and the new dimensions that are added to the problem when dealing with real-time crowds.

2.11.1 . Scale

The first new dimension we have to add is the scale of our problem. Our simulation, rendering animated characters, must be performed for a crowd, so we introduce a new axis going from one to many agents, taking into account that normally more agents implies more computation time.

In this thesis we will consider a crowd a multi-agent system going from several tens of agents to an order of magnitude of thousands of agents. Although a crowd could also imply several thousands of millions of agents, these would fall in the category of macroscopic simulations, and here we are focused on microscopic simulations.

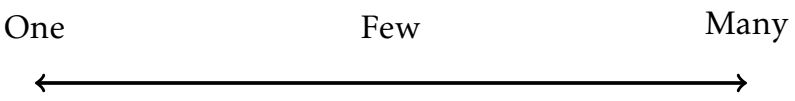


Figure 2.20: Scale axis

2.11.2 . Performance

Directly related to this we have a second dimension which is the performance. Our whole system can go from running offline, which usually means achieving less than 12 frames per second (fps), to running in real-time (25 fps) and even in stereo for a virtual reality device (50 fps, meaning we need to perform two renders per frame).

FIGURE 2.21: Performance axis

2.11.3 . INTEGRATION, GLOBAL COHERENCE AND CONSISTENCY

Finally, in our crowds problem, the system needs to integrate all of these elements, simulation, animation and rendering, in such a way that we visualize animated characters representing the output of a real-time simulation. As the reader might have sensed, this is not straight forward as there are many elements that have to be taken into consideration. Moreover, these tree areas can interact and collaborate but at the end of the day we want the final result to be coherent and consistent with the final simulation (see Figure 2.22).

For example, a simulation module can output a state to the animation module, which can synthesize animations, but in the process it might be necessary to perform some adjustments to the positions of the agents. In such case the animation module should feed back the modifications to the simulation module in order to maintain consistency between both modules. Another example is a simulation system that works using as actions the animations clips. In this case both the simulation and animation could be performed by the same module.

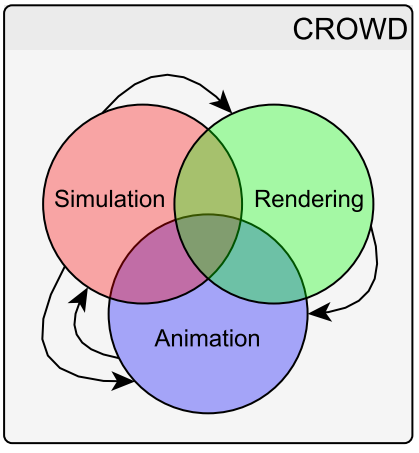


FIGURE 2.22: Simulation, animation and rendering of crowds are three overlapping research areas dependent on each other which are continuously interacting.

In any case, the important thing to note here is that in performance, the scalability of the problem affects all the areas as the crowds increases its size. Therefore bottlenecks can appear in any of them, requiring optimizations and the development of novel techniques that are able to outperform current ones. We must keep in mind that the final goal is to have a more realistic simulation, animation and rendering, with the highest number of agents, and running as fast as possible.

2.12 . Problem Statement

Throughout this chapter we have presented different dimensions or axis of complexity that our crowd problem presents. Here is a summary of all the dimensions of the problem to show its magnitude and why it is necessary to work on so many different aspects.

- + Agent Complexity
- + Control Granularity
- + Environment Complexity
- + Animation Quality
- + Visual Quality
- + Variety
- + Scale
- + Performance
- + Global coherence and consistency

The ideal outcome would be to have the maximum possible values in all these dimensions at the same time. Unfortunately, pushing the boundaries of one axis might compromise the others, specially the performance one. Therefore,

as we will see in the state of the art sections, as well as in our contributions, thresholds will appear to regulate all of these aspects.

3 . State of the Art

Now that most of the concepts of this thesis have been introduced, we present in this chapter the state of the art in the different areas of crowd research. We start with an overview of crowd simulation models, data-driven techniques and planning solutions. Then we introduce most of the character animation techniques to synthesize crowd motion, using root velocity or footstep driven. We follow with a complete study on real-time crowd rendering, including skinning, point-based, image-based, culling, and LOD techniques, and hardware improvements. Finally we present some of the existing tools for research on crowds.

3.1 . State of the Art on Crowd Simulation

In this section we review the state of the art on crowd simulation. First, we give a small overview on macroscopic models. Secondly, we focus then on the local motion in microscopic models. Third, we talk about data-driven methods. Fourth, we talk about computing high-level paths and planning global movement. And finally we present some previous work using multiple resolutions for the planning problem.

3.1.1 . Macroscopic Models

Macroscopic models consider crowd behaviors as flows and do not take into account individual behaviors. These models are more to be applied to compute traffic simulations [Sewall et al., 2010] or the capacity of large building structures such as stadiums, conference centers, etc. (see Figure 3.1) [Narain et al., 2009]. Also, they are not usually in real-time as they need to do a lot of computations.

Figure 3.1: 80,000 people on a trade show floor. [Narain et al., 2009]

Regression models are based on statistic relations between flow variables. This way they predict under specific circumstances pedestrian flow operations, which also depends on the infrastructure (stairs, corridors, etc.) [Milazzo et al., 1998].

In *Route choice models*, agents find their path based on the concept of utility. The best way is defined by trying to maximize the utility of their destinations [Hoogendoorn, 2003].

Using Markov chain models, *Queuing models* describe the movement of the agents from one node of the network to another [Løvås, 1994]. This nodes are usually rooms and links are portals or doors. Markov chains work with a set of states and transition probabilities.

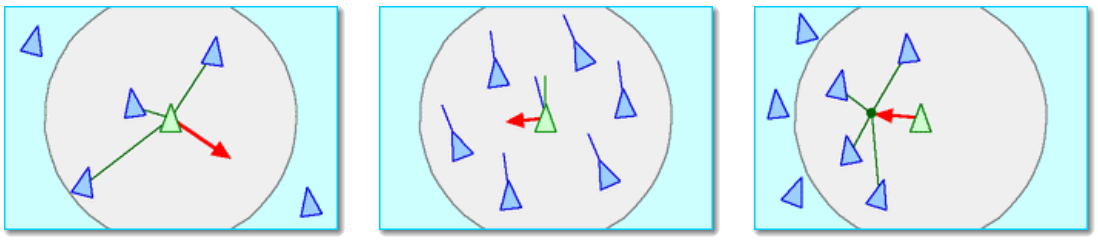
We can also find *Gas-kinetics* models where they compare crowds with fluid or gas dynamics, so they use their theories to determine how crowd density and velocity must change over time using partial differential equations [Henderson, 1971].

3.1.2 . Microscopic Models and Local Movement

Microscopic models work on an individual scale, thus computing for each agent at each time where he is. We divide them in different approaches, the difference between them being in their agent representations, their local movement functions, and the discretization of space and time.

3.1.2 . Rule-Based Models

One of the earliest methods proposed for local movement with collision avoidance is the Boids algorithm [Reynolds, 1987, 1999], where movement is achieved for low-and-medium density crowds in a flocking or swarming style, using three simple rules represented by three forces affecting each agent: a force pushing it away from the location of its closest neighbors (see Figure 3.2 (a)); a force aligning it's velocity to that of it's neighbors (see Figure 3.2 (b)); and a force drawing it towards the average position of its local neighbors (see Figure 3.2 (c)). At each time, the sum of the forces from all three rules determines the next velocity for each agent.



(A) (B) (C)

FIGURE 3.2: The three rules of the Boids algorithm. (a) Separation: steer to avoid crowding local flockmates. (b) Alignment: steer towards the average heading of local flockmates. (c) Cohesion: steer to move toward the average position of local flockmates. [Reynolds, 1987, 1999]

A problem of *Rule-based models* is that they lack the contact between agents and therefore do not present any "pushing" behavior. So they are conservative and avoid contact as long as they can, and when necessary (with high densities) they apply "wait" rules for agents. Rule-based models can also be combined with cognitive models [Shao and Terzopoulos, 2005]; or different rules can be applied to the crowds, groups or individuals to achieve more realistic global behaviors [Thalmann et al., 1999].

3.1.2 . Social Forces Models

In *Social force Models* [Helbing et al., 2000], real forces such as repulsive interaction, friction forces, dissipation, and fluctuations are modeled as "virtual" social forces. For each agent Newton's equations of motion are solved, and the model is fast enough to be successfully applied in real-time.

Although *social force models* are designed to be as simple as possible, compared to other models, its pedestrian behavior can be more realistic. Each agent is represented by a circle (with its own diameter) in the locomotion plane, and the model gives coordinates and velocities in continuous space, as well as interactions with other objects. Human crowd behavior is modeled mixing sociopsychological and physical factors.

Helbing's model [Helbing et al., 2000] is the most known social forces model. Applying repulsion and tangential forces to simulate the interaction between people and obstacles, it allows for realistic "pushing" behavior and variable flow rates (see Figure 3.3).

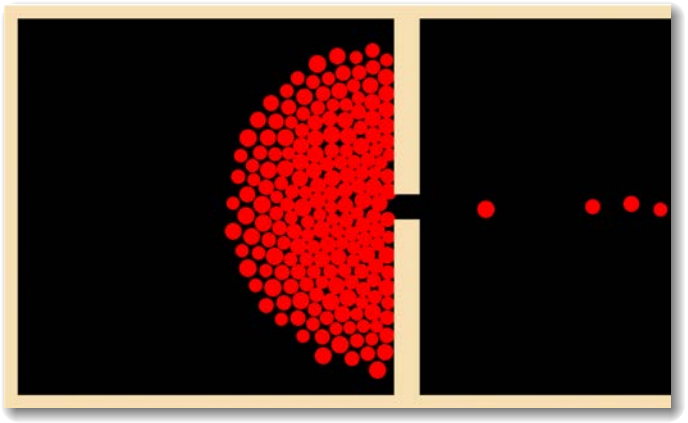


FIGURE 3.3: Simulation of pedestrians moving with identical desired velocity towards the 1m-wide exit of a room of size 15m X 15m. [Helbing et al., 2000]

For low density crowds some work was done using particle simulation approaches. The motion of groups with significant physics can be modeled using a particles system and dynamics [Brogan and Hodgins, 1997]. Finally, the social forces model can be extended to include individualism [Braun et al., 2003].

Extensions have been done by combining psychological and geometrical rules with a social and physical forces model, being able to handle high density crowds. *HiDAC* [Pelechano et al., 2007] exhibits a wide variety of emergent behaviors from agent line formation to pushing behavior. It also handles different situations and personalities, like impatient individuals avoiding bottlenecks, or a panic situation with agents pushing they way through the crowd (see Figure 3.4).

3.1.2 . VELOCITY-BASED MODELS

Velocity-based models take ideas from robotics and work with *velocity obstacles*, commonly abbreviated VO, which is the set of all velocities of an agent that will result in a collision with another agent, assuming that this last one maintains its current velocity. If a velocity inside the velocity obstacle is chosen, then the two agents will eventually collide; otherwise such a collision is guaranteed not to occur [Fiorini and Shillert, 1998]. Therefore agents are not passively moving, but reacting to each other, which results in oscillations. *Reciprocal Velocity Obstacles* (RVO) Models [van den Berg et al., 2008] avoid these oscillations by implicitly assuming that the other agents make a similar collision-avoidance reasoning. Then, instead of choosing a new velocity outside the VO, they take

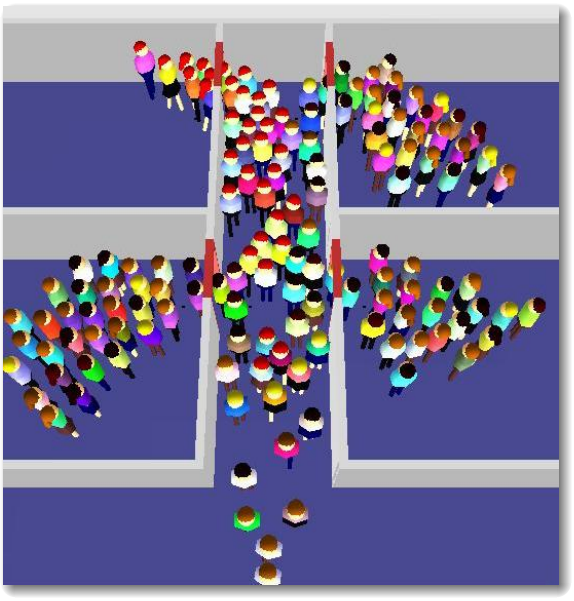
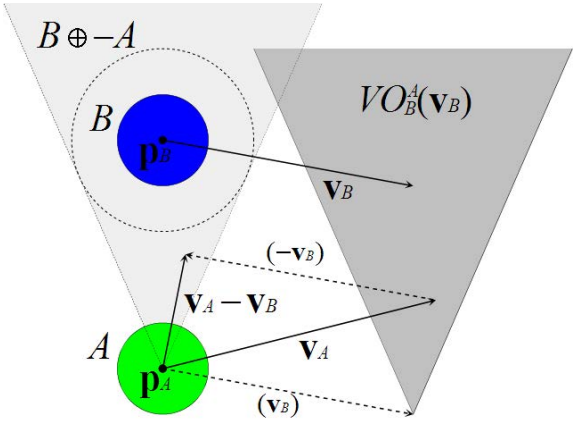


FIGURE 3.4: Red-headed people exhibit panic behavior and push others to open their way through the crowd. [Pelechano et al., 2007]



$$\gamma(p, v) = \{p + tv \mid t > 0\}$$
$$VO^A_B(v_B) = \{v_A \mid \gamma(p_A, v_A - v_B) \cap B \oplus -A \neq \varnothing\}$$

FIGURE 3.5: Reciprocal Velocity Obstacle (RVO) avoid collisions between agents by assuming they all reach to each other in the same way, and taking the average between the current velocity and the one of the Velocity Obstacle (VO). [van den Berg et al., 2008]

the average of a VO and the current velocity (see Figure 3.5). A difference between velocity-based models and social forces models is that, while the first are simple to implement but difficult to tune, the second are complex to implement but are collision-free guaranteed.

3.1.2 . Continuum Dynamics Models

In *Continuum Crowds* [Treuille et al., 2006], a model based on continuum dynamics is presented. Instead of working on a per-agent basis, they view motion as a per-particle energy minimization, and adopt a continuum perspective on the system. Using a dynamic potential field, they can simultaneously integrate global navigation with moving obstacles, such as the other agents. This allows them to efficiently solve the motion of large crowds without explicitly avoiding collisions. This model exhibits important emergent phenomena observed such as people forming lanes, or crossing groups of people forming vortices (see Figure 3.6). The problem of continuum crowds is that they trade-off the individuality for performance. The variability between agents is lost at the expense of having a real-time planning of optimal behavior, with minimal computation per agent.

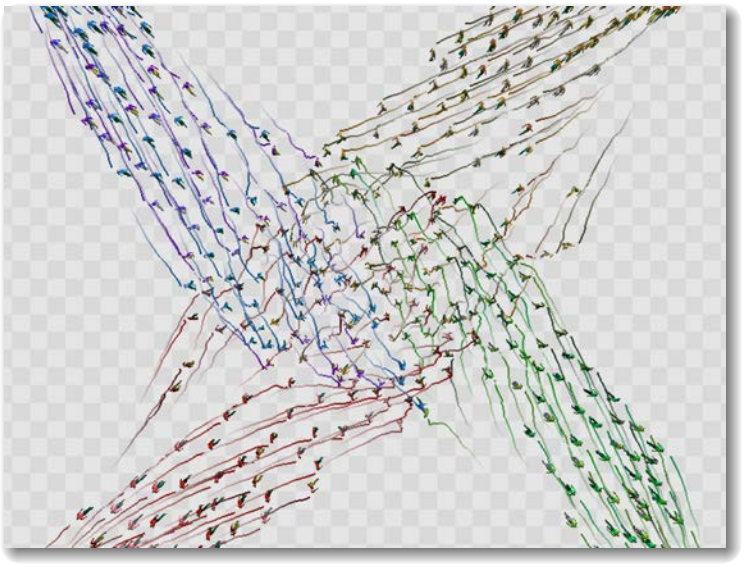


Figure 3.6: A vortex forms as four groups cross. [Treuille et al., 2006]

3.1.2 . Cellular Automata Models

In *cellular automata models* [Chenney, 2004] space and time are discrete and physical quantities take a finite set of discrete values. A cellular automaton is formed by a regular uniform lattice or grid with one or more discrete variables at each cell (see Figure 3.7). The values of these variables define the state of each cell, and are affected in discrete time steps by the previous values of the adjacent cells. Also, these updates are made simultaneously according to a set

of local rules [Wolfram, 1983], which define the decision-making behavior of the automata. The global group behavior emerges then as the result of the interactions of the local rules that each agent applies in its neighboring cells.

Although there are a lot of *cellular automata models* fast and simple to implement [Chenney, 2004, Tecchia et al., 2001, Torrens, 2007], they do not allow for contact between agents. Since space is discretized, individual agents can only move to cells that are adjacent and free. Realistic results are then obtained for low density crowds, but not for high-density crowds where pushing behavior is necessary and therefore discrete cells are not a valid approach. We can achieve more realistic paths in the grid using precomputed paths toward goals and storing them [Loscos et al., 2003].

Figure 3.7: A frame showing people moving through a city, driven by a flow tiling of the streets. Internal boundary conditions prevent people from walking through building walls. [Chenney, 2004]

3.1.2 . Footstep-Driven Approaches

Some approaches have tried to change the simulation paradigm by using more complex agent representations, such as footsteps. They can be physically based but generated off-line [Felis and Mombaur, 2012]. Or they can be generated online from an input path computed by a path planner [Egges and van Basten, 2010]. Footsteps can also be planned using a 2D approximation of an inverted spherical pendulum model of bipedal locomotion [Singh et al., 2011]. Figure 3.8 illustrates this.

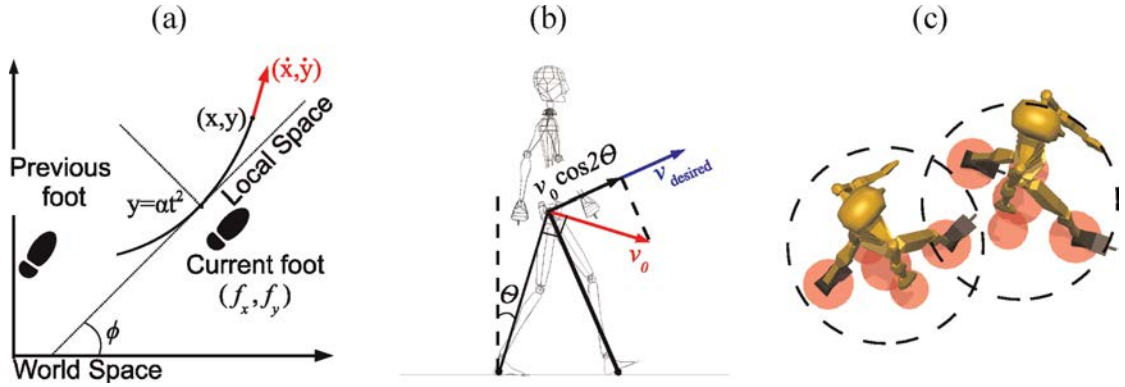


FIGURE 3.8: A footstep navigation model. (a) Depiction of state and action parameters. (b) A sagittal view of the pendulum model used to estimate energy costs. (c) The collision model uses five circles that track the torso and feet over time, allowing tighter configurations than a single coarse radius. [Singh et al., 2011]

The footsteps approach allows for better collision bounds, having characters getting closer to each other, in contrast to a simple collision radius model. It also allows for a better fitting and cooperation between characters at doorways or narrow corridors.

3.1.2 . ANIMATION-DEPENDENT PLANNERS

Microscopic models can also be classified into two main sets based on whether they only focus on calculating the position of the root ignoring the animations, or whether they plan respecting the underlying animations. The first set focuses on simulating realistic behaviors regarding overall character navigation and do not worry about animations. In fact sometimes their goal is to simply model agents as cylinders that move around a virtual environment avoiding collisions. The second set, which carries out planning while being aware of the animation clips available, need to perform some pre-process to analyze the set of animation clips available to plan paths respecting constraints between the feet and the floor. In some cases, if the animation set is handmade, then the analysis is not necessary because the animations have already been built with specific parameters (such as speed, angle of movement and distance between feet) which are taken into consideration when planning.

We have already talked about the first group. The second group works directly with the set of available animations to construct motion graphs [Kovar et al.,

2008, Min and Chai, 2012, Ren et al., 2010, Zhao and Safonova, 2009], or pre-computed search trees [Lau and Kuffner, 2006]. Figure 3.9 shows one of these trees. These approaches try to reach the goal by connecting motions to each other [Witkin and Popovic, 1995], sometimes limiting the movements of the agents. Other methods try to use motion graphs in the first group combining it with path planners [van Basten et al., 2011]. Having a large animation database reduces the limitations in terms of freedom of movement, but also makes the planning more time consuming. The ideal solution would be one that could find a good trade-off between these two goals: freedom of movement and fast planning.

Our work [Beacco et al., 2013b], presented in 4.2 is an animation-dependent planner that uses the footstep paradigm. We plan local motion using the a database of motion clips consisting of individual footsteps.

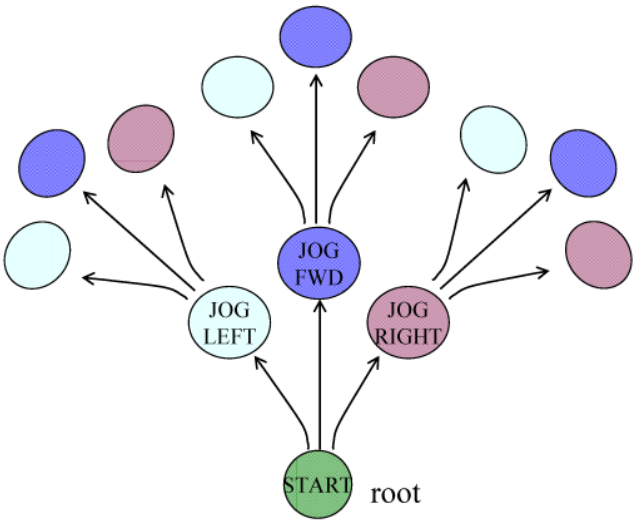


Figure 3.9: Precomputed search trees are used to plan interactive goal-driven animation, but limit the number of possible movements. [Lau and Kuffner, 2006]

3.1.3 . Data-Driven Techniques

Data-driven algorithms have been used in computer animation, for motion capture of single characters, facial motion capture, or even trees. In crowds the emphasis is on multi-character interactions, being able to capture subtle actions. We can divide data-driven techniques in two categories of approaches: the ones analyzing data in order to extract parameters for a model, such as social forces models or rule-based models; and the example-based techniques, which synthesize motion directly from teh input data.

3.1.3 . Parameter Extraction

A prediction based approach to crowd steering from motion capture data was proposed in [Paris et al., 2007] and [Pettré et al., 2009]. They had a controlled environment, where goals are known and the data acquisition is easier and more accurate. The extracted data is used to estimate the time of collision between entities.

Data from videos can also be used to modify the social forces model of Helbing [Helbing et al., 2000]: the new forces try to keep social groups together using the vision of the agent, attraction and repulsion forces [Moussaïd et al., 2009, 2010].

The information from videos, plus a stochastic crowd model can be used to predict future individual motion [Pellegrini et al., 2010]. The position and movements of players in a game can also be tracked with multiple cameras, in order to generate a motion field from all the players and predict their future points of interest [Kim et al., 2010].

User can also define the crowd motion. A sketch-based interface was proposed where the user draws example paths [Oshita and Ogiwara, 2009]. The simulation parameters are then extracted from it, such as guiding paths, speed, distances, crowd regularity, etc.

3.1.3 . Example-Based Simulation

If an agent in the simulation can find a person in a video facing a similar situation or state, we can copy part of its trajectory or actions to mimic its behavior [Lee et al., 2007, Lerner et al., 2007, Musse et al., 2007]. Behavior is influenced by different factors. Potentially influencing factors include personality, emotions, terrain, obstacles, surrounding people, etc. But not all factors have the same amount of influence. In a preprocess videos are tracked, examples are defined using the surroundings of each person, and a database is build. During the simulation of an agent, a query is defined and the database is explored to find a matching example. If so, the trajectory is copied and followed by the agent until a new one is needed (see Figure 3.10).

The crowds by example method has several issues: they are not as fast as other methods, they do not give a high-level control of the characters and have occasional errors. Some of these problems are addressed in [Charalambous and Chrysanthou, 2014].

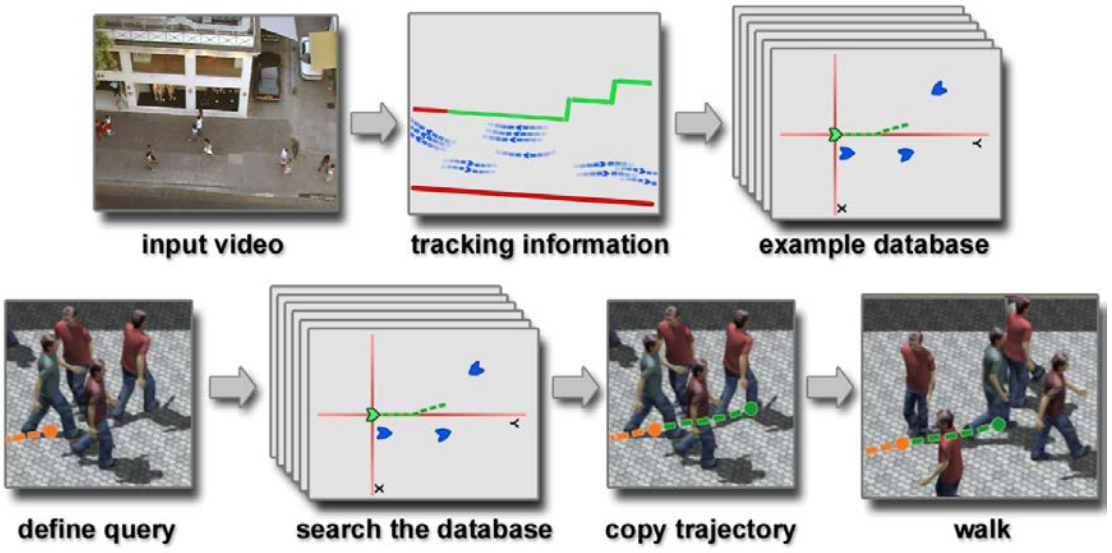


Figure 3.10: Crowds by example overview. The top row depicts the construction of a database, which takes place during preprocessing: the input video is manually tracked generating a set of trajectories. These are encoded as examples and stored in the database. At run-time, bottom row, the trajectories of the agents are synthesized individually by encoding their surroundings (forming a query) and searching the database for a similar example. The trajectory from the example is copied over to the simulated agent. [Lerner et al., 2007]

3.1.4 . Global Planning

By global planning we refer to all the pathfinding algorithms that can be applied to the navigation problem in crowd simulation. Navigation meshes can be automatically computed from the environment [Mononen, 2009, Oliva and Pelechano, 2013], and the resulting graph is used to perform the search from one start node to a goal node.

3.1.4 . A* and Weighted A*

One of the classic algorithms on navigation planning is the *A** algorithm [Hart et al., 1972], which derives from another classic algorithm by Dijkstra. Basically it tries to reduce a cost function **f = g + h**, where **g** is the cost to reach the current state and **h** is an heuristic determining the remain cost to get to the goal (for example the Euclidian distance, or the Manhattan distance). So A* gives always the optimal path if it exists.

A modified version of A*, but faster is the *Weighted A**, where the cost function is now **f = g + w*h**. Here **w** represents a weight by which we increase the importance of the heuristic term, so that we are able to reach the goal faster. Then Weighted A* is going to give us a sub-optimal path (rather than an optimal one), but with an important performance improvement.

3.1.4 . Incremental Planners

Incremental planners speed up searches bu reusing information from previous searches to speed up the current search. This might be important in unknown or dynamically changing domains, since problems can be now solved much faster than solving them repeatedly from scratch.

D* Lite [Koenig and Likhachev, 2002] is able to repair plans when changes in transition costs are detected, and guarantees optimality. To do so it maintains a least cost path from start to goal. The algorithm stores the cost **g(s)** from start to state **s**, and a one-step look-ahead **rhs(s)**. rhs(s) is 0 if **s** is the goal state; otherwise it is defined as the minimum sum of the cost from **s** to **s'**, from all successors states **s'** of **s**, plus the cost **g(s')**. D* Lite ensures that a least cost path

will have been found by the time the algorithm terminates. *Lifelong Planning A** [Koenig et al., 2004] can handle changes in transition costs and in addition or deletion of nodes (basically a changing environment) and reuses previous plan.

3.1.4 . Anytime Algorithms

An *anytime algorithm* is an algorithm bounded by time. Anytime planning is well suited for real-world planning problems where time for deliberation is limited. They find a feasible solution quickly and continually work on improving it until time runs out.

*ARA** [Likhachev et al., 2003] starts by finding a sub-optimal solution. This is fastly done using a Weighted A* search with an initial loose bound. Then it progressively relaxes the bound while it reuses previous efforts (reuses previously computed costs). This makes it significantly more efficient than other anytime methods, since it only expands a state at most once in a given search, and only inconsistent states from the previous search are considered for the next search. It also provides theoretical guarantees on bounds of sub-optimality, and it is able to find an optimal solution if time permits.

3.1.4 . Anytime Dynamic A*

The *Anytime Dynamic A** planner [Likhachev et al., 2005] combines the properties of incremental planners such as *D* Lite* [Koenig and Likhachev, 2002] and anytime algorithms such as *ARA** [Likhachev et al., 2003] to provide an algorithm which efficiently repairs its solutions to accommodate world changes and agent movement, while providing solution guarantees under strict time constraints. It performs repeated backward searches (from goal to start), reusing previous search efforts to iteratively produce solutions with improved bounds on optimality, like ARA*. This is done using an inflation factor ϵ which effectively weighs the contribution of the heuristic value in estimation of node costs, thus focusing the search towards the goal, expanding fewer nodes to produce ϵ sub-optimal solutions [Pearl, 1984].

3.1.5 . Multi-Domain and Hierarchical Planning

Planning based control of autonomous agents has demonstrated control of single agents with large action spaces [Choi et al., 2003, Fraichard, 1999, Shapiro et al., 2007]. In an effort to scale to a large number of agents, meet real-time constraints, and handle dynamic environments, a large variety of methods [Pettré et al., 2008] have been proposed. The complexity of the domain is made simpler [Lau and Kuffner, 2005] to reduce the branching factor of the search, or the horizon of the search is limited to a fixed depth [Choi et al., 2011, Singh et al., 2011]. Anytime planners [Likhachev et al., 2003, van den Berg et al., 2006] tradeoff optimality to satisfy strict time constraints, and have been successfully demonstrated for motion planning for a single character [Safonova and Hodgins, 2007]. Randomized planners [Hsu et al., 2002, Shapiro et al., 2007] expand nodes in the search graph using sampling methods, greatly reducing search efforts to make it a feasible solution in high-dimensional, continuous domains. The work in [Hoff et al., 2000] exploits the use of graphics hardware to enable interactive motion planning in dynamic environments.

Hierarchical planners [Botea et al., 2004, Bulitko et al., 2007, Holte et al., 1996] reduce the problem complexity by precomputing abstractions in the state space, which can be used to speed up plan efforts. Given a discrete environment representation, neighboring states are first clustered together to precompute abstractions for high-level graphs. Different algorithms are proposed [Kring et al., 2010] which plan paths hierarchically by planning at the top level first, then recursively planning more detailed paths in the lower levels, using different methods [Lacaze, 2002, Sturtevant and Geisberger, 2010] to communicate information across hierarchies. These include using the plans in high-level graphs to compute heuristics for accelerating searches in low-level graphs [Holte et al., 2005], using the waypoints as intermediate goals, or using the high-level path to define a tunnel [Gochev et al., 2011] to focus the search in the low-level graph. The work in [Arikan and Forsyth, 2002] demonstrates the use of randomized search in a hierarchy of motion graphs for interactive motion synthesis.

In 4.1 we present a work that builds on top of excellent recent contributions [Levine et al., 2011], showcasing the use of space-time planning for global navigation in dynamic environments, for a single agent. Levine *et al.* [Levine et al., 2011] uses parametrized locomotion controllers to efficiently reduce the

branching factor of the search and assumes that object motion have known trajectories, thus mitigating the need for replanning. Lopez *et al.* [Lopez et al., 2012] introduces a dynamic environment representation which is computed by deducing the evolution of the environment topology over time, thus enabling space-time collision avoidance with no prior knowledge of how the world changes. In contrast, we use multiple heterogeneous domains of control, and present a planning-based control scheme that reuses plan efforts across domains to demonstrate real-time, multi-character navigation, in constantly changing dynamic environments. Instead of automatically computing abstractions from a given representation, we develop a set of heterogeneous domains with different state and action representations that provide trade-offs in control fidelity and computational performance, and investigate different methods of communicating between domains to meet our application needs.

3.1.6 . Crowd Simulation Conclusions

One of the goals of interactive applications such as videogames is to have high fidelity navigation of interacting autonomous agents in non-deterministic, dynamic virtual worlds. The environment and agents are constantly affected by unpredictable forces (for example a human input or a player action), making it impossible to accurately extrapolate the future world state to make optimal decisions. These complex domains require robust navigation algorithms that can handle partial and imperfect knowledge, while still making decisions which satisfy space-time constraints.

Different situations require different granularity of control. An open environment with no agents and static obstacles requires only coarse-grained control while cluttered dynamic environments require fine-grained character control with careful planned decisions that have spatial and temporal precision. Some situations, like potential deadlocks at narrow doorways, may require explicit coordination between multiple agents.

The problem domain of interacting autonomous agents in dynamic environments is therefore extremely high-dimensional and continuous, with infinite ways to interact with objects and other agents. Having a rich action set, and a

system that makes intelligent action choices, facilitates robust, intelligent virtual characters, at the expense of interactivity and scalability. Greatly simplifying the problem domain yields interactive virtual worlds with hundreds and thousands of agents that exhibit simple behavior. The ultimate, far-reaching goal is still a considerable challenge: a real-time system for autonomous character control that can handle many characters, without compromising control fidelity.

Previous work simulates crowds by decoupling global navigation [Kallmann, 2010, Sung et al., 2005] and local collision avoidance [Pelechano et al., 2008], or demonstrates space-time planning for global navigation for a single character [Levine et al., 2011], while meeting real-time constraints. These approaches provide a tradeoff between number of agents, control fidelity, and environment complexity. But to the best of our knowledge, none of the proposed techniques efficiently accounts for the dynamic nature of the environment at all levels of the decision-making process.

3.2 . State of the Art on Crowd Animation

Although the literature on character animation is very large, this thesis is focused on real-time crowds. The animation problem we want to solve is to transform the movements of the agents within a simulation into the motion of a 3D animated character. Therefore this section presents a global overview of motion synthesis, and then the related work on how to synthesize the crowd motions, whether the input is a root trajectory or a footsteps trajectory.

3.2.1 . Motion Synthesis

There is a large work in the literature about synthesizing motion and particularly walking locomotion. We can classify the different approaches in three categories: *procedural techniques*, *physics-based techniques* and *example-based techniques*.

3.2.1 . Procedural Techniques

Procedural techniques are those creating motion from scratch, using empirical and biomechanical concepts. They offer a high level of control but they are usually not perceived as realistic. Boulic et. al. presented an model for human walking based on biomechanical models [Boulic et al., 1990]. Although they have a high control over the animation, the final result seems unnatural. Procedural techniques may only be useful for particular kinds of movement, like running [Bruderlin and Calvert, 1989].

3.2.1 . Physics-Based Techniques

Physics-based techniques use dynamics and physical properties to generate realistic animations, with realistic torques on joints [Faloutsos et al., 2001, Popović and Witkin, 1999]. The main issues are that you have less control over the animation, and that they are computationally expensive. This makes them unsuitable for our real-time purposes.

3.2.1 . Example-Based Techniques

Example-based techniques are those reusing existing motions to generate a clip of locomotion. The motions used are mostly motion capture clips, in order to make the results more natural. The clips can be concatenated to generate new sequences, or they can be blended or merged by parametrization to synthesize completely new clips.

Motion Concatenation

Clips of motion can be stitch together using some transition between them. The resulting *motion concatenation* can result in a very natural motion, as no part of it is invented or completely new. For example, motion patches is another technique used to synthesize interactive motions between different characters [Kim et al., 2012]. They tile, spatially and temporally, deformable motion patches. Each motion patch is a collection of motion fragments encapsulating interactions among characters (see Figure 3.11). But the resulting motions generate a seamless simulation of virtual characters interacting with each other in a non-trivial manner, which means they cannot correspond to any crowd simulation module.

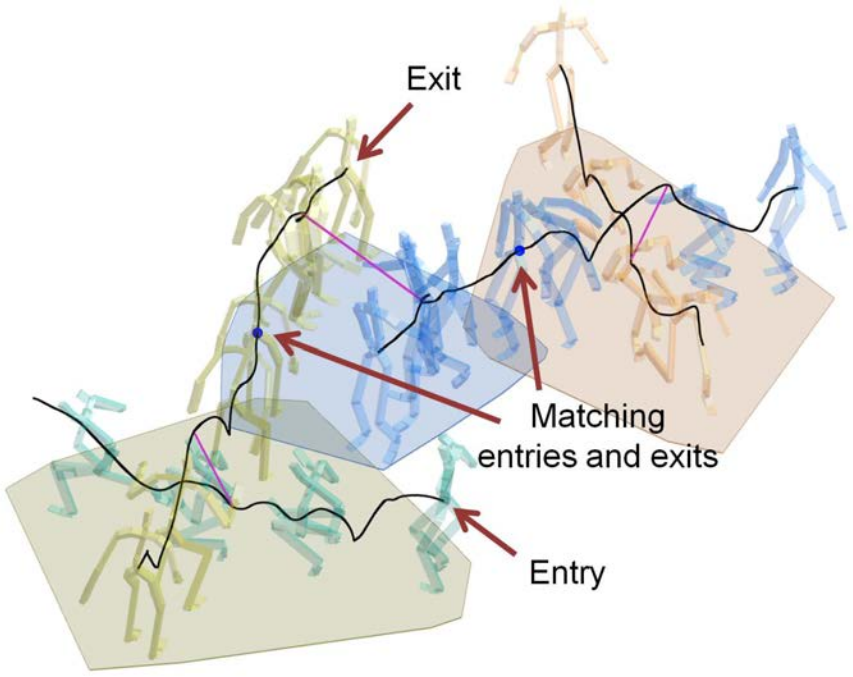


Figure 3.11: Stitching deformable patches as their matching entries and exits. Each patch is depicted as a convex polygon that encloses its motion paths projected on the ground. [Kim et al., 2012]

Motion Parametrization

By *motion parametrization* or *motion blending* we refer to those techniques that interpolate between different existing motions. They generate new motions corresponding to a specific parameter, such as an end effector position. In that spirit, Heck et. al. presented their parametric motion graphs [Heck and Gleicher, 2007], which offered a higher level of control than other paradigms (see Figure 3.12). A good survey on motion blending and interpolation techniques can be found in [Feng et al., 2012].

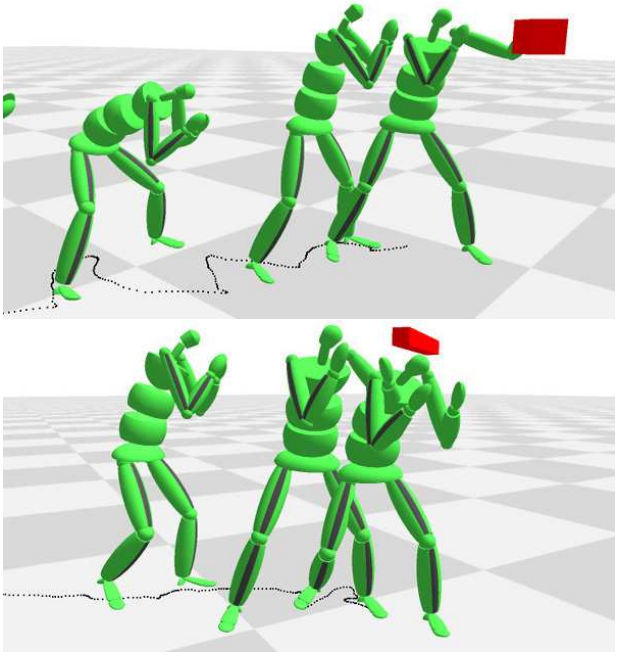


Figure 3.12: An interactively controllable boxing character that uses parametric motion graphs. The character is punching towards a user-requested target in the top image. In the bottom image, the character is ducking below a user specified height. [Heck and Gleicher, 2007]

Retrieving motions

All these approaches need to use existing motion clips. To obtain them we can either have an artist creating them manually, with software such as 3D Studio Max [Autodesk, 2014a] or Maya [Autodesk, 2014c], or we can use motion capture systems. The first can be more accessible and more flexible, but their creation is very time consuming and results are strongly dependent on the artists abilities and experience. On the contrary, motion capture clips are usually more realistic and natural than manually created ones.

FIGURE 3.13: Andy Serkis starring as Caesar in 20th Fox movie *"The Dawn of the Planet of the Apes"*, 2014. The performance of the actor is capture by complex motion capture systems used by *Weta Digital*, using multiple cameras and markers (left), and then transferred to its digital character (right).

FIGURE 3.14: *Microsoft Kinect 2.0* is a low cost depth camera. One of its applications can be to perform markless motion capture.

Motion capture systems can go from very expensive with high quality results, like those used for movies (see Figure 3.13) or high budget videogames, to more low cost solutions with not such good results, like those using the *Microsoft Kinect* camera (see Figure 3.14). These systems can also be divided into those needing markers to obtain positions of the actor, and into markless solutions working with computer vision techniques or depth cameras. It is also important to notice that motions directly obtained by a motion capture system tend to have a lot of noise that needs to be cleaned up [Ikemoto et al., 2006, Kovar et al., 2002b].

Fortunately, the web has some useful databases of motion capture clips which are available for the community. That is the case for example of the Carnegie Mellon University database [University, 2013], which offers thousands of motions properly classified by categories.

3.2.2 . Synthesizing Crowd Motions From Root Trajectories

Addressing the problem of integrating animated 3D figures with a crowd simulation, we can classify crowd simulation models into two approaches. The first encompasses those models that calculate the movement of agents in a virtual environment without taking into consideration the underlying animation. The second is defined by those that have a core set of animation clips that they play back, or blend between, to move from one point to another.

When the first group outputs root positions for the agents, it suffers from an artifact known as foot sliding (see Figure 3.15), since the position of the characters is updated without considering the foot position. Notice this is not exactly the same problem addressed by other works removing motion capture noise [Ikemoto et al., 2006]. If a user controls the character with a 3rd person controller, it is common to work on a root velocity basis, because the user wants to move the character around in an agile way. In such cases, like video-games, real-time response is critical and artifacts such as foot skating can be ignored. But if the character is not controlled by the user, as it happens in a crowd simulation system, then this becomes a major artifact since it is repeated over all the agents in the crowd, making it even more noticeable.

The second approach, reviewed in 3.1.2.7, puts effort into avoiding foot-sliding

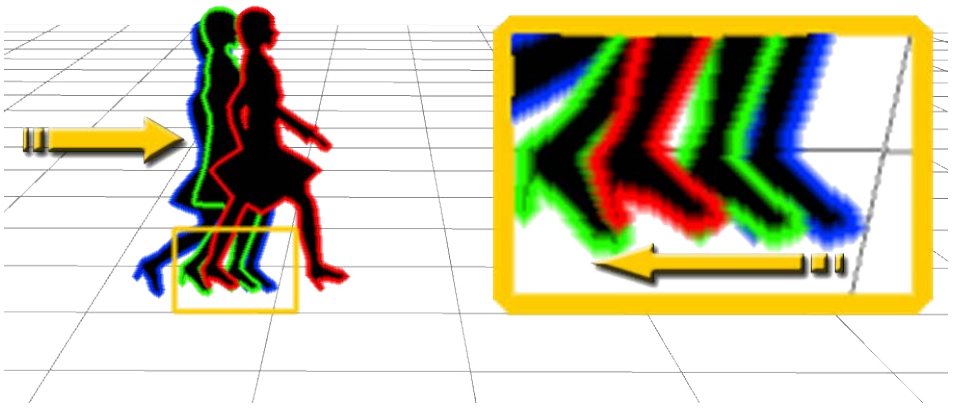


Figure 3.15: Foot Sliding. Here three consecutive frames (blue, green and red) of a walking animation show how the foot, instead of being planted, seems to slide backwards on the floor while the character is moving forward.

while limiting the number of possible movements for each agent. These are basically the example-based methods. Lau and Kuffner [Lau and Kuffner, 2006] introduced precomputed search trees for planning interactive goal-driven animation (see Figure 3.9). These motion concatenation models are often limited to a small graph of animations and play one animation clip after another, thus moving the agent according to the root movement in each animation clip. As such they do not usually perform well in interactive, dynamic environments, especially with dense crowds where collision response forces have an impact.

Recent trends in character animation include driving physical simulations by motion capture data or using machine learning to parametrize motion for simplified interactive control. Examples are inverse kinematics and motion blending based on gaussian process statistics or geostatistics of motion capture data [Grochow et al., 2004, Mukai and Kuriyama, 2005]. Some approaches create a Delaunay triangulation with the root linear and angular velocities of all the available clips, in order to parametrize and interpolate them [Pettré and Laumond, 2006, Pettré et al., 2003] (see Figure 3.16). Such techniques avoid foot sliding but are computationally more expensive.

Through interpolation and concatenation of motion clips, new natural looking animations can be created [Witkin and Popovic, 1995]. Kovar et. al. introduced motion graphs [Kovar et al., 2002a]. Zhao et. al. extended them to improve connectivity and smooth transitions [Zhao and Safanova, 2007] (see Figure 3.17). These techniques can avoid foot sliding by using the root velocity of the original motion data, but they require a large database of motion capture data to allow for interactive change of walking speed and orientation.

Menardais et al. were able to synchronize and adapt different clips without motion graphs [Ménardais et al., 2004]. Proportional derivative controllers and optimization techniques are used [Da Silva et al., 2008, Yin et al., 2007] to drive physically simulated characters. Goal-directed steps can perform a controlled navigation [Wu and Zordan, 2010]. While such techniques show very impressive results for a single character in real time, the computational costs mean they are not suitable for real time large crowd simulations. Kovar et. al. [Kovar et al., 2002b] presented an online algorithm to clean up foot sliding artifacts of motion capture data. But the technique is computationally not suitable for large real time crowds.

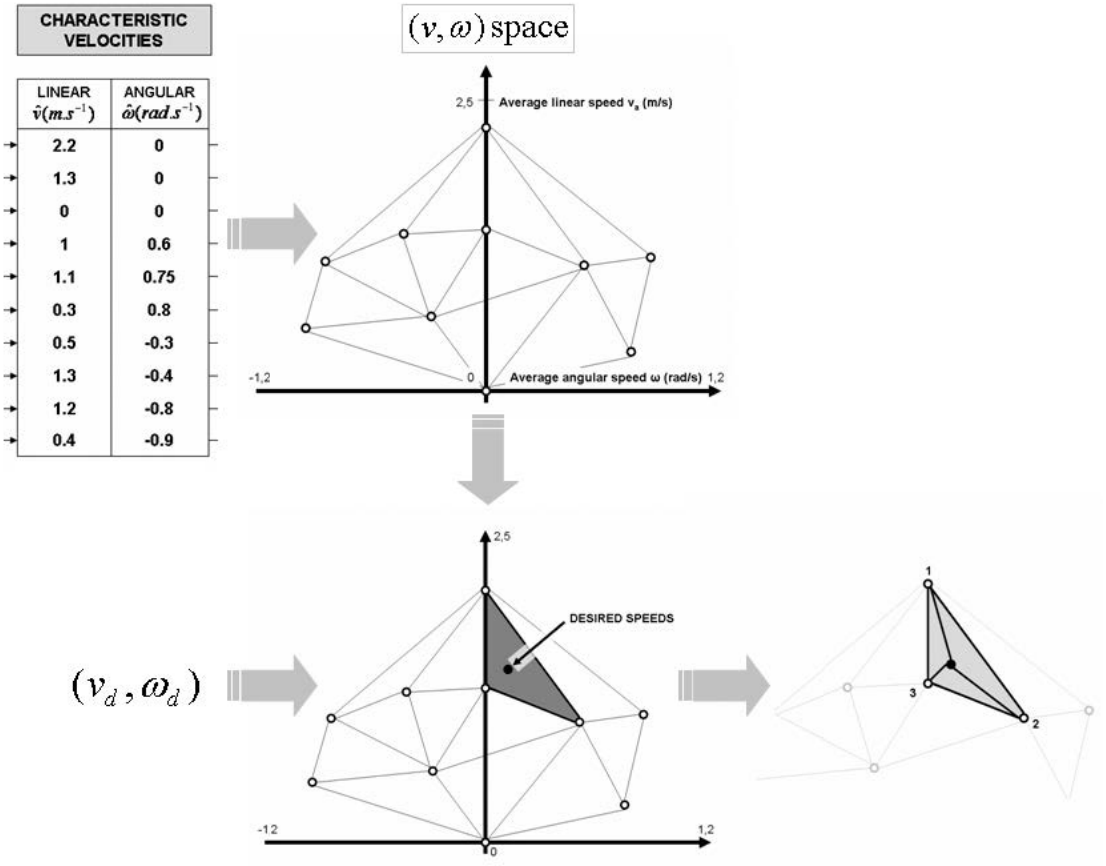


FIGURE 3.16: Motion capture selection and weighting process using a Delaunay triangulation from the linear and angular velocities. [Pettré and Laumond, 2006]

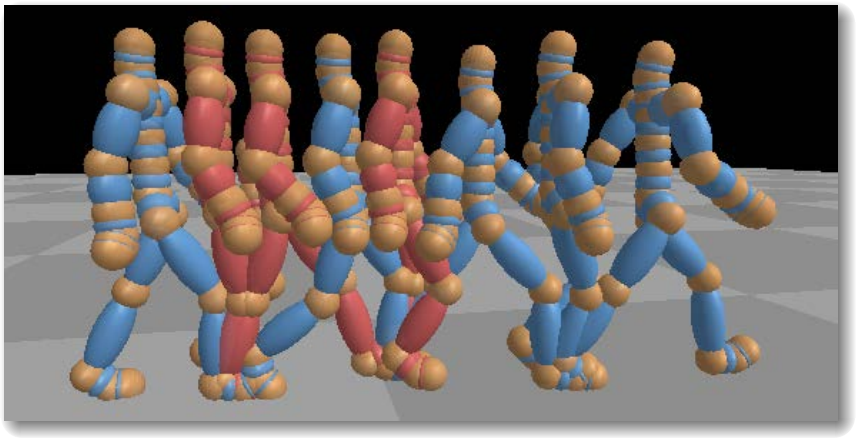


FIGURE 3.17: Using a motion graph with good connectivity smooth transitions between walks with different step length can be synthesized. Pose shown in blue belong to the original data-set, poses in red are interpolated poses introduced during the construction of the graph with good connectivity. [Zhao and Safanova, 2007]

FIGURE 3.18: A character spins while following a straight line. When fixed obstacles suddenly appear in the path, the character automatically switches to obstacle avoidance mode and navigates around the obstacles. [Treuille et al., 2007]

FIGURE 3.19: Comparison of synthesized motion of human character with semi-procedural adjustments (foreground) and without (background). [Johansen, 2009]

Treuille et. al. [Treuille et al., 2007] generated character animations with near-optimal controllers using low-dimensional basis representation (see Figure 3.18). This approach also uses graphs but, unlike previous models, blending can occur between any two clips of motion. They also avoid foot sliding by re-rooting the skeletons to the feet and specifying constraint frames, but their method requires hundreds of animation clips which is very time consuming to gather.

There are some *semi-procedural* animation systems, like the one by Johansen [Johansen, 2009], that work with a small set of animations and use inverse kinematics only over the legs to ensure ground contacts and adapt the feet to possible slopes of the terrain (see Figure 3.19).

Recently, Gu and Deng [Gu and Deng, 2010] increased the variety realism creating new stylized motions from a small motion capture data-set. Maim et. al. [Maim et al., 2009b] apply linear blending to animations selected based on the agent's velocity and morphology achieving nice animations for crowds but without eliminating foot sliding.

In 5.1 we present a new example based approach, the Animation Planning Mediator (APM). We select the parameters to feed a motion synthesizer while it feeds back to the crowd simulation module the required updates to guarantee consistency. It works even with a small amount of clips, and allows a large and continuous variety of movements. Our method can be used with any crowd simulation software, since it is the crowd simulation module which drives the movement of the virtual agents and our module limits its work to adjusting the root displacement and skeletal state.

3.2.3 . Footstep-Driven Animation Systems

As reviewed in 3.1.2.6, a novel steering method was proposed in [Singh et al., 2011], using footsteps to navigate characters in dynamic crowds. This generates a timed sequence of footsteps, but they have just been followed by animation techniques off-line, using software such as Autodesk 3D Studio Max [Autodesk, 2014a]. So, even if their simulation output is in real-time, it is not straight forward to integrate it with animations in real-time.

Footstep-driven animation systems [Girard and Maciejewski, 1985] produce unnatural results using procedural methods. Often, the global pelvis motion is determined first and then the leg motion is adapted using IK. The work in [Ko and Badler, 1996] performed a global optimization over an extracted center of mass trajectory to maximize the physical plausibility and perceived comfort of the motion, in order to satisfy the footprint constraints. Similarly, a procedural space-time based approach was presented in [van de Panne, 1997], where a physics-based optimizer determines the root trajectory, and then IK is used to determine the leg motion. This was extended for quadruped locomotion in [Torkos and van de Panne, 1998]. In [Chung and Hahn, 1999] a procedural hierarchical system generates locomotion over footprints laid over uneven terrain using biomechanical principles.

Some physics controllers are aware of footsteps and try to follow them [Coros et al., 2008, Wu and Zordan, 2010]. These methods can even automatically generate reactive steps in response to external forces (see Figure 3.20). Their main problem is still the performance, which is good for one or a small set of characters, but no for large groups nor crowds.

The work in [Chai and Hodgins, 2007] uses a statistical dynamic model learned from motion capture data in addition to user-defined space-time constraints (such as footsteps) to solve a trajectory optimization problem. The more constraints defined the better the motion results (see Figure 3.21), but this means that it is not enough to define foot constraints in order to obtain a realistic and natural motion. They also admit to have foot sliding when only root and one foot are used as constraints. In [Choi et al., 2003] random samples of footsteps make a roadmap going from one point to another, used to find a minimum-cost sequence of motions matching it which are then retargeted to the exact foot placements. But using roadmaps requires a preprocess, meaning it only works for fix environments.

Large semi-parametric motion graphs can be created by discretely blending all pairs of motions from a standard motion graph [Safonova and Hodgins, 2007]. These graphs can then be pruned and searched using a global algorithm. This offers a greater accuracy than standard motion graphs, because it allows the interpolation of two motion. With this approach motion can be generated over a set of constraints like footsteps, but it becomes computationally expensive (order of minutes).

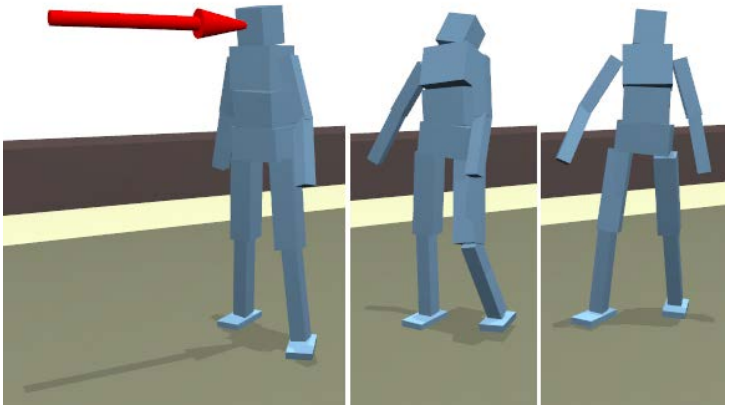


FIGURE 3.20: A reactive step generated automatically in response to a disturbance. [Wu and Zordan, 2010]

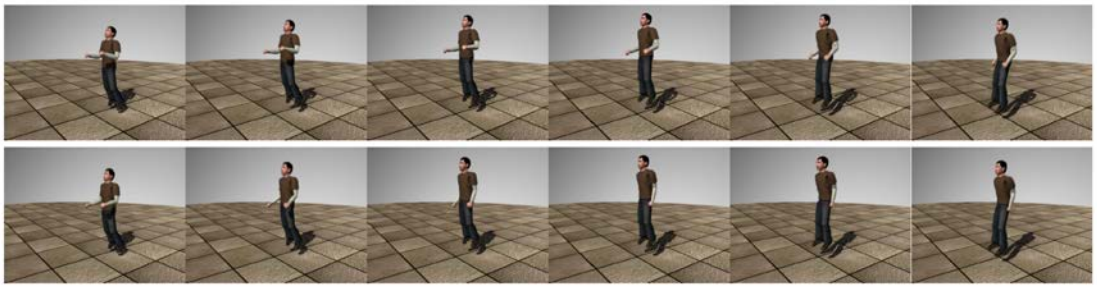


FIGURE 3.21: A user can fine tune an animation by incrementally adding constraints: (top) jumping generated by the user using five key trajectories (both hands, both feet, and root); (bottom) a slightly different jumping motion generated after adjusting the positions of the hands at the top of the jump. [Chai and Hodgins, 2007]

There are other interpolation schemes working with constraints: radial basis functions are used in [Rose et al., 2001], geostatical motion interpolation in [Mukai and Kuriyama, 2005], and a k-nearest neighbor interpolation scheme in [Kovar and Gleicher, 2004]. They all need to have motion clips semantically identical, and annotated with key events, like foot stance. The main problem of all the motion parametrized approaches is the non-linearity of the orientation domain, and the constraints defined by an articulated skeleton, which consists of joints and bones. Because of that, blending motions does not yield a linear parametrization of the space, and the resulting motion does not exactly correspond to the desired ending parameter. The error of these methods depends on the number of examples in the parameter space.

This problem can be solved by modifying the resulting motion. The trajectory of the root can be forced to follow the desired curvature, and retargeting applied to solve foot sliding [Park et al., 2002]. Or the motion can be transformed using IK [Grassia, 2000]. The desired position can also be iteratively adjusted in the direction of the error vector [Rose et al., 2001]. In any case, the exact parameterization cannot be guaranteed.

Other techniques perform resampling to reduce the error. Nearly convex weights can be randomly generated, and pseudo-examples created by blending nearby examples with these weights [Kovar and Gleicher, 2004] (see Figure 3.22). This is also used in parametric motion graphs [Heck and Gleicher, 2007].

A parametric space called the *step-space* is introduced and used in [Egges and van Basten, 2010, van Basten et al., 2010, 2011]. In a pre-process individual

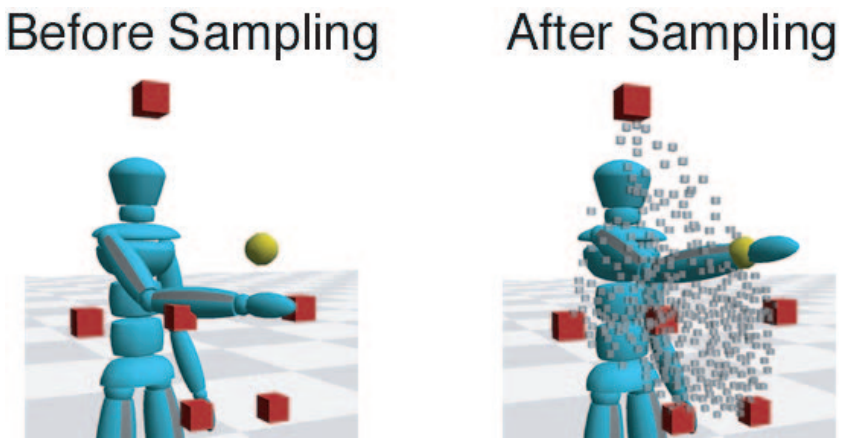


FIGURE 3.22: **Left:** Six example reaching motions create a sparse sampling of parameter space that leads to an innacurate parameterization. (The dots indicate parameter samples and the yellow sphere shows the desired location of the wrist. **Rigth:** A denser sampling can be automatically generated, providing greater accuracy. [Kovar and Gleicher, 2004]

steps are extracted from recorded motions. They consider a step the displacement of one foot, and they define the step-space with 10 parameters (see Figure 3.23). These include the swing foot start and end position and orientation, the supporting foot position and orientation, and 4 temporal parameters corresponding to different phases of the step. To synthesize motion they adopt a greedy nearest-neighbor approach over larger motion databases. To ensure spatial constraints, the character is properly aligned with the footsteps and reinforced with inverse kinematics, while temporal constraints are satisfied using time warping (see Figure 3.24). These techniques guarantee exact foot positioning, and no end-effector trajectory is modified. But their computational cost makes them unsuitable for real-time animation of large groups of agents.

In chapter 5, section 5.2 we present a method that produces visually appealing results with foot placement constraints, using only a handful of motion clips and can seamlessly follow footstep-based control trajectories, while preserving the global appearance of the motion. Compared to [Johansen, 2009], we exploit the combination of multiple parameter spaces for footstep-precision control. This reduces the dimensionality of the problem, compared to [van Basten et al., 2011]. Unlike previous work in the literature, our method can synthesize animations for a large number of characters in real-time, following footstep trajectories for different walking styles and even running motions with a small flying phase.

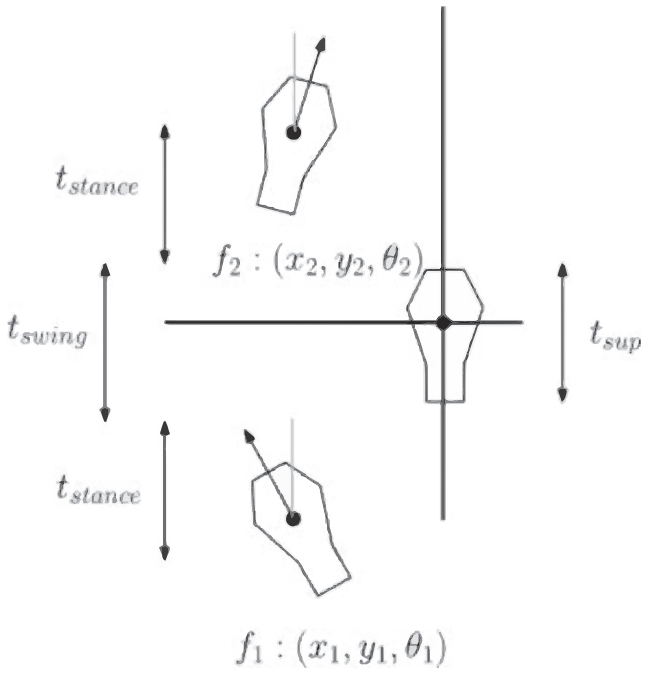


Figure 3.23: The step space: a step is represented by 10 parameters: the swing foot f1 moving to f2, the supporting foot and 4 temporal parameters.[Egges and van Basten, 2010, van Basten et al., 2010, 2011]

Figure 3.24: A character walks over a planned footsteps trajectory avoiding the red obstacles. [van Basten et al., 2011]

Footstep Extraction

When using example based methods for footstep driven motion, an important problem to address is how to analyze the available motion clips, and moreover how to detect and extract individual footsteps. Determining foot stances from motion capture data is not always trivial, due to noise and possible retargeting errors. But it is possible to determine foot downs in locomotion. In [van Basten and Egges, 2009] they use a height and velocity based footstep detector. By detecting significant changes in height matching to significant changes in velocities they identify foot downs.

Assumptions can also be done, such as that when we walk there is always one foot on the floor, or that both feet are in contact with the floor during a brief phase between swings. In [Johansen, 2009] in place animations are used, so foot downs are detected when feet velocities invert their sign. An advantage of this is that jumps or walks with a flying phase are supported.

3.2.4 . Crowd Animation Conclusions

Motion synthesis methods are divided in procedural techniques, physic based techniques and example-based techniques. Procedural techniques have a high level of control but have unnatural results. Physically based techniques are either computationally expensive or do not offer enough control over the animation. The more suitable techniques for our purposes are example-based techniques, which can offer natural motions with high control with acceptable computational costs. This applies whether we try to follow root or footstep trajectories.

To achieve natural looking crowds, most of the current work in crowd simulation uses avatars with a limited number of animation clips. In most cases, problems arise when the crowd simulator module updates the position of the root for each agent without taking into account movement of the feet. This yields unnatural simulations with foot sliding artifacts. Other problems can emerge from the lack of coherence between the orientation of the geometric representation of the agents and direction of movement. As we increase model sophistication through enhanced path selection, avoidance, rendering, and inclusion of more behaviors, these artifacts become more noticeable. In order

to avoid these problems, most approaches either perform inverse kinematics, which might imply a high computational cost that does not allow large crowd simulations in real time, or adopt approaches where the animation clip being used drives the root position which limits movements and speeds.

To accurately follow a footsteps trajectory, footstep driven animation systems are mostly locomotion systems with space-time constraints. Moreover, the output trajectory can be modified by external perturbations such as uneven terrain. Motion parametrization techniques offer natural and accurate results. The problem then is to retrieve large enough databases of motion capture clips. But a threshold also appears between the accuracy of these approaches and their computational cost, which is critical in the case of multiple agents or crowds.

Ideally, in order to animate crowds of hundreds of agents in real-time, we would want a method using an animation database with as less clips as possible. This is because retrieving natural motion capture clips is expensive and difficult, specially if we want variety of characters, with properly rigged skeletons and retargeted motions. Having more clips also implies to use more memory. It is also desirable to minimize the number of animations to blend between, since the cost of blending increases with the number of animations blended. Therefore it is necessary to find a good small set of animations to achieve accurate results. Even though accuracy is desirable, in many cases it is not as important as keeping real-time computation or a natural motion with no distracting artifacts. We therefore have a tradeoff between the number of animations in our library and the computational cost of the technique.

3.3 . State of the Art on Crowd Rendering

This section aims to study, classify and compare existing approaches for real-time crowd rendering. We first overview character animation techniques, as they are highly tied to crowd rendering performance. Then we analyze the state of the art in crowd rendering including point-based rendering, image-based rendering, culling techniques, level-of-detail, and hardware-related techniques. We also discuss other factors affecting performance and realism of crowds such as lighting, shadowing, clothing and variability. Finally we make an exhaustive comparison of the most relevant approaches in the field.

3.3.1 . Character Animation and Skinning

In this subsection we give an overview of general aspects of character animation that influence rendering performance. These aspects range from how the character is represented to how this representation is modified to handle animations.

Some character animation methods focus on achieving highly-realistic, physically-accurate mesh deformations for applications without real time requirements. Physically-based methods simulate the internal structures of the body (bones, tendons, muscles and fat tissues [Aubel and Thalmann, 2000, Jimenez et al., 2011, McLaughlin et al., 2011, Sueda et al., 2008]), achieve a high level of realism, and might even support dynamic effects such as muscle bulges, but at a high computational cost. Despite the high visual quality achieved by these methods, they are too expensive for rendering a large number of characters under real-time constraints. In these cases it is necessary to have more efficient methods capable of animating multiple models interactively. This survey focuses on crowd rendering, and thus we will start by reviewing character animation methods that can be handled in real time for large groups of characters. For more details on the skinning subject we refer the reader to a recent complete course [Jacobson et al., 2014].

3.3.1 . Skeletal Animation

The most extended approach for animating 3D characters is *skeletal animation* [Magnenat-Thalmann et al., 1988]. In this technique, the character is represented by a mesh or 'skin', and an underlying skeleton. The skin is an arbitrary polygonal mesh and the skeleton is a hierarchy of bones carefully placed so that they fit inside the skin. Initially both of them are designed in a reference pose, and during run time the mesh representing the skin will be deformed following the bones' movement (see Figure 2.14).

To determine how the vertices of the mesh will be deformed, each bone is associated with a portion of the mesh. For example, all the vertices forming the left hand will be linked to the left hand bone. In some places, a portion of the mesh is associated to more than one bone, by defining a weight (influence) associated to each one. These vertices will then be deformed based on the weights of all its linked bones that are being moved. Therefore, an animation can be defined by the movement of the bones, and the associated vertices will move along with the skeleton. The process of defining these weights, as well as the skeleton fitting, is called *rigging* and it is typically done manually. However, there are some automatic procedures [Baran and Popović, 2007, Pantuwong and Sugimoto, 2012, Ramirez et al., 2008], that can simplify the rigging process considerably. Recently developed online tools allow users to rig any biped character within minutes [Mixamo, 2014]. There is also recent work to transfer the rigging between different characters that share the same skeleton hierarchy[Bharaj et al., 2012], a process known as retargeting.

3.3.1 . Animation Blending

An animation is defined by a series of *keyframes*, each one defining a different *pose* for an instant of time t. Poses consist of a geometric transformation for each *bone* of the *skeleton*. These geometric transformations are usually rotations encoded as matrices, resulting in one matrix per bone and per keyframe. During the animation, new poses can be computed at arbitrary times by spherically interpolating the rotations of the bones between the two closest keyframes.

Linear blend skinning is the standard algorithm for low-cost skinning. Transformations are represented by the skeleton matrices, which are blended linearly

FIGURE 3.25: Linear blend skinning produces several artifacts such as the "candy-wrapper" effect, because the resulting matrix no longer represents a rigid transformation. [Kavan et al., 2007]

according to the applied rigging. Besides skin deformations, linear blend skinning can be used to animate other deformable elements such as cloth, since it is considerably faster than physically based cloth simulation [Cordier and Magnenat-Thalmann, 2005].

The direct linear combination of matrices is known to suffer from blending artifacts in the deformed skin. A typical artifact is the "candy wrapper" effect, where the skin collapses into itself (see Figure 3.25). This occurs because the weighted sum of matrices representing rigid transformations (with neither scale nor shear) is not necessarily another rigid transformation, but a general affine transformation. Rigid transformation matrices can be decomposed into quaternion plus translation pairs, which can be independently blended linearly to always get rigid transformations [Hejl, 2004, Kavan and Žára, 2005]. The problem of doing it separately is that transformations become dependent with the body-space coordinate system, meaning that vertices are going to rotate around the origin of the body-space instead of the actual pivot point of the closer joint. *Dual quaternions* approaches [Kavan et al., 2007, 2008a, McCarthy, 1990] reduce these artifacts in an elegant way, by blending quaternions whose elements are dual numbers.

In recent years cage-based deformations have been applied to character animation. The main advantages of cage-based deformation techniques are their simplicity, relative flexibility and speed. The idea is to use one or more cages

enclosing the model to facilitate the animation while preserving the smoothness of the deformed meshes. Chen et al. [Chen et al., 2011] presented an efficient approach that can generate both low and high-frequency surface motions such as muscle deformation and vibrations with little user intervention. Given a surface mesh, they construct a lattice of cubic cells embracing the mesh and apply lattice-based smooth skinning to drive the surface primary deformation with volume preservation. Secondary deformations are handled through lattice shape matching with dynamic particles. Gonzalez et al. [González García et al., 2013] recently proposed a versatile deformation scheme, allowing the usage of heterogeneous sets of coordinates and different levels of deformation, ranging from a whole-model deformation to a very localized one. This locality allows for faster evaluation and a reduced memory footprint, and thus outperforms single-cage approaches in flexibility, speed, and memory requirements for complex editing operations.

Implicit skinning [Vaillant et al., 2013] is the first purely geometric method handling skin contact effects and muscular bulges in real-time. The typical artifacts of geometric skinning techniques are avoided through an implicit surface representation. Every frame they approximate the mesh by a set of implicit surfaces, which are combined in real-time and used to adjust the position of mesh vertices starting from their smooth skinning position. This process is performed without any loss of detail and can seamlessly handle contacts between skin parts. Since it is a post-process, the method can be added to any standard animation pipeline. The method requires no intensive computation step such as collision detection and can achieve real-time performance for simple animations, although it is not fast enough for crowd rendering.

3.3.1 . Non-Skeletal Animation

An alternative approach to skeletal animation is *morph target animation*, also known as *per-vertex animation*, *shape interpolation* or *blend shapes*, where vertex positions are stored not only for the reference pose, but also for each keyframe [Lorach, 2007, Winkler et al., 2010]. Vertex positions are then interpolated within keyframes to obtain new frame deformations. An advantage of morph target animation over skeletal animation is that it provides artists with more control over the movement because they can define arbitrarily the individual positions

of the vertices within a keyframe, without being constrained by skeletons. This can be useful for animating cloth, skin, and facial expressions because it can be difficult to conform those things to the bones that are required for skeletal animation. In the field of crowd rendering, Ulicny et al. [Ulicny et al., 2004] avoid computing the deformation of a character mesh by storing pre-computed deformed meshes for each keyframe of the animation, and then carefully sorting these *static meshes* to take cache coherency into account. Switching consecutive meshes creates the illusion of an animation. Unfortunately, these techniques require a large amount of memory to store the animations and they are seldom used for crowd rendering.

3.3.1 . Individuality

Individuality in crowd animation refers to the possibility of having as many different animations as possible so that individuals within the crowd can be animated with multiple speeds, styles and gaits. In many situations crowds are just animated with a handful of animations that are run with a certain time offset to avoid synchronized animations. Although animations have a small memory footprint, the computation of all the blended poses can become a major performance bottleneck. Moreover, the final pose is represented by a set of matrices that must be used to transform all the avatar vertices. This transformation is usually performed in the vertex shader. Matrices can be computed in the CPU and then sent to the GPU, but this approach consumes a significant amount of CPU-GPU bandwidth. Alternatively, keyframe matrices can be preloaded onto the GPU, at the expense of GPU memory space. Both approaches thus benefit from matrix compression techniques.

3.3.2 . Point-Based Techniques

Levoy and Witted's report [Levoy and Whitted, 1985] early suggested the use of points as a new primitive to render geometry. The idea is to render a surface using a vast amount of points. A Gaussian filter or surface splatting [Zwicker et al., 2001] can be performed to fill in the possible gaps. Kobelt and Botsch presented a survey on point-based techniques [Kobbelt and Botsch, 2004]. But it was Bærentzen [Bærentzen, 2005] who proposed to use point-based models

to replace objects that are far away from the camera. Point-based rendering is more useful and faster when the triangles of a model cover a pixel or less (as there is neither triangle setup nor interpolation).

Point-sampled objects do not need to store and maintain globally-consistent topological information. Therefore they are more flexible when compared to triangle meshes. Nevertheless this technique has some limitations. For instance, if the point samples are the result of decimating the mesh for level of detail, they become independent from the original mesh and loading animations becomes difficult.

An alternative multi-resolution representation for animated geometry is proposed by Wand and Strasser [Wand and Straßer, 2002] who combine pre-filtered point samples and triangles arranged into an octree. Their randomized sampling scheme guarantees that sample points are distributed sufficiently uniformly on the animated geometry at any time during the animation (see Figure 3.26), at the expense of requiring a separate multi-resolution hierarchy for each pair of consecutive keyframes. Larkin et al. designed a time-critical system where point samples are distributed for every agent depending on its selected level of detail [Larkin and O'Sullivan, 2011].

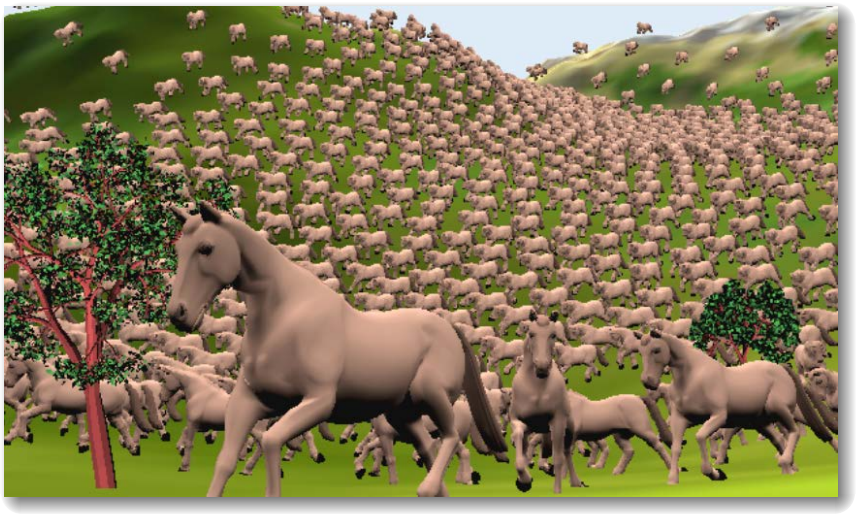


Figure 3.26: While close-up characters use triangle meshes, background characters can be rendered with point splats [Wand and Straßer, 2002].

Toledo et al. presented a system where an additional skeleton contains an octree per limb [Toledo et al., 2014]. Each level of the octree represents a different level-of-detail, and animations can be automatically transferred to each node, thus reducing the memory consumption (see Figure 3.27).

FIGURE 3.27: Some approaches allow animations to be applied to both point-based and polygon-based representations [Toledo et al., 2014].

3.3.3 . IMAGE-BASED TECHNIQUES

An impostor is in essence a simple primitive that has the capacity to fool the viewer. As opposed to LODs, impostors are not just a simplified version of the original geometry, but a different primitive conceived to replace it under appropriate viewing conditions. Impostor representations range from simple billboards (3D sprites) textured with an image of the rendered object, to a small set of textured polygons allowing the recovery of surface details and parallax effects. Although early impostors were designed for static objects, they can be also used to render animated objects and crowds of agents.

Millan and Rudomin performed an strict comparison between point-based techniques and image-based techniques [Millan and Rudomin, 2006a]. Their point-based characters required a variable number of points between 3 and 280, thus resulting in a more inefficient render than an impostor image-based approach using only a quad or two triangles per agent.

Since impostors are essentially images, there are two main approaches: to generate dynamically these images at runtime, or to pre-compute and store them into a texture atlas and access them when necessary.

3.3.3 . Dynamic Impostors

The virtual human impostor used by Aubel et al [Aubel et al., 1998] is a simple textured quad which rotates to continuously face the viewer. A snapshot of the virtual human is mapped onto it and re-used over several frames (see figure 3.28).

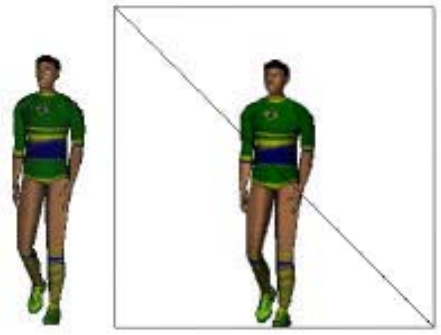


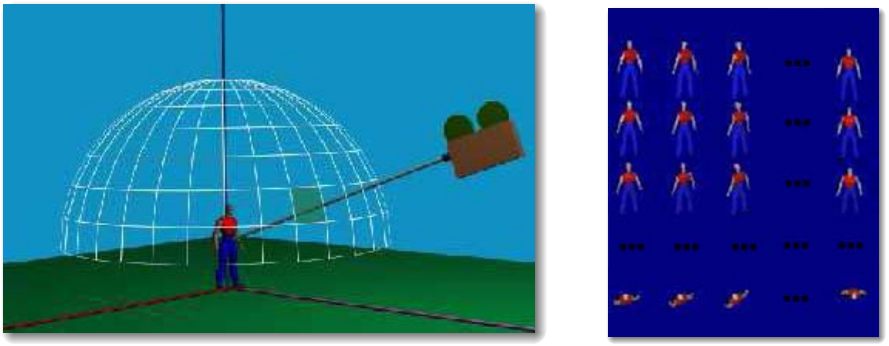
Figure 3.28: A football player and its (somewhat oversized) impostor. [Aubel et al., 1998]

As the humanoid moves or the camera moves, the mapped texture might need to be refreshed. To take updated snapshots an off-screen buffer is set up and a multiresolution virtual human is placed in front of the camera in the right posture. The virtual human is then rendered and copied into texture memory and so ready to be mapped onto the billboard.

To decide whether or not to refresh the texture they proposed two fast algorithms. The first one tests distance variations between some pre-selected points in the skeleton, so they can decide if the pose has changed significantly. This obviously sub-samples the animation.

The second algorithm does not test independently the camera motion and the character's orientation because it is not important to know what factor caused the visual variation. Instead, they test the variations of the modelview matrix corresponding to the transformation under which the viewer sees the virtual human. These impostors are dynamic in the sense that they are not precomputed, but they change dynamically depending on the results of the two algorithms above at every frame.

The off-screen buffer can be set up in a pre-process, adjusting the frustum to the character. The stored impostor can also be re-used for other human meshes, and since posing up the character would have been done with the whole geometry,



(A) (B)

FIGURE 3.29: Discretising the view direction between the object and the viewpoint (a) allows to generate a texture with all the captured directions for one frame of one animation (b). The process must be repeated for every animation frame. [Tecchia and Chrysanthou, 2000]

the approach is not slower than rendering the 3D geometry. But even if the impostor is re-used, after a few frames it will finally be discarded.

The main limitation of these kind of approaches is that replacing the whole geometry by a textured plane might introduce occlusion problems. For example, imagine a character sitting on a chair as two independent meshes. If one, or both are replaced by a textured quad, it is not clear how they can be arranged spatially to avoid visibility problems. This is due to the depth values of the impostor fragments which are unlikely to be the same as those of the actual geometry.

3.3.3 . PRE-GENERATED IMPOSTORS

Pre-generated impostors were first used by Tecchia et al. [Tecchia and Chrysanthou, 2000] by rendering each character from several viewpoints and for every animation frame of a simple animation cycle (see Figure 3.29). The images were stored in a single texture atlas, and each crowd agent was rendered as a single polygon with suitable texture coordinates according to the view angle and frame.

Pre-generated impostors with improved shading have also been used by Tecchia et al. in [Tecchia et al., 2002] by adding shadows to each agent. Since the shadow is just the projection onto the ground of the character's silhouette, they can project the polygon of the impostor onto the ground, using the shadow

coverage to darken the ground (see Figure 3.30). This fake shadow is valid only on planar geometry with a parallel light source, but gives plausible results with small overhead.

Figure 3.30: A scene with shadowing pre-generated impostors.[Tecchia et al., 2002]

Pre-generated impostors can achieve rendering of crowds consisting of tens of thousands of agents. Unfortunately, although image and texture compression techniques can be applied to the resulting texture atlases, they still require large amounts of memory due to the per-view, per-frame replication. Some memory savings can be achieved by removing intermediate frame textures, and generating them online using morphing techniques [Yuksel et al., 2013]. An additional limitation of pre-generated impostors is that, depending on the texture resolution, close-up characters appear clearly pixelated. These impostors do not allow interpolation or blending between two or more different animations.

3.3.4 . Geopostors

Dobbyn et al. [Dobbyn et al., 2005] introduced the Geopostors, a hybrid system combining pre-generated impostors with a polygon-based representation. Figure 3.31 shows how impostors are used for far agents while the ones close to the camera are rendered with full geometry.

FIGURE 3.31: Geopostors. Far agents are rendered with impostors while closer ones are rendered with geometry. [Dobbyn et al., 2005]

The switching between the mesh and the impostor is based on the impostor image pixel size to impostor texel size ratio. Ideally this ratio should be 1:1, because aliasing starts when a texel is bigger than a pixel.

An extension of this approach was made by Pettre et al. [Pettré et al., 2006], combining the animation quality of dynamic meshes with impostors and adding a third LOD using the high performance offered by static meshes, i.e. meshes where animated poses were already computed.

3.3.5 . LAYERED IMPOSTORS

In geopostors, the visual gap between flat impostor and geometry might, for some view directions, be too large to completely avoid popping artifacts. Coic et al., [Coic et al., 2007] described a similar hybrid system but with three LODs, adding an intermediate layered impostors LOD between flat impostor and geometry to help achieving continuity during transitions. Instead of a single textured polygon, an adaptive number of layers of the color texture are drawn, depending on the texel's depth (see Figure 3.32). These layers fill a volume in the 3D scene and can be shaded dynamically using color textures enriched with depth and normal channels.

Layered impostors are rendered in layers parallel to the image plane. For each of these layers, the pixels that correspond to a certain depth are selected. After selecting the required number of layers, they divide the volume captured during preprocessing into as many intervals as the number of layers, defining the intervals of depth for selecting pixels in the color texture. The selection of the right pixels for each depth interval is done in a fragment shader, where lighting is also computed.

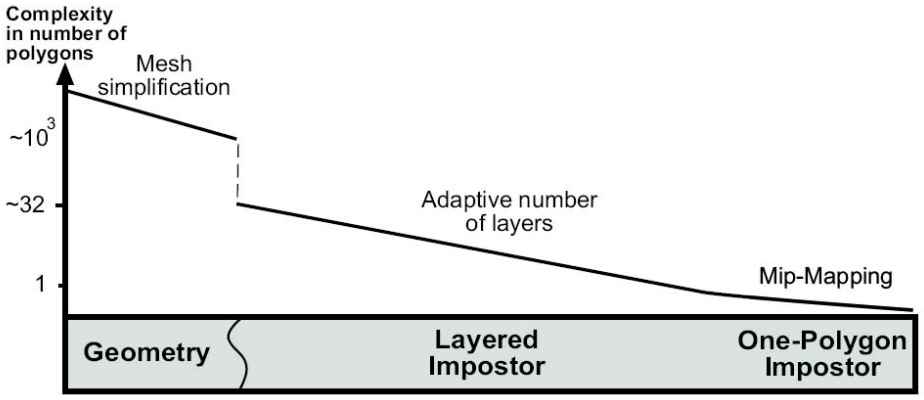


FIGURE 3.32: Volumetric layered based impostors rendering scheme: between geometry and the one-polygon impostor, an adaptive number of layers is used for a layered impostor. [Coic et al., 2007]

To extend the validity of the layered impostors, overlapping depth intervals can be used. Without overlapping, cracks appear on the layered impostor as soon as the viewpoint slightly differs from the pre-computed one. By drawing a small part of the previous and next layers, these gaps are avoided, extending the lifetime of the layered impostor and decreasing the density of precomputed views (see Figure 3.33).

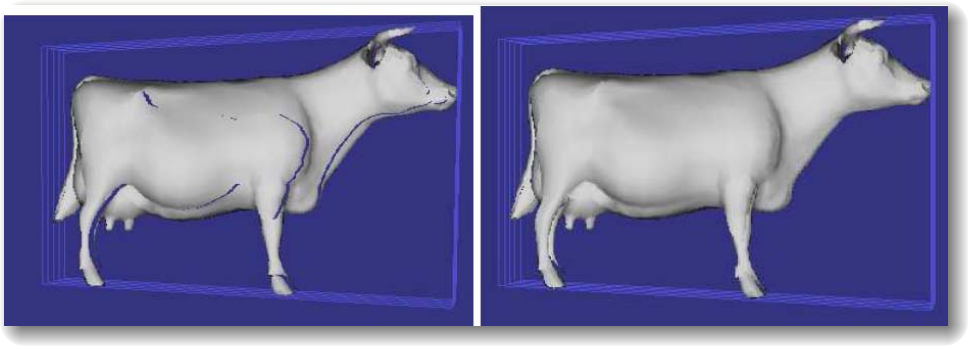


FIGURE 3.33: A cow rendered with 5 layers and dynamic lighting, without (left) and with overlapping (right). [Coic et al., 2007]

Although this approach improves visual quality and fills the gap between the polygonal representation and the flat impostors, it adds additional channels (normal, depth and several layers), which worsens the memory problem. Layered impostors are slower to render than one-polygon impostors.

3.3.6 . Polypostors

Polypostors were introduced by Kavan et al. [Kavan et al., 2008b] to reduce the memory requirements of pre-generated impostors whilst mantaining rendering performance. Each polypostor consists of a collection of 2D (planar) primitives, each one representing an individual body part for a given view direction (thus avoiding a per-frame memory consumption).

The original 3D character is cut into several body parts in order to minimize occlusion issues. The original skeletal animation is applied to the body parts. The composition of these body parts gives the same animation as the one provided originally. For the first frame of the animation, each body part is rendered and enclosed within textured 2D polygons, using a standard contour tracing algorithm. For all subsequent frames, an algorithm based on dynamic programming shifts the vertices of the 2D polygons so that they approximate the actual rendered image as closely as possible (see top of Figure 3.34). This algorithm matches two textured polygons in an optimal way with respect to a chosen error metric.

At run-time, the deformed polygons are composited in depth order, creating the illusion of an animated 3D character. Since polypostors approximate the animation by deforming their texture, they are not as accurate as other impostors. They can be applied only to animations that can be described as deformations of the initial key-frame. They can produce artifacts with views where there is a lack of texture information in the first key-frame, due e.g. to disoccusion effects (see bottom of Figure 3.34).

3.3.7 . Per-Joint Impostors

We present in chapter 6 two approaches adopting the polypostor idea of having impostors per body parts instead of per-character. We maximize performance by using a collection of pre-computed impostors sampled from a discrete set of view directions. The first method is based on relief impostors [Beacco et al., 2011] and the second one on flat impostors [Beacco et al., 2012]. Characters are animated by applying the joint rotations directly to the impostors, instead of choosing a single impostor for the whole character from a set of predefined

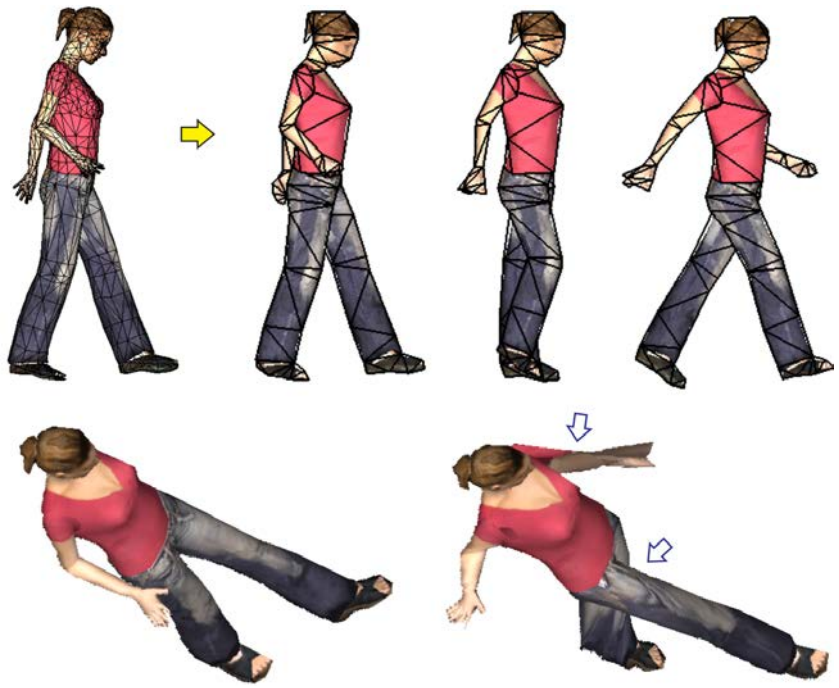


FIGURE 3.34: An example of a polypostor animation, overlaid with wireframe (top). Note that the character animation is created simply by displacing polygon vertices (stretching the texture accordingly). The Polypostor texture is generated from the first key-frame (bottom left) and deformed for subsequents key-frames, producing artifacts due to lack of data in areas that have become visible (bottom right). [Kavan et al., 2008b]

poses. This representation supports any arbitrary pose and thus the agent behavior is not constrained to a small collection of predefined clips.

Previously, in the same spirit, Aubel et al. [Aubel et al., 2000] divided each character into coherent parts by using the natural segmentation of joints. However, their subdivision was used exclusively for handling visibility issues rather than for animating each part separately as in [Beacco et al., 2012]. Maim et al. [Maim et al., 2009a] sampled individual parts from multiple view directions too, but their animation-independent impostors are limited to rigid accessories such as hats, wigs or backpacks.

3.3.8 . Culling Techniques

Culling techniques aim at discarding objects or parts of objects which are not visible [Cohen-Or et al., 2003]. In this section we show how these algorithms can be applied to crowd rendering and the problems that can be encountered.

Frustum culling is a very common technique for discarding all of the objects that are not inside the camera frustum, thus avoiding sending them to the graphics card and speeding up the rendering [Assarsson and Möller, 2000, Bishop et al., 1998, Clark, 1976]. The test to determine if a point or a bounding volume (box, spheres and cylinders are typical shapes) is partially or totally inside the frustum can be performed before the render instruction. In the case of crowd rendering we can simply use the bounding sphere of the agents to approximate their shape. Although this test is very fast, it has to be performed for every agent in the crowd, which can be very large (tens or hundreds of thousands). In the case of having most of the agents within the frustum, then this test will consume a large amount of processing time that could have been used to render more agents.

With the goal of applying *frustum culling* to crowd rendering there are some strategies that can be applied. If the simulation and the rendering run in parallel, the frustum test can be performed in the simulation thread which sets a visibility flag. Another possibility consists of keeping some data structure that can speed up the process of determining which agents need to be rendered, such as spatial hashing[Reynolds, 2006] or hierarchical representations[Toledo et al., 2014].

Visibility or *occlusion culling* algorithms determine if an object is occluded by some other geometry (mainly static geometry) before rendering it [Klosowski and Silva, 2001]. The visibility test can be performed using preprocessed data (requiring an organization of the whole geometry), using structures such as *kd-trees* that encapsulate all the scene visibility information. In the case of crowds, since agents are moving around a virtual environment, the visibility test needs to be performed against the static geometry of the scene. However, due to the dynamic nature of the agents, it is not enough to use specific methods that often apply exclusively to static geometry. One efficient way to perform occlusion culling is to first render the scene and then render the agents discarding the occluded fragments with a simple depth test. Notice that this still requires sending them to the graphics card to be rendered since culling is performed in the GPU.

Another possibility to significantly increase performance is to use hardware-based occlusion queries [Wimmer and Bittner, 2005] by sending simpler geometry to the GPU such as bounding volumes. This allows for conservative culling,

since discarding objects whose bounding volume is completely occluded is safe, but we might fail to discard some invisible agents.

When having a very large crowd, and depending on the kind of camera motion we have (for example a first person camera), it is very common to have agents occluding other agents. The early z-culling [Mitchell and Sander, 2004] feature, implemented in most of the recent GPUs, allows for a fragment to be discarded before it is processed by the fragment shader (although some practices, like modifying the fragment's depth programmatically, disable this feature). One could think that depth test and early z-culling should be enough to handle these situations and avoid rendering thousands of agents, but depth algorithms depend highly on the order in which primitives are rendered. Clustering techniques or spatial data structures allowing an efficient front-to-back traversal of the crowd agents, such as KD-trees or BSP-trees, can benefit from early z-culling. We must remember though, that crowd agents will be moving around the environment, and therefore these structures should be dynamically updated every frame according to the new positions of the agents. Hernández et al. use the new transform feedback mechanism of modern GPUs to perform view frustum culling efficiently [Hernández and Rudomin, 2011].

3.3.9 . Level-Of-Detail (LOD)

A well-known crowd acceleration technique is level-of-detail rendering [Clark, 1976, Luebke et al., 2002], where the appropriate representation of each character is chosen according to its image contribution [Pratt et al., 1997]. The basic idea is that, as characters are placed farther away from the camera, less details can be perceived on their screen projection and thus simpler, cheaper representations can be used (see Figure 2.15). Ulicny et al. [Ulicny et al., 2004] replaced full geometrical models with lower resolution ones, and were able to create complex scenes with thousands of characters. Pettre et al. [Pettré et al., 2006] significantly improved performance by using four discrete level-of-detail meshes for the humans in their crowd.

3.3.9 . Generation

In this section we discuss only mesh simplification techniques for character animation, since alternative representations have been discussed above. The automatic generation of simplified models is a long-standing problem, and many algorithms have been proposed in the literature for the general case of static meshes.

Progressive meshes [Hoppe, 1996] use edge collapse operations to dynamically and progressively reduce the geometric complexity of the mesh. The method stores the edge collapses in order (which are invertible operations) in a data structure, and allows a smooth choice of the level of detail desired. Vertex clustering methods [Rossignac and Borrel, 1993] group vertices of the input mesh according to a surrounding grid, and then discard the resulting degenerate triangles. Although they can work with any input mesh, they poorly preserve its details. Unfortunately, all these conventional mesh simplification techniques typically focus on position-based meshes only, ignoring important animation features such as blend weights and indices. Thus they perform poorly on animated characters as they introduce animation artifacts due to the loss of vertices in important areas and discontinuities on their attributes. With less vertices, even the best skinning techniques can produce inconsistent deformations and artifacts. Larkin et al. [Larkin and O'Sullivan, 2011] explored the perception of texture, silhouette and lighting artifacts of the different character's LODs. Even recent progressive encoding schemes like the POP buffer [Limper et al., 2013] are not suitable for mesh animations.

Only a few works address the problem of simplifying animated characters. Schmalstieg and Furhmann's approach [Schmalstieg and Fuhrmann, 1999] break the mesh surface into single bone regions (for vertices associated only to one bone) and additional regions for multiple weighted bones. These surfaces were simplified separately and then stitched together. Some methods try to minimize simplification errors from not just the resting pose, but a set of example poses [DeCoro and Rusinkiewicz, 2005, Mohr and Gleicher, 2003]. Other approaches work with a model and a set of frames with the deformed vertex positions, to build a multiresolution hierarchy, letting the surface change topology for each frame, and therefore showing a simplified surface for each frame [Huang et al., 2006, Kircher and Garland, 2005, Payan et al., 2007, Zhang and Wu, 2007].

The main problem of all these methods is that they are limited to a predefined animation set. Landreneau and Schaefer [Landreneau and Schaefer, 2009] proposed an edge collapse method guided by an error metric that measures deviation from the original deformed shape considering positions and weights. They produce not only new vertices, but also skin weights, and therefore can create keyframes for new animations with new poses from the original ones. This underlies more memory consumption for each additional animation. Willmott [Willmott, 2011] based his method on vertex clustering and was able to handle attribute discontinuities and preserve animation features.

Applications using LOD techniques also need to decide the number of representations to encode (with implications both in memory footprint and potential popping artifacts when switching representations). The encoding of the different LODs of an avatar is also critical to minimize the number of state changes. For example, if the different LOD levels are just reduced versions of the same geometry, and share attribute names, shaders and textures, it makes sense to store the different meshes in the same data structure or to have access to all of them from the same class instance in order to just change the draw call and share the same state.

3.3.9 . Tessellation

Figure 3.35: Tessellation allows rendering large crowds of characters with extreme details in close-up (left). The same character without using tessellation looks significantly less detailed (right). [Tatarchuk et al., 2008]

Tessellation shaders, available in consumer graphics cards since 2011, are able to subdivide faces and thus add detail to existing models without increasing the memory footprint or requiring additional bandwidth. Tessellation shaders

can be applied to crowd rendering to generate high-quality representations on-the-fly from a coarse mesh, in a view-dependent manner. Multiple dynamic levels of detail are possible by adjusting the tessellation levels used by the tessellation primitive generator. An interesting fact about tessellation is that when animating a tessellated character, only the base original vertices are transformed, so the new ones are generated inside the transformed ones. This provides higher resolution for animated models with the same vertex shader performance. Tatarchuk et al. [Tatarchuk et al., 2008] applied these concepts to obtain highly detailed characters for close-up views (see Figure 3.35). Tessellation can be used in combination with displacement maps to add geometry detail rather than to smooth surfaces. In this case, texture seams can lead to noticeable artifacts unless conveniently handled [Tatarchuk et al., 2008].

3.3.9 . LOD Selection

Assuming multiple LODs are available, the application has to decide the most suitable LOD to render for each agent, when to switch from one LOD to another, and how to do this switch as seamlessly as possible to avoid popping artifacts. Early LOD selection techniques relied on object-space distance thresholds defining ranges for each LOD. A better approach is to select the LOD according to (a conservative estimation of) the area of the screen projection of the character. Hardware occlusion queries report pixel counts and thus can be used also to select a proper LOD considering not only screen-projected area, but also partial visibility (if an object is hardly visible, a lower resolution model can be used). However, naive occlusion queries often have a detrimental effect on performance unless more sophisticated techniques exploiting temporal and spatial coherence of visibility are used [Mattausch et al., 2008].

Zach and Karner presented an automatic and dynamic selection of LOD by computing an estimation of the rendering cost and the perceptual benefits [Zach et al., 2002]. Hernández et al. implemented a dynamic LOD selection and view frustum culling on the GPU, by using the new transform feedback mechanism [Hernández and Rudomin, 2011]. More recently Toledo et. al. implemented a tiling of the environment, tagging each agent with the code of the tile they are occupying, and making it easy and fast to dynamically update it

[Toledo et al., 2014]. The tagged code allows for a fast distance computation to the camera and LOD selection.

A related problem is when and how to switch from one LOD to another, as this has a significant impact on visual quality [Southern and Gain, 2003]. If LODs are selected based on a range of distances, all agents crossing a border line immediately exhibit a LOD, and thus potential pop-up effects will be amplified to one zone and therefore more likely to be noticeable. On the other hand, an agent walking over such border lines will be constantly switching between two LODs and might easily catch the eye. A solution is to add a minimum validation time, biased with a random offset, that the agent needs to be in the new zone before further switches are allowed. The random offset allows more unpredictable and asynchronous changes of LODs. Another possibility to reduce pop-up artifacts is to blend the renders of two LODs in the vicinity of threshold values. This solution has the potential to eliminate popping artifacts, at the expense of rendering two representations instead of one (at least for a subset of the characters), and thus reducing performance.

3.3.10 . Hardware Improvements

When having crowds with hundreds or thousands of agents, it is often desired to have each agent to play different animations with different poses. Although animations have a small memory footprint, the computation of all the blending poses can become a major bottleneck and ruin performance. With the introduction of programmable pipelines, a number of costly CPU computations were moved to the GPUs. Beeson and Bjorke highly accelerated the skinning process by computing it directly in the GPU [Beeson and Bjorke, 2004]. Skinning in the GPU requires transfer of both the original vertices of the avatar and the set of matrices of each animation frame. The final pose is a set of matrices that must be used to transform all the avatar vertices in the vertex shader, so the GPU bandwidth is critical.

3.3.10 . Instancing and Pseudo Instancing

Primitive instancing [Carucci, 2005] optimize rendering by drawing multiple copies of an object using a single call. Through instancing, the graphics processor deals with per-instance geometry transformations and appearance modifications, releasing the main processor from this task. A GPU acceleration crowd rendering is presented by Millan and Rudomin in [Millan and Rudomin, 2006b], alternating the use of impostors with polygonal meshes drawn through pseudo-instancing (see Figure 3.36).

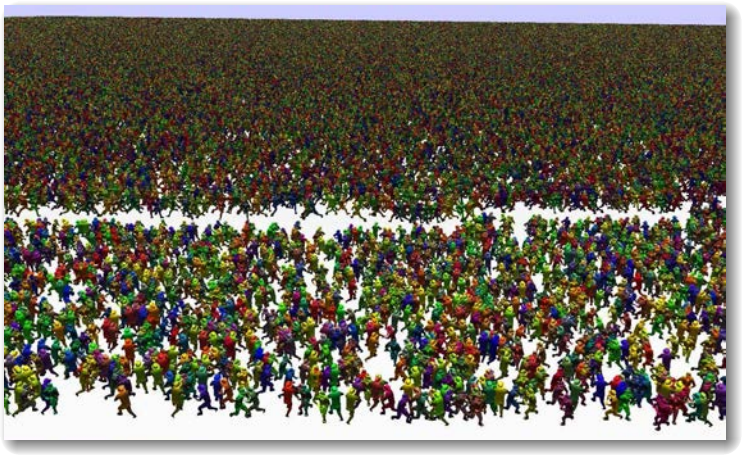


Figure 3.36: One million characters rendered with pseudo-instancing. [Millan and Rudomin, 2006b]

Even when instancing is originally designed for static objects, a similar technique may be used to render large crowds of animated characters. To achieve this goal, a pseudo-instancing technique is used, where geometry is updated on every animation frame and sent to the graphics memory to be used later for rendering nearby characters. Pseudo-instancing takes advantage of the efficiency of using persistent vertex attributes, such as color or transformations, to provide information for an entire instance.

However, this model update implies copying information into graphics memory. Therefore, to maximize the outcome of this technique, several copies of the same object must be rendered in every frame. This is a problem when using animated models, since every different animation pose needs to be sent to the graphics memory. As a workaround, a few poses can be selected, and nearby characters are rendered using the closest pose to the ones selected.

The main difference is that, in instancing, only one call is used to render all primitives, while pseudo-instancing requires one call to a display list to render each instance. However, these calls are very efficient in OpenGL, so similar performance levels are achieved by both techniques.

3.3.10 . Palette Skinning

Matrix palette skinning, introduced by Dudash [Dudash, 2007b], avoids sending to the GPU the transformation matrices for each bone and character instance. In matrix palette skinning, bone matrices for each frame and for each animation are stored in graphics memory. This allows each agent to have its own distinct pose and animation [Dudash, 2007a]. Note however that palette skinning only saves memory bandwidth; it does not affect the number of matrix operations in the vertex shader.

3.3.10 . Dynamic Caching

Recently, Lister et al. [Lister et al., 2010] improved the efficiency of linear-blend skinning by using the temporal and intra-crowd coherencies that are inherent within populated scenes. They achieved it through the allocation of a small geometry cache within which transformed key-poses can be stored. These key-poses are then re-used by multi-pass rendering, between multiple agents and across multiple frames.

The cache of skinned key-poses is a maintained fixed-sized cache, from which crowd members can be reconstructed by interpolation. Generic poses may be shared amongst crowd members to significantly reduce the number that must be stored.

This cache size becomes also a trade off between the rendering performance and the memory usage, because it is the number of characters that have the key-poses stored in the cache that will have the greatest effect on the rendering performance. Clearly, the choice of which key-poses to store is critical to maximize the potential of the approach. Since this is a NP-hard problem, they present a greedy algorithm suitable for real-time applications.

FIGURE 3.37: Dynamic caching accelerates the rendering of an animated crowd. [Lister et al., 2010]

3.3.11 . INCREASING REALISM

Elements such as lighting, shadowing, clothing and variability increase realism at the expense of some performance overhead. Again, we focus on techniques suitable for real-time crowd rendering.

Lighting and shading improve the way we perceive the characters. Jarabo et al. recently made a perception study on how important lighting is for the overall perceived realism of dynamic scenes [Jarabo et al., 2012]. Self-shadowing and self-inter-reflection can help the human eye to interpret the animation and expression of the avatars. Two main techniques are well-known for casting shadows: shadow maps and shadow volumes. Williams introduced shadow maps [Williams, 1978], using a depth map build from each light source to determine whether a fragment is illuminated or not. Two main problems arise from this. First, since textures are used, aliasing problems appear. Second, an additional render of the scene is required for each light source, since animated characters invalidate precomputed shadow maps. Shadow volumes, introduced by Crow [Crow, 1977], are more expensive to calculate, but resolve the aliasing problems by using a semi-infinite frustum. This frustum is extended back from the silhouette of the object away from the light. As mentioned in 3.3.3.2, shadows for animated crowds can be rendered using impostors [Tecchia et al., 2002] (see Figure 3.38), but this technique does not support characters casting shadows onto each other. For more information about real-time shadow techniques see [Eisemann et al., 2013].

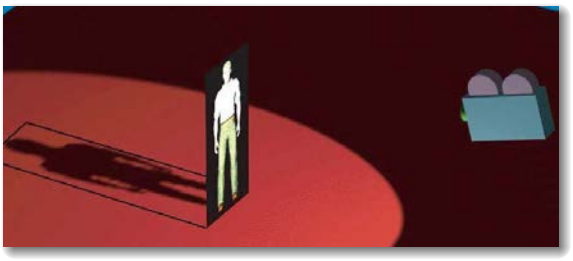


FIGURE 3.38: The shadow of each agent is the projection onto the ground of the character's silhouette. [Tecchia et al., 2002]

Traditionally 3D characters are modeled clothed, with their clothes being part of the human mesh since having them modeled as separate elements and adding clothing simulation is very expensive. Detecting collisions with the human body [Chen et al., 2013] is prohibitively expensive for real-time crowds. McDonnell et al. [McDonnell et al., 2006] perceptually evaluated different LOD representations of humans wearing physically simulated clothing. They show that impostors can depict the deformation properties of clothing. Some recent games include cloth simulation for the main character or a small amount of them, but real-time cloth simulation for crowds is beyond the state-of-the-art.

A related problem is to give each agent of the crowd an individual aspect. McDonnell et al. [McDonnell et al., 2008] performed perceptual studies to determine which aspects are more critical to identify clones. The ideal would be to have unique instances of each avatar, but due to obvious memory and modelling budgets, repetitions are inevitable. Some approaches attempt to add variability and create new differentiated instances of the same base character. Maim et al. [Maim et al., 2009a] proposed a method for attaching accessories to individual agents, and a generic technique for adding detailed color variety and patterns, by using segmentation maps over the human and accessory meshes. Their approach is scalable for all levels of detail, including impostors. McDonell et al. [McDonnell et al., 2009] use selective color variation to generate the illusion of variety as full color variation (see Figure 3.39). Lister et al. also [Lister et al., 2010] add geometric diversity using tangent space morph targets.

FIGURE 3.39: Crowd using two template models with colour, texture and accessory variation. [McDonnell et al., 2009]

3.3.12 . COMPARISON

Table 3.1 summarizes the crowd rendering approaches discussed in previous sections. We classify and compare techniques considering the underlying representation and animation technique. We have also summarized the limitations of each approach, highlighting the parameter or element that can be considered as the one causing the trade-off between visual quality and performance. In the latter columns we evaluate (as high, medium or low) the limitations of each method in terms of memory cost, visual artifacts and computing cost.

The explanation of each column follows:

- **Type:** Geometry-based, Image-based or Hybrid.
- **Representation:** The geometric representation(s) used to render each agent.
- **Animation:** The animation technique used to animate each agent.
- **Tradeoff:** The main parameter of the approach that implies a tradeoff between visual quality and performance.
- **Limitations:** The main limitations of the approach in terms of memory, visual quality and rendering cost, and other aspects with a high impact on the final results or the implementation.
  - **Memory:** We give the required memory cost of each approach by pointing which parameters it depends on:

    - Per agent type (A)
    - Per geometry complexity (G)
    - Per number of frames (F)
    - Per number of views, if it has a discrete number of views, like planar impostors (V)
    - Per number of joints (J)
    - Per texture resolution (R)
  - **Artifacts:** We classify visual and animation artifacts in three categories:
    - Image Quality: blocky aspect (Bl), pixelization (Pix), cracks or gaps (Crk), and animation artifacts due to inconsistent geometry deformation ($Anim_D$).
    - Temporal Discontinuity: popping when changing of LOD (Pop_L), when changing of view point (Pop_V), when changing of frame (Pop_F).
    - Spacial Consistency: visibility or occlusion problems with the scene ($Occl_S$), and with the agent itself ($Occl_A$).
  - **Cost:** The global computational cost of rendering a crowd is affected by some of the following elements:
    - Per agent type (A)
    - Per total number of vertex operations (V_x)
    - Per total number of fragments ($Frag$)

3.3.13 . Crowd Rendering Conclusions

We have reviewed and compared a large number of crowd rendering approaches. Overall, each technique falls somewhere within the triangle representing the quality/memory/performance trade-offs. Geometry-based approaches offer the best visual quality, but their performance depends strongly on the number of polygons per agent. Current GPU techniques such as instancing, palette skinning and dynamic caching can help to speed up the rendering, but as the size of the crowd increases, performance will be severely affected.

						Limitations			
Approach	**Reference**	**Type**	**Representation**	**Animation**	**Tradeoff**	**Memory**	**Artifacts**	**Cost**	**Other**
LOD	[Pratt et al., 1997]	Geometry	Mesh	Skeletal	Distance or pixel size	$A \times G$	Bl, $Anim_D$, Pop_L	V_x	Simplification problem
Dynamic Impostors	[Aubel et al., 1998]	Image	Oriented billboard	Skeletal	Refresh criterium		Pix, $Occl_S$, Pop_V, Pop_F	A	Limited reusability
Pre-generated Impostors	[Tecchia and Chrysanthou, 2000]	Image	Oriented billboard	Texture cyclical	# of views	$A \times F \times V \times R$	Pix, Pop_V, Pop_F, $Occl_S$	A	
Point-based Impostors	[Wand and Straßer, 2002]	Hybrid	Triangle and Point Mesh	Mesh cyclical	Distance or pixel size	$A \times G$	Bl, Pix, Pop_L	V_x	Aliasing
Static Geometry	[Ulicny et al., 2004]	Geometry	Oriented billboard and mesh	Mesh cyclical	Distance or pixel size	$A \times G \times F$	Pop_F	V_x	
Geopostors	[Dobbyn et al., 2005]	Hybrid	Oriented billboard and mesh	Texture cyclical and skeletal	Distance or pixel size	$A \times F \times V \times R$	Pop_L, Pop_V, Pop_F, $Occl_S$	A	
Pseudo-Instancing	[Millan and Rudomin, 2006b]	Geometry	Mesh	Skeletal	#Meshes in the GPU	$A \times F \times V \times R$	Pix, Pop_V, Pop_F	A	Too much data in the GPU
Volumetric Layered Impostors	[Coic et al., 2007]	Hybrid	Oriented layered billboards and Mesh	Texture cyclical and skeletal	Distance or pixel size	$A \times F \times V \times R$	Pop_V, Pop_F, $Occl_S$	A	
Polypostors	[Kavan et al., 2008b]	Hybrid	1 Oriented billboard per body part, and mesh	Texture deformation and skeletal	Distance or pixel size	$A \times F \times V \times R$	Crk, Pop_V, $Occl_A$	A	
Dynamic Caching	[Lister et al., 2010]	Geometry	Mesh	Closest pose (skeletal)	Caché size	$A \times G$	Pop_L, Pop_F	A	
Relief Per-Joint Impostors	[Beacco et al., 2011]	Hybrid	6 relief impostors per body part, and mesh	Skeletal	Distance or number of fragments	$A \times J \times R$	Crk, Pop_L	$Frag$	High per fragment cost
Flat Per-Joint Impostors	[Beacco et al., 2012]	Hybrid	One oriented billboard per body part, and mesh	Skeletal	Distance and # of view angles	$A \times V \times J \times R$	Pop_L, Pop_V	A	
Hierarchical Point-Based	[Toledo et al., 2014]	Hybrid	Octree per limb, and mesh	Skeletal	Distance and hierarchy level	$A \times J$	Crk	A	

TABLE 3.1: Comparison of the limitations of the main approaches on crowd rendering in the literature. *Quick reference: MEMORY:* **A**: agent type; **G**: geometry complexity; **F**: number of frames; **V**: number of views; **J**: number of joints; **R**: texture resolution. *IMAGE QUALITY:* **Bl**: blocky aspect; **Pix**: pixelization; **Crk**: cracks; $\mathbf{Anim_D}$: animation deformation; $\mathbf{Pop_L}$: popping changing of LOD; $\mathbf{Pop_V}$: popping changing of view; $\mathbf{Pop_F}$: popping changing of frame; $\mathbf{Occl_S}$: occlusion problems with the scene; $\mathbf{Occl_A}$: occlusion problems with the agent itself. *COST:* **A**: agent type; $\mathbf{V_x}$: vertex operations; **Frag**: number of fragments

There is currently no standard solution for representing far away characters, though most crowd rendering systems use at least a different representation for these. LOD techniques are commonplace to improve performance as the number of agents increases. Some issues specific to character rendering are how to generate simplified models of avatars that need to be deformed by animations, and how to switch between LODs without popping artifacts.

Using only points as primitives reduces the rendering cost, but the animation can suffer from visual artifacts when drastically reducing the number of primitives. This problem can be alleviated by proper point sampling, at the expense of having a different sampling for each keyframe.

Image-based representations offer the highest performance. We can trade off memory for quality to provide a better sampling of view directions and animation frames to minimize popping artifacts, but in practice image-based representations are suitable only for distant characters.

Hybrid approaches combine mesh-based and image-based or point-based representations to display characters at different viewing distances. Special care must be taken to prevent visual artifacts when switching from one representation to another. Hybrid approaches have now reached a finer granularity, from using a single texture for the whole character, to using per-joint impostors at different skeleton levels, but it is unclear how much further these techniques can go in adding details while still being more efficient than the original geometry.

From an implementation point of view, all the new hardware improvements such as instancing need to take important considerations into account. Appropriate grouping data structures for avatars, textures and animations, and minimizing 3D API state changes are critical to get all the benefits of these improvements. The general trend is to move as many computations as possible to the GPU, achieving higher parallelization of per-agent computations and releasing the CPU for other tasks. Skinning and animation blending can be performed efficiently in the GPU. As we free the CPU from rendering and animation tasks, the CPU can spend more resources to the crowd simulation, although lately there has been a tendency towards moving also some simulation tasks to the GPU. We will probably reach a point where all crowd simulation, animation and rendering will be performed in the GPU at different shader stages. In this

scenario, GPU memory and CPU-GPU bandwidth could become major limiting factors.

Adding shadows to the crowd scene increases realism, but at the high cost of having to render the scene for each light. Impostors can be used to do so and have approximate shadows, although they do not support self shadows. Clothing the characters also adds realism but there is still no physical clothing simulation fast enough for crowd rendering. Repeating instances of the same avatars is inevitable, specially to benefit from instancing and similar techniques, but some variety can be added to the crowd by attaching different accessories or by adding color variety and editable patterns to some base meshes.

3.4 . Existing Tools for Crowds

During the course of this thesis, the author has make use of multiple tools, engines and libraries. Due to the real-time nature of our problem, and the willing of integrating simulation, animation and rendering, it has not always been straight forward to deal with them. This section tries to make an overview of the existing tools that can be used in order to perform a research on crowds, or to add crowds to some existing system.

3.4.1 . Commercial Solutions

There are some well known commercial tools for modeling and simulation software such as 3D Studio Max [Autodesk, 2014a], Maya [Autodesk, 2014c] or Blender [Blender-Foundation, 2014] that allow us to add crowds to a virtual scene. There exist also many plug-ins for these packages that can be used to extend their basic features. For example Golaem Crowd [Goalem, 2014] is a powerful plug-in for Maya [Autodesk, 2014c]. Golaem Crowd is a complete commercial package for crowd authoring, including tools for placing crowds, create behaviors, animate characters, create diversity of agents, and render the resulting simulations. Although they offer some real-time previsualizations in order to help the artists creating new crowds and defining behaviors, their target is mostly the movie industry producing high quality offline renders. It is not as much a research platform as it is a commercial production tool. A similar package is included with newer versions of 3D Studio Max [Autodesk, 2014a]. Also in the movie industry, a major competitor is Massive [Massive, 2014], an expensive crowd simulation tool that has been used in many films.

3.4.2 . Libraries

In the crowd simulation literature there are some steering behavior libraries such as RVO2 Library [van den Berg et al., 2014] or OpenSteer [Reynolds, 2014], which also includes a demo software for simple visualizations of the implemented steering behaviors through simple 2D representations. We can also

find SteerSuite [Singh et al., 2009], a flexible but easy-to-use set of tools, libraries, and test cases for steering behaviors. The spirit of SteerSuite [Singh et al., 2009] was to make it practical and easy-to-use, yet flexible and forward-looking, to challenge researchers and developers to advance the state of the art in steering. Although their simulation part is very complete, there is not such thing as animated 3D characters for visualization of the simulated crowds.

For character animation, there is just a few libraries that we can easily include in any C++ project. A commercial example would be Granny 3D [RAD-Game-Tools, 2014], which includes a complete animation system including features such as blend-graphs, character behavior, events synchronization or procedural IK. But again it is a solution for artist and for commercial products, not for doing research. Some free solutions are Animadead [Butterfield, 2014], a skeletal animation library with basic functionalities, or Cal3D [Cal3D, 2014] another skeletal animation library which also includes an animation mixer and integrates morph targets. The Hardware Accelerated Library for Character Animation, HALCA [Spanlang, 2009], extends the Cal3D library to include new features such as GLSL shaders support, morph animations, hardware accelerated morph targets (blend shapes), dual quaternion skin shaders, JPEG texture files, direct joint manipulation and other additions. The FBX SDK [Autodesk, 2014b] is another library allowing to read FBX files, which is a widely used and extended format for character modeling and animation. The API includes some skinning examples, but you should still program your animation library to use FBX models imported with it. Our framework currently offers HALCA [Spanlang, 2009] as the animation library, but the user can create its own characters based on any other library.

3.4.3 . Engines

In the case of applications that require real time crowd simulation, such as video games, there are several tools commonly used both commercial (Unreal [Epic, 2014], Unity [Unity, 2014] and GameBryo [Gamebase-USA, 2014]) and open source (Ogre [Ogre, 2014] and Panda 3D [Carnegie-Mellon-University, 2014]). Unreal [Epic, 2014] is a widely licensed game engine in the professional video game industry, with powerful and refined tools. Unity [Unity, 2014] is a newer game engine that is also used for professional games, although it is more

widely extended in the indy game development community. It offers a render engine, a complete animation system called Mecanim, with a very user-friendly interface for authoring state machines (such as blend trees) and retargeting capabilities. Mecanim also includes modules for steering behavior and navmesh generation. It is relatively easy to start a project and learn how to work with it, and a lot of researchers are starting to use it. But it still remains a commercial game engine, and you have to develop your extensions using scripts. You can use your own C++ code, which is faster, but you need to implement plug-ins and wrappers for them. Gamebryo [Gamebase-USA, 2014] is a similar product, modular and extendable, but still focused on game design. Ogre [Ogre, 2014] and Panda 3D [Carnegie-Mellon-University, 2014] are open source graphic render engines that include some features like animation systems or simple AI modules, which can be easily extended.

With most of the systems described above, you will find severe limitations when trying to scale up your work. For instance you may have developed a new rendering technique for thousands of deformable characters, but the selected engine may only animate a few hundred in real time. Using Unity [Unity, 2014] you might be restricted to a fixed renderer and to its animation system, unless you implement your own using plug-ins. Implementing and integrating rendering plug-ins to Unity is possible although not straight forward, and might require a pro-license. The rendering pipeline in Unity is not always clear, and our experience says you can not completely control the OpenGL state. Ogre [Ogre, 2014] has more potential, and has a basic animation system, which should be improved in newer versions, but you still need to integrate it with your AI libraries. The learning curve of the Unreal Engine [Epic, 2014] is hard, and might not be worth if you are aiming for research and not for a professional and commercial appealing result.

3.4.4 . Research Platforms

None of the previous solutions offers a flexible yet customizable framework for the research community to work with when it comes to real time requirements, giving them freedom to modify either simulation, animation, rendering or any combination of these parts. In addition to that you might not be willing to pay expensive licenses if you are only targeting research applications. There is

though some research platform in the crowd simulation field that are worth to mention.

CAROSA [Allbeck, 2010] is an architecture to author the behavior of agents, and obtain heterogeneous populations inhabiting a virtual environment. Its framework enables the specification and control of actions, and is able to link human characteristics and high level behaviors to animated graphical depictions. Although it does not include research tools for rendering, it is prepared to be used on an external software. ADAPT [Shoulson et al., 2013] is an open-source Unity library delivering a platform for designing and authoring functional, purposeful human characters in a rich virtual environment. Its framework incorporates character animation, navigation, and behavior with modular interchangeable and extensible components. But since it is a library for Unity, it does not allow you to control rendering. Project Metropolis [O'Sullivan and Ennis, 2011] aims to create the sight and sounds of a convincing crowd of humans and traffic in a complex cityscape. They also focus on exploring the perception of virtual humans and crowds, through psychophysical experiments with human participants. But Metropolis is a large and complicated research project, with tens of research goals, rather than a tool for researchers to get started working in the crowd simulation field. Similar to what we aim to do, SmartBody [Shapiro, 2014] is a character animation library that provides synchronized locomotion, steering, object manipulation, lip syncing, gazing and nonverbal behavior in real-time. It uses Behavior Markup Language (BML) to transform behavior descriptions into real-time animations. SmartBody is a good tool to develop and explore virtual human research and technologies, but it is focused on one character, or a small number of characters, and not on crowds as it is our desire.

3.4.5 . Conclusions on Existing Tools for Crowds

Achieving simulation, animation and rendering of crowds in real-time is a major challenge. Although each of these areas has been studied separately and improvements have been presented in the literature, the integration of these three areas in one real-time system is not straight forward. There are some commercial tools that provide aids to simulate crowds, but in most cases there

are limitations that cannot be overcome, such as finding bottlenecks in a different area than the one you are researching on, thus pushing you from meeting real time constraints as we increase the size of the crowd.

Due to the high computational needs of each of these three areas individually, the process of integrating animation, simulation and rendering of real time crowds, often presents trade-offs between accuracy and quality of results. But these three areas cannot be treated in a completely separated way as most current tools do, since there is a strong overlapping between them, and users need to be aware of this when setting up a simulation. For example, we cannot increase the number of animations easily if we are rendering exclusively with impostors.

Currently it is not easy to find a real time framework that allows you to easily work on one of these areas. If for example you want to focus your research on a new steering behavior, you will start and do most of your experiments by visualizing a set of circles or cylinders representing your different agents. But in the end you would like to see your results represented by 3D animated characters. The switch from having 2D circles to fully articulated 3D characters in real time can be very time consuming. Or for example, your research interest may be focused on implementing a new representation for rendering thousands of characters. Once you achieve real time visualization of such a huge amount of characters, you do not want to end up displaying them in a grid formation or giving the agents random positions where they stay in place. Instead, one would want them to be animated and moving in a virtual environment, if possible with collision avoidance and natural animation. Again this is not straight forward in current frameworks. And finally, the same applies for the animation field, if you are defining new animation controllers for 3D characters, you would like to test them with hundreds of characters moving around a virtual environment with realistic rendering.

We thus believe the graphics community lacks of a specific tool in order to be able to quickly get started on a new research project related to crowd animation, visualization or simulation. In chapter 7 we present an attempt to do that, with a new prototyping testbed for crowds that lets the researcher focus on one of these areas of research at a time without loosing sight of the others.

Publications and More

A recent tutorial covers in more detail some of these and other aspects of the crowd research [Pelechano et al., 2014], including some of the contributions of this thesis. Also, the contents of the state of the art on crowd rendering have been submitted as a survey for a journal publication, and are currently under revision.

4 . Contributions to Crowd Simulation

This chapter presents our two contributions to the crowd simulation area. The first one is a real-time planning framework, that enables the use of multiple heterogeneous problem domains of differing complexities for navigation in large, complex, dynamic virtual environments. The original navigation problem is decomposed into a set of smaller problems that are distributed across planning tasks working in these different domains. An anytime dynamic planner is used to efficiently compute and repair plans for each of these tasks, while using plans in one domain to focus and accelerate searches in more complex domains. We demonstrate the benefits of our framework by solving many challenging multi-agent scenarios in complex dynamic environments requiring space-time precision and explicit coordination between interacting agents, by accounting for dynamic information at all stages of the decision-making process.

One of those domains could be defined by just 8 possible directions of root movement, but another domain of higher resolution could include all the possible footsteps reachable by the character. In our second contribution we present a planner that given any set of animation clips outputs a sequence of footsteps to follow from an initial position to a goal such that it guarantees obstacle avoidance and correct spatio-temporal foot placement. We use a best-first search technique that dynamically repairs the output footstep trajectory based on changes in the environment. We show results of how the planner works in different dynamic scenarios with trade-offs between accuracy of the resulting paths and computational speed, which can be used to adjust the search parameters accordingly.

4.1 . Planning in Multiple Domains

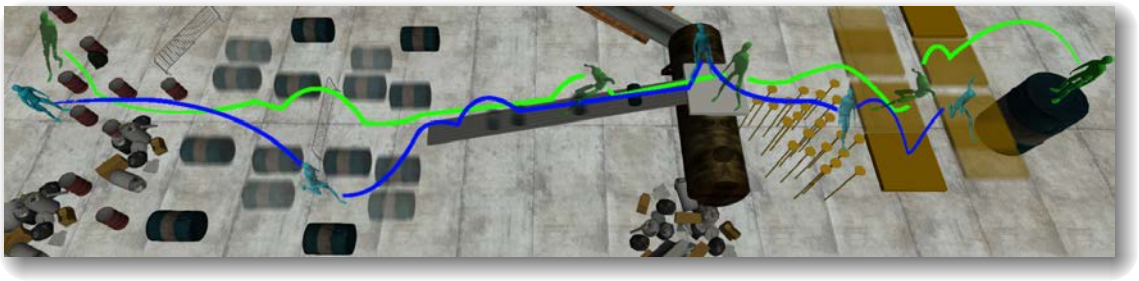


Figure 4.1: Two agents navigating with space-time precision through a complex dynamic environment.

In this section we propose a real-time planning framework for multi-character navigation that uses multiple heterogeneous problem domains of differing complexities for navigation in large, complex, dynamic virtual environments. We define a set of problem domains (spaces of decision-making) which differ in the complexity of their state representations and the fidelity of agent control. These range from a static navigation mesh domain which only accounts for static objects in the environment, to a space-time domain that factors in dynamic obstacles and other agents at much finer resolution. These domains provide different trade-offs in performance and fidelity of control, requiring a framework that efficiently works in multiple domains by using plans in one domain to focus and accelerate searches in more complex domains.

A global planning problem (start and goal configuration) is dynamically decomposed into a set of smaller problem instances across different domains, where an anytime dynamic planner is used to efficiently compute and repair plans for each of these problems. Planning tasks are connected by either using the computed path from one domain to define a "tunnel" to focus searches, or using successive waypoints along the path as start and goal for a planning task in another domain to reduce the search depth, thereby accelerating searches in more complex domains. Using our framework, we demonstrate real-time character navigation for multiple agents in large-scale, complex, dynamic environments, with precise control, and little computational overhead.

Role in this work

This work has been done in collaboration with the University of Pennsylvania, as the result of a stay of 4 months. I was invited to the Center for *Human Modeling and Simulation* [HMS, 2014] by Professor Norman I. Badler, and supervised by Doctor Mubbasir Kapadia. It was during my stay that we started discussing the idea and the fundamentals of this project. I implemented the first prototypes of the different domains, the first planner used and the fist tunnels used. I was also in charge of the animation system. After my stay, the collaboration continued, through several iterations until we finally obtained a publication [Kapadia et al., 2013]. Although Doctor Mubbasir Kapadia was the first author of the paper, since he did the writing and was directing the project, I still was an important part part of it. In the final version of this work, the final planners were implemented by Francisco García. Motion capture animations were obtained by Vivek C. Reddy. Doctor Nuria Pelechano and Professor Norman I. Badler were supervisors. Doctor Mubbasir Kapadia also implemented the tasks planner. My particular role was again in the integration of all elements, the animation system, obtaining figures and screenshots from test scenarios, and video production.

4.1.1 . Overview

The problem domain of a planner determines its effectiveness in solving a particular problem instance. A complex domain that accounts for all environment factors such as dynamic environments and other agents, and has a large branching factor in its action space can solve more difficult problems, but at a larger cost overhead. A simpler domain definition provides the benefit of computational efficiency while compromising on control fidelity. Our framework enables the use of multiple heterogeneous domains of control, providing a balance between control fidelity and computational efficiency, without compromising either.

A global problem instance P_0 is dynamically decomposed into a set of smaller problem instances $\{P^{'}\}$ across different planning domains $\{\Sigma_i\}$. Subsection 4.1.2 describes the different domains, and subsection 4.1.3 describes the problem decomposition across domains. Each problem instance $P^{'}$ is assigned a planning task $T(P^{'})$, and an anytime dynamic planner is used to efficiently compute and repair plans for each of these tasks, while using plans in one domain to focus and accelerate searches in more complex domains. Plan efforts across domains are reused in two ways. The computed path from one domain can be used to define a *tunnel* which focuses the search, reducing its effective branching factor. Each pair of successive waypoints along a path can also be used as start,goal pairs for a planning task in another domain, thus reducing the search depth. Both these methods are used to focus and accelerate searches in more complex domains, providing real-time efficiency without compromising on control fidelity. Subsection 4.1.4 describes the relationships between domains.

4.1.2 . Planning Domains

A problem domain is defined as $\Sigma = \langle \mathbb{S}, \mathbb{A}, c(s,s'), h(s,s_{goal}) \rangle$, where the state space $\mathbb{S} = \{\mathbb{S}_{self} \times \mathbb{S}_{env} \times \mathbb{S}_{agents}\}$ includes the internal state of the agent $\mathbb{S}_{self}$, the representation of the environment $\mathbb{S}_{env}$, and other agents $\mathbb{S}_{agents}$. $\mathbb{S}_{self}$ may be modeled as a simple particle with a collision radius. $\mathbb{S}_{env}$ can be an environment triangulation with only static information or a uniform grid representation with dynamic obstacles. $\mathbb{S}_{agents}$ is defined by the vicinity within which neighboring agents are considered. Imminent threats may be considered individually or just represented as a density distribution at far-away distances. The action space $\mathbb{A}$ defines the set of all possible successors $succ(s)$ and predecessors $pred(s)$ at each state s, as shown in Equation 4.1. Here, $\delta(s,i)$ describes the i^{th} transition, and $\Phi(s,s')$ is used to check if the transition from s to s' is possible. The cost function $c(s,s')$ defines the cost of transition from s to s'. The heuristic function $h(s,s_{goal})$ defines the estimate cost of reaching a goal state.

$$succ(s) = \{s + \delta(s,i) | \Phi(s,s') = \mathsf{TRUE} \quad \forall i\} \tag{4.1}$$

A problem definition $P = \langle \Sigma, s_{start}, s_{goal} \rangle$ describes the initial configuration of the agent, the environment and other agents, along with the desired goal configuration in a particular domain. Given a problem definition P for domain Σ, a planner searches for a sequence of transitions to generate a plan $\Pi(\Sigma, s_{start}, s_{goal}) = \{s_i | s_i \in \mathbb{S}(\Sigma)\}$ that takes an agent from s_{start} to s_{goal}.

4.1.2 . Multiple Domains of Control

We define 4 domains which provide a nice balance between global static navigation and fine-grained space-time control of agents in dynamic environments. Figure 4.2 illustrates the different domain representations for a given environment.

Static Navigation Mesh Domain Σ_1. This domain uses a triangulated representation of free space and only considers static immovable geometry. Dynamic obstacles and agents are not considered in this domain. The agent is modeled as a point mass, and valid transitions are between connected free spaces, represented as polygons. The cost function is the straight line distance between the center points of two free spaces. Additional connections are also precomputed (or manually annotated) to represent transitions such as jumping with a higher cost definition. The heuristic function is the Euclidean distance between a state and the goal. Searching for an optimal solution in this domain is very efficient and quickly provides a global path for the agent to navigate. We use Recast [Mononen, 2009] to precompute the navigation mesh for the static geometry in the environment.

Dynamic Navigation Mesh Domain Σ_2. This also uses triangulations to represent free spaces and coarsely accounts for dynamic properties of the environment to make a more informed decision at the global planning layer. The work in [van Toll et al., 2012] embeds population density information in environment triangulations to account for the movement of agents at the global planning layer. We adopt a similar method by defining a time-varying density field $\phi(t)$ which stores the density of moveable objects (agents and obstacles) for each polygon in the triangulation at some point of time t. $\phi(t_0)$ represents the density of agents and obstacles currently present in the polygon. The presence of objects and agents in polygons at future timesteps can be estimated by querying their plans (if available). The space-time positions of deterministic

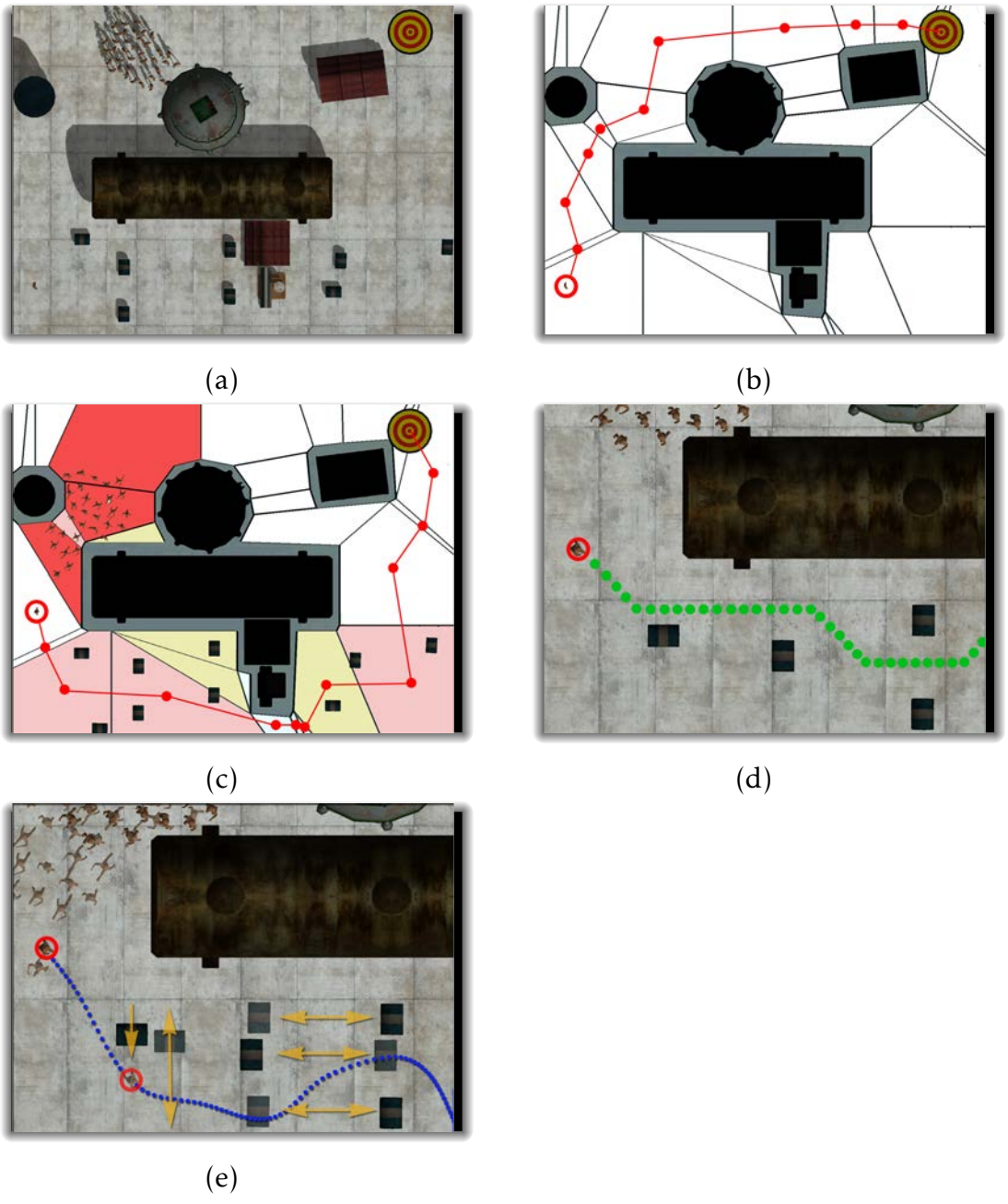


FIGURE 4.2: (a) Problem definition with initial configuration of agent and environment. (b) Global plan in static navigation mesh domain Σ_1 accounting for only static geometry. (c) Global plan in dynamic navigation mesh domain Σ_2 accounting for cumulative effect of dynamic objects. (d) Grid plan in Σ_3. (e) Space-time plan in Σ_4 that avoids dynamic threats and other agents.

objects can be accurately queried while the future positions of agents can be approximated based on their current computed paths, assuming that they travel with constant speed along the path without deviation. $\phi(t)$ contributes to the cost of selecting a waypoint in Σ_2 during planning. The resolution of the triangulation may be kept finer than Σ_1 to increase the resolution of the dynamic information in this domain. Hence, a set of global waypoints are chosen in this domain which avoids crowded areas or other high cost regions.

Grid Domain Σ_3. The grid domain discretizes the environment into grid cells where a valid transition is considered between adjacent cells that are free (diagonal movement is allowed). An agent is modeled as a point with a radius (orientation and agent speed is not considered in this domain). This domain only accounts for the current position of dynamic obstacles and agents, and cannot predict collisions in space-time. The cost and heuristic are distance functions that measure the Euclidean distance between grid cells.

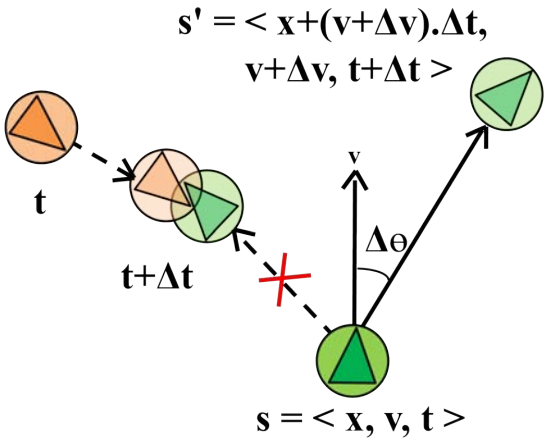


FIGURE 4.3: The Space-Time Domain Σ_4.

Space-Time Domain Σ_4. This domain models the current state of an agent as a space-time position with a current velocity $(\mathbf{x}, \mathbf{v}, t)$. Figure 4.3 shows the schematic illustration of the state and action space in Σ_4, showing a valid transition, and an invalid transition due to a space-time collision with a neighboring agent. The transition function $\delta(s, i)$ for Σ_4 is defined below:

$$\delta(s,i) = \{\Delta\mathbf{v}_i \cdot \Delta t | \Delta\mathbf{v}_i = (\Delta v_i \cdot \sin\Delta\theta_i, \Delta v_i \cdot \cos\Delta\theta_i) \forall i\}$$

where $\Delta v = \{0, \pm a\}$ is the possible speed changes and $\Delta\theta = \{0, \pm\frac{\pi}{8}, \pm\frac{\pi}{4}, \pm\frac{\pi}{2}\}$ is the possible orientation changes the agent can make from its current state. For example, $\Delta\mathbf{v} = a, \Delta\theta = \frac{\pi}{8}$ produces a transition where the agent accelerates by a for the duration of the timestep and rotates by $\frac{\pi}{8}$. The bounds of $\Delta\theta$ are limited between $\{-\frac{\pi}{2}, \frac{\pi}{2}\}$ to limit the maximum rate of turning. Transitions are also bound so that the speed and acceleration of an agent cannot exceed a given threshold. Jumps are additionally modeled as a high cost transition between two space-time points such that the region between them may be occupied or untraversable for that time interval. Inspite of the coarse discretization of $\Delta\theta$, the branching factor of this domain is much higher, providing greater degree of control fidelity with added computational overhead.

Σ_4 accounts for all obstacles (static and dynamic) and other agents. The traversability of a grid cell is queried in space-time by checking to see if moveable obstacles and agents occupy that cell at that particular point of time, by using their published paths. For space-time collision checks, only agents and obstacles that are within a certain region from the agent, defined using a foveal angle intersection, are considered. The cost and heuristic definitions have a great impact on the performance in Σ_4. We use an energy based cost formulation that penalizes change in velocity with a non-zero cost for zero velocity. Jump transitions incur a higher cost. The heuristic function penalizes states that are far away from s_{goal} in both space and time. This is achieved using a weighted combination of a distance metric and a penalty for a deviation of the current speed from the speed estimate required to reach s_{goal}.

The domains described here are *not* a comprehensive set and only serve to showcase the ability of our framework to use multiple heterogeneous domains of control in order to solve difficult problem instances at a fraction of the computation cost. Our framework can be easily extended to use other domain definitions (e.g., a footstep domain).

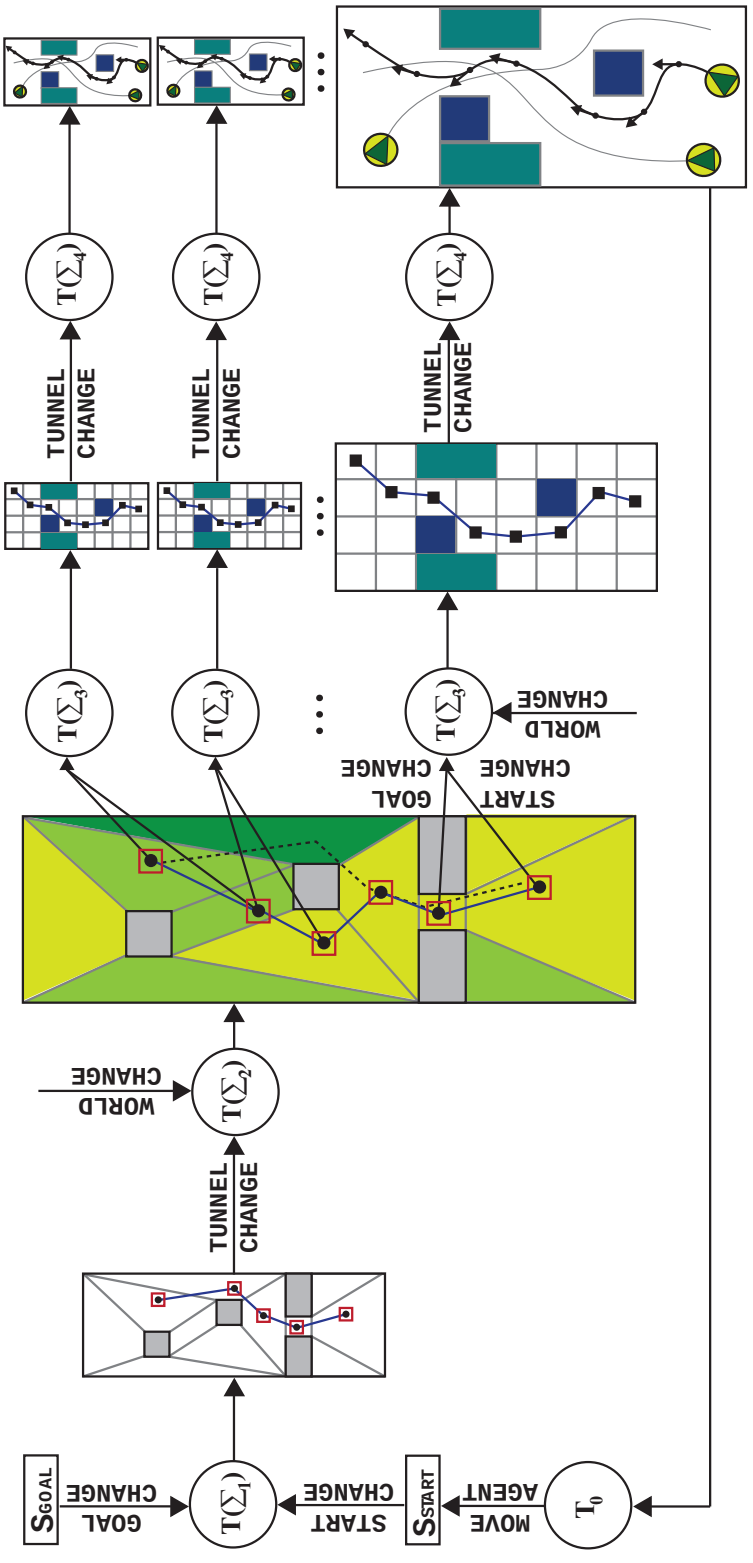


FIGURE 4.4: Expanded illustration of domain relationship shown in Figure 4.5(b). A global problem instance (start and goal state) is decomposed into a set of smaller problem instances across multiple planning domains. Planning tasks $T(\Sigma)$ are assigned to each of these problems and scheduled using a dynamic priority scheme based on events from the environment and other tasks.

4.1.3 . Problem Decomposition and Multi-Domain Planning

Figure 4.5(a) illustrates the use of tunnels to connect each of the 4 domains, ensuring that a complete path from the agents initial position to its global target is computed at all levels. Figure 4.5(b) shows how Σ_2 and Σ_3 are connected by using successive waypoints in $\Pi(\Sigma_2)$ as start and goal for independent planning tasks in Σ_3. This relation between Σ_2 and Σ_3 allows finer-resolution plans being computed between waypoints in an independent fashion. Limiting Σ_3 (and Σ_4) to plan between waypoints instead of the global problem instance ensures that the search horizon in these domains is never too large, and that fine-grained space-time trajectories to the initial waypoints are computed quickly. However, completeness and optimality guarantees are relaxed as Σ_3, Σ_4 never compute a single path to the global target.

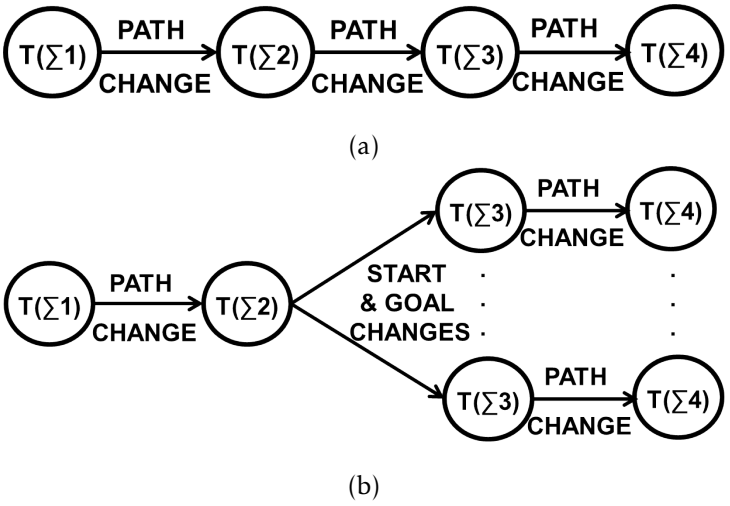


Figure 4.5: Relationship between domains. (a) Use of tunnels to connect each of the 4 domains. (b) Use of successive waypoints in $\Pi(\Sigma_2)$ as start, goal pairs to instantiate multiple planning tasks in Σ_3 and Σ_4.

Figure 4.4 illustrates the different events that are sent between planning tasks to trigger plan refinement and updates for the domain relationship in Figure 4.5(b). Σ_1 is first used to compute a path from s_{start} to s_{goal}, ignoring dynamic obstacles and other agents. $\Pi(\Sigma_1)$ is used to accelerate computations in Σ_2, which refines the global path to factor in the distribution of dynamic objects in the environment. Depending on the relationship between Σ_2 and Σ_3, a single planning task or multiple independent planning tasks are used in Σ_3. Finally, the plan(s) of $T(\Sigma_3)$ are used to accelerate searches in Σ_4.

Changes in s_{start} and s_{goal} trigger plan updates in $T(\Sigma_1)$, which are propagated through the task dependency chain. $T(\Sigma_2)$ monitors plan changes in $T(\Sigma_1)$ as well as the cumulative effect of changes in the environment to refine its path. Each $T(\Sigma_3)$ instance monitors changes in the waypoints along $\Pi(\Sigma_2)$ to repair its solution, as well as nearby changes in obstacle and agent position. Finally, $T(\Sigma_4)$ monitors plan changes in $T(\Sigma_3)$ (which it depends on) and repairs its solution to compute a space-time trajectory that avoids collisions with static and dynamic obstacles, as well as other agents.

Events are triggered (outgoing edges) and monitored (incoming edges) by tasks, creating a cyclic dependency between tasks, with T_0 (agent execution) monitoring changes in the plan produced by the particular $T(\Sigma_4)$, which monitors the agents most imminent global waypoint. Tasks that directly affect the agent's next decision, and tasks with currently invalid or sub-optimal solutions are given higher priority. Given the maximum amount of time to deliberate t_{max}, the agent pops one or more tasks that have highest priority and divides the deliberation time across tasks (most imminent tasks are allocated more time). Task priorities constantly change based on events triggered by the environment and other tasks. For more details on planning tasks, we refer the reader to our paper [Kapadia et al., 2013].

4.1.4 . Relationship Between Domains

The complexity of the planning problem increases exponentially with increase in dimensionality of the search space – making the use of high-dimensional domains nearly prohibitive for real-time applications. In order to make this problem tractable, planning tasks must efficiently use plans in one domain to focus and accelerate searches in more complex domains. Section 4.1.4.1 describes a method for mapping a state from a low-dimensional domain to one or more states in a higher dimensional domain. Sections 4.1.4.2 and 4.1.4.3 describe two ways in which plans in one domain can be used to focus and accelerate searches in another domain.

4.1.4 . Domain Mapping

We define a $1:n$ function $\lambda(s,\Sigma,\Sigma^{'})$ that allows us to maps states in $\mathbb{S}(\Sigma)$ to one or more equivalent states in $\mathbb{S}(\Sigma^{'})$.

$$\lambda(s,\Sigma,\Sigma^{'}) : s \rightarrow \{s' | s' \in \mathbb{S}(\Sigma^{'}) \wedge s \equiv s'\} \tag{4.2}$$

The mapping functions are defined specifically for each domain pair. For example, $\lambda(s,\Sigma_1,\Sigma_2)$ maps a polygon $s \in \mathbb{S}(\Sigma_1)$ to one or more polygons $\{s'|s' \in \mathbb{S}(\Sigma_2)\}$ such that s' is spatially contained in s. If the same triangulation is used for both Σ_1 and Σ_2, then there exists a one-to-one mapping between states. Similarly, $\lambda(s,\Sigma_2,\Sigma_3)$ maps a polygon $s \in \mathbb{S}(\Sigma_2)$ to multiple grid cells $\{s'|s' \in \mathbb{S}(\Sigma_3)\}$ such that s' is spatially contained in s. $\lambda(s,\Sigma_3,\Sigma_4)$ is defined as follows:

$$\lambda(s,\Sigma_3,\Sigma_4) : (\mathbf{x}) \rightarrow \{(\mathbf{x} + W(\Delta\mathbf{x}), t + W(\Delta t))\} \tag{4.3}$$

where $W(\Delta)$ is a window function in the range $[-\Delta,+\Delta]$. The choice of t is important in mapping Σ_3 to Σ_4. Since we use λ to effectively map a plan $\Pi(\Sigma_3, s_{start}, s_{goal})$ in Σ_3 to a tunnel in Σ_4, we can exploit the path and the temporal constraints of s_{start} and s_{goal} to define t for all states along the path. We do this by calculating the total path length and the time to reach s_{goal}. This allows us to compute the approximate time of reaching a state along the path, assuming the agent is traveling at a constant speed along the path.

4.1.4 . Mapping Successive Waypoints to Independent Planning Tasks.

Successive waypoints along the plan from one domain can be used as start and goal for a planning task in another domain. This effectively decomposes a planning problem into multiple independent planning tasks, each with a significantly smaller search depth.

Consider a path $\Pi(\Sigma_2) = \{s_i | s_i \in \mathbb{S}(\Sigma_2), \forall i \in (0,n)\}$ of length n. For each successive waypoint pair (s_i, s_{i+1}), we define a planning problem $P_i = \langle \Sigma_3, s_{start}, s_{goal} \rangle$ such that $s_{start} = \lambda(s_i,\Sigma_2,\Sigma_3)$ and $s_{goal} = \lambda(s_{i+1},\Sigma_2,\Sigma_3)$. Even though λ may return multiple equivalent states, we choose only one candidate state. For each

problem definition P_i, we instantiate an independent planning task $T(P_i)$which computes and maintains path from s_i to s_{i+1} in Σ_3. Figure 4.5 illustrates this connection between Σ_2 and Σ_3.

4.1.4 . Tunnels

The work in [Gochev et al., 2011] observes that a plan in a low dimensional problem domain can often be exploited to greatly accelerate high-dimensional complex planning problems by focusing searches in the neighborhood of the low dimensional plan. They introduce the concept of a tunnel $\tau(\Sigma_{hd}, \Pi(\Sigma_{ld}), t_w)$ as a sub graph in the high dimensional space Σ_{hd} such that the distance of all states in the tunnel from the low dimensional plan $\Pi(\Sigma_{ld})$ is less than the tunnel width t_w. Based on their work, we use plans from one domain in order to accelerate searches in more complex domains with much larger action spaces. A planner is input a low dimensional plan $\Pi(\Sigma_{ld})$ which is used to focus state transitions in the sub graph defined by the tunnel $\tau(\Sigma_{hd}, \Pi(\Sigma_{ld}), t_w)$.

To check if a state s lies within a tunnel $\tau(\Sigma_{hd}, \Pi(\Sigma_{ld}), t_w)$ without precomputing the tunnel itself, the low dimensional plan $\Pi(\Sigma_{ld})$ is first converted to a high dimensional plan $\Pi^{'}(\Sigma_{hd}, s_{start}, s_{goal})$ by mapping all states of Π to their corresponding states in $\Pi^{'}$, using the mapping function $\lambda(s, \Sigma_{ld}, \Sigma_{hd})$ as defined in Equation 4.2. Note that the resulting plan $\Pi^{'}$ may have multiple possible trajectories from s_{start} to s_{goal} due to the $1:n$ mapping of λ. Next, we define a distance measure $\mathbf{d}(s, \Pi(\Sigma))$ which computes the distance of s from the path $\Pi(\Sigma)$. During a planning iteration, a state is generated if and only if $\mathbf{d}(s, \Pi(\Sigma_{hd})) \leq t_w$. This is achieved by redefining the $\mathtt{succ}(s)$ and $\mathtt{pred}(s)$ to only consider states that lie in the tunnel. Furthermore, node expansion can be prioritized to states that are closer to the path by modifying the heuristic function as shown in below.

$$h_t(s, s_{start}) = h(s, s_{start}) + |\mathbf{d}(s, \Pi(\Sigma))| \tag{4.4}$$

Note that the heuristic $h_t(s, s_{start})$ is an estimate of the distance from s to s_{start} since we use a backward search from s_{goal} to s_{start} to accomodate start movement. For spatial domains Σ_1, Σ_2, and Σ_3, $\mathbf{d}(s, \Pi(\Sigma))$ is the perpendicular distance between s and the line segment connecting the two nearest states in $\Pi(\Sigma)$.

$\mathbf{d}(s, \Pi(\Sigma_4))$ will return a two-tuple value for spatial distance as well as temporal distance.

TunnelChangeUpdate. When the tunnel changes, previously visited nodes that are no longer within the new tunnel are assigned an infinite cost and the changes are propagated to their successors. Also, their heuristic values are updated to reflect the new tunnel distance using Equation 4.4, which re-prioritizes node expansion to nodes that are closer to the new path. The tunnel width $t_w$is inversely proportional to the inflation factor ϵ. Thus, a high ϵ focuses the search within a narrow tunnel, which is iteratively expanded when ϵ is reduced to increase the breadth of the search. Due to the extremely dynamic nature of the planning tasks, we find that a reasonably narrow tunnel allows solutions to be returned very quickly which can be improved, if time permits. If the tunnel is too narrow, however, no plan maybe returned, requiring a replan in a wider tunnel.

Completeness and Optimality Guarantees. The use of tunnels enables AD* to leverage plans across domains in order to expedite searches in high-dimensional domains. However; by modifying the definition of succ(s) and pred(s) to prune nodes that lie outside the tunnel, we sacrifice the strict bounds on optimality provided by AD*, as nodes that lie outside the tunnel may lead to a more optimal solution. By iteratively expanding the tunnel width t_w, when the search is unsuccessful, we ensure that a solution will be found, if one exists. For practical purposes, we find that a constantly dynamic world mitigates the need for strict optimality bounds as solutions are constantly invalidated, before their use. In our experiments (Section 4.1.5.1), we find that the computational benefit of using tunnels far outweighs its drawbacks, providing an exponential reduction in the nodes expanded, while still producing reasonable quality solutions.

4.1.5 . Results

4.1.5 . Comparative Evaluation of Domain Relationships

We randomly generate 1000 scenarios of size $100m \times 100m$, with random configurations of obstacles (both static and dynamic), start state, and goal state and record the effective branching factor, number of nodes expanded, time to compute a plan, success rate, and quality of the plans obtained. The effective branching factor is the average number of successors that were generated over the course of one search. Success rate is the ratio of the number of scenarios for which a collision-free solution was obtained. Plan quality is the ratio of the length of the static optimal path and the path obtained. A plan quality of 1 indicates that the solution obtained was able to minimize distance without any deviations. Similar metrics for analyzing multi-agent simulations have been used in [Kapadia et al., 2011b]. The aggregate metrics for the different domains and domain relationships are shown in Table 4.1. Rows 3 and 6 in Table 4.1 include the added time to compute plans in earlier domains for tunnel search, to provide an absolute basis of comparison. All experiments were performed on a single-threaded 2.80 GHz Intel(R) Core(TM) i7 CPU.

Σ_1 and Σ_2 can quickly generate solutions but is unable to solve most of the scenarios as they don't resolve fine-grained collisions. The use of plans from Σ_1 accelerates searches in Σ_2 (Table 4.1, Row 3). However, the real benefit of using both Σ_1 and Σ_2 is evident when performing repeated searches across domains in large environments when an initial plan $\Pi(\Sigma_1)$ accelerates repeated refinements in Σ_2 (and other subsequent domains). Using Σ_3 in a large environment takes significantly longer to produce similar paths. Σ_4 is unable to find a complete solution for large-scale problem instances (we limit maximum number of nodes expanded to 10^4), and the partial solutions often suffer from local minima, resulting in a low success rate. The benefit of using tunnels is evident in the dramatic reduction of the effective branching factor and nodes expanded for Σ_4.

When using the complete global path from Σ_3 as a tunnel for Σ_4 (Figure 4.5(a) and Row 6 in Table 4.1), the effective branching factor reduces from 21.5 to 5.6, producing an exponential drop in node expansion and computation time, and enabling complete solutions to be generated in the space-time domain. This

planning task is able to successfully solve nearly 92% of the scenarios that were generated. However, since s_{start} and s_{goal} are far apart, the large depth of the search prevents this from being used at interactive rates for many agents.

By using successive waypoints in $\Pi(\Sigma_2)$ as s_{start} and s_{goal} to create a series of planning tasks in Σ_3 and Σ_4 (Figure 4.5(b) and Row 7 in Table 4.1), we reduce the breadth *and* depth of the search, allowing solutions to be returned at a fraction of the time (6 ms), without significantly affecting the success rate. The tradeoff is that independent plans are generated between waypoints along the global path, creating a two-level hierarchy between the domains.

Domain	BF	N	T	S	Q
$T(\Sigma_1)$	3.7	43	3	0.17	0.76
$T(\Sigma_2)$	4.6	85	8	0.23	0.57
$T(\Sigma_2,\Pi(\Sigma_1))$	2.1	17	5	0.32	0.65
$T(\Sigma_3)$	7.4	187	18	0.68	0.73
$T(\Sigma_4)$	21.5	10^4	2487	0.34	0.26
$T(\Sigma_4,\Pi(\Sigma_3,\Sigma_2,\Sigma_1))$	5.6	765	136	0.92	0.64
$\sum T_i(\Sigma_4,\Pi(\Sigma_3,\Sigma_2,\Sigma_1))$	5.4	75	8	0.86	0.58

TABLE 4.1: Comparative evaluation of the domains, and the use of multiple domains. **BF** = Effective branching factor. **N** = Average number of nodes expanded. **T** = Average time to compute plan (ms). **S** = Success rate of planner to produce collision-free trajectory. **Q** = Plan quality. Row 6,7 corresponds to the domain relationships illustrated in Figures 4.5(a) and (b) respectively.

Conclusion. The comparative evaluations of domains shows that no single domain can efficiently solve the challenging problem instances that were sampled. The use of tunnels significantly reduce the effective branching factor of the search in Σ_3 and Σ_4, while mapping successive waypoints in $\Pi(\Sigma_2)$ to multiple independent planning tasks reduce the depth of the search in Σ_3 and Σ_4, without significantly impacting success rate and quality. For the remaining results in this section, we adopt this domain relationship as it works well for our application of simulating multiple goal-directed agents in dynamic environments at interactive rates. Users may choose a different relationship based on their specific needs.

4.1.5 . Performance

We measure the performance of the framework by monitoring the execution time of each task type, with multiple instances of planning tasks for Σ_3 and Σ_4. We limit the maximum deliberation time $t_{max} = 10$ ms, which means that the total time executing any of the tasks at each frame cannot exceed $10ms$. For this experiment, we limit the total number of tasks that can be executed in a single frame to 2 (including T_0) to visualize the execution time of each task over different frames. Figure 4.6 illustrates the task execution times of a single agent over a 30 second simulation for the scenario shown in Figure 4.2(a). The execution task T_0 which is responsible for character animation and simple steering takes approximately $0.4-0.5$ ms of execution time every frame. Spikes in the execution time correlate to events in the world. For example, a local non-deterministic change in the environment (Frames 31,157) triggers a plan update in $T(\Sigma_3)$, which in turn triggers an update in $T(\Sigma_4)$. A global change such as a crowd blocking a passage or a change in goal (Frames 39, 237,281) triggers an update in $T(\Sigma_2)$ or $T(\Sigma_1)$ which in turn propagates events down the task dependency chain.

Note that there are often instances during the simulation when the start and goal changes significantly or when plans are invalidated, requiring planning from scratch. However, we ensure that our framework meets real-time constraints due to the following design decisions: (a) limiting the maximum amount of time to deliberate for the planning tasks, (b) intelligently distributing the available computational resources between tasks with highest priority, and (c) increasing the inflation factor to quickly produce a sub-optimal solution when a plan is invalidated, and refining the plan in successive frames.

Memory. $T(\Sigma_1)$ and $T(\Sigma_2)$ precomputes navigation meshes for the environment whose size depend on environment complexity, but are shared by all agents in the simulation. The runtime memory requirement of these tasks is negligible since it expands very few nodes. The memory footprint of $T(\Sigma_3)$ and $T(\Sigma_4)$ is defined by the number of nodes visited by the planning task during the course of a simulation. Since each planning task in Σ_3 and Σ_4 searches between successive waypoints in the global plan, the search horizon of the planners is never too large. On average, the number of visited nodes is 75 and 350 for $T(\Sigma_3)$ and $T(\Sigma_4)$ respectively with each node occupying $16-24$ bytes in memory. For 5

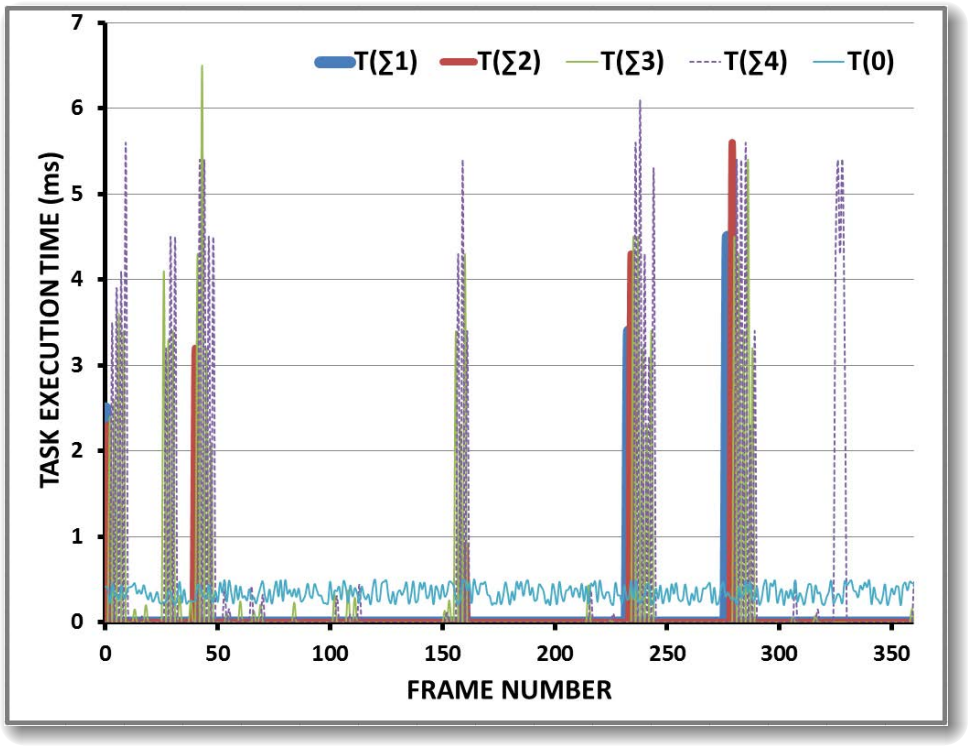


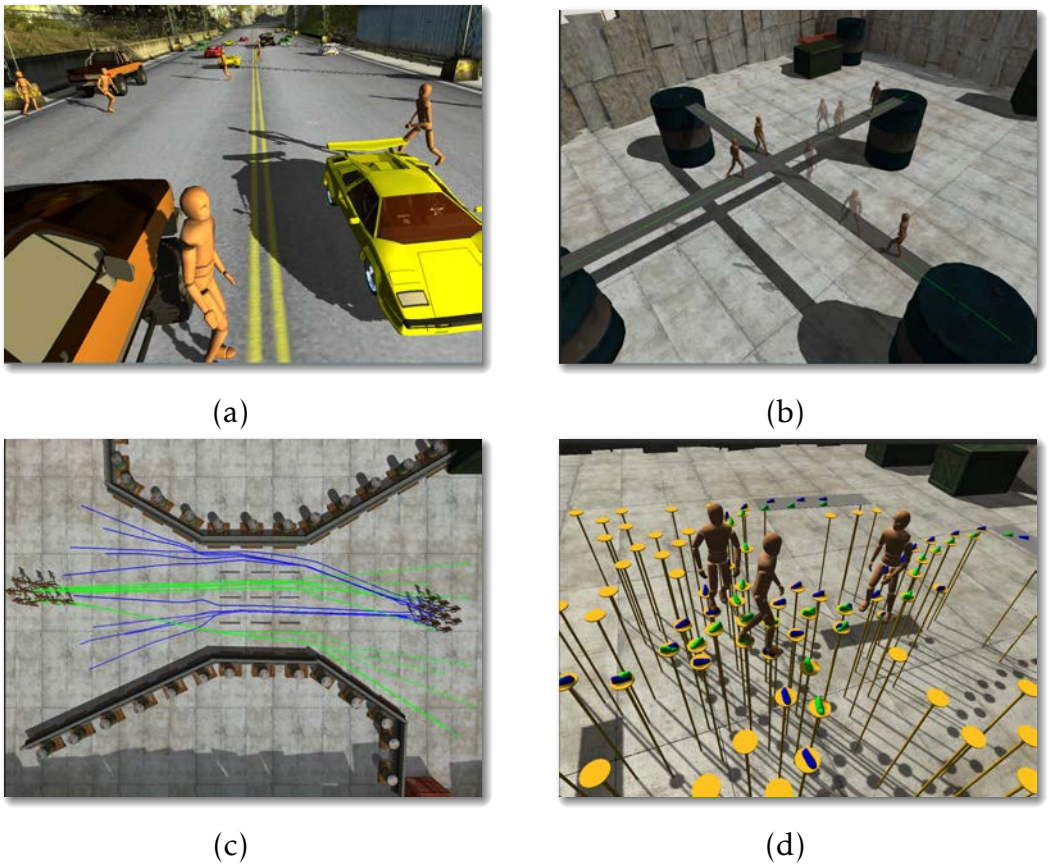
FIGURE 4.6: Task execution times of the different tasks in our framework over the course of a 60 second simulation.

running instances of $T(\Sigma_3)$ and $T(\Sigma_4)$, this amounts to approximately $45KB$ of memory per agent. Additional memory for storing other plan containers such as OPEN and CLOSED are not considered in this calculation as they store only node references and are cleared after every plan iteration.

Scalability. Our approach scales linearly with increase in number of agents. The maximum deliberation time *for all* agents can be chosen based on the desired frame rate which is then distributed among agents and their respective planning tasks at each frame. The cost of planning is amortized over several frames and all agents need not plan simultaneously. Once an agent computes an initial plan, it can execute the plan with efficient update operations until it is allocated more deliberation time. If its most imminent plan is invalidated, it is prioritized over other agents and remains stationary till computational resources are available. This ensures that the simulation meets the desired framerate.

4.1.5 . Scenarios

We demonstrate the benefits of our framework by solving many challenging scenarios (Figure 4.7) requiring space-time precision, explicit coordination between interacting agents, and the factoring of dynamic information (obstacles, moving platforms, user-triggered changes, and other agents) at all stages of the decision process. All results shown here were generated at 30 fps or higher, which includes rendering and character animation. We use an extended version of the ADAPT character animation system [Johansen, 2009] for the results.

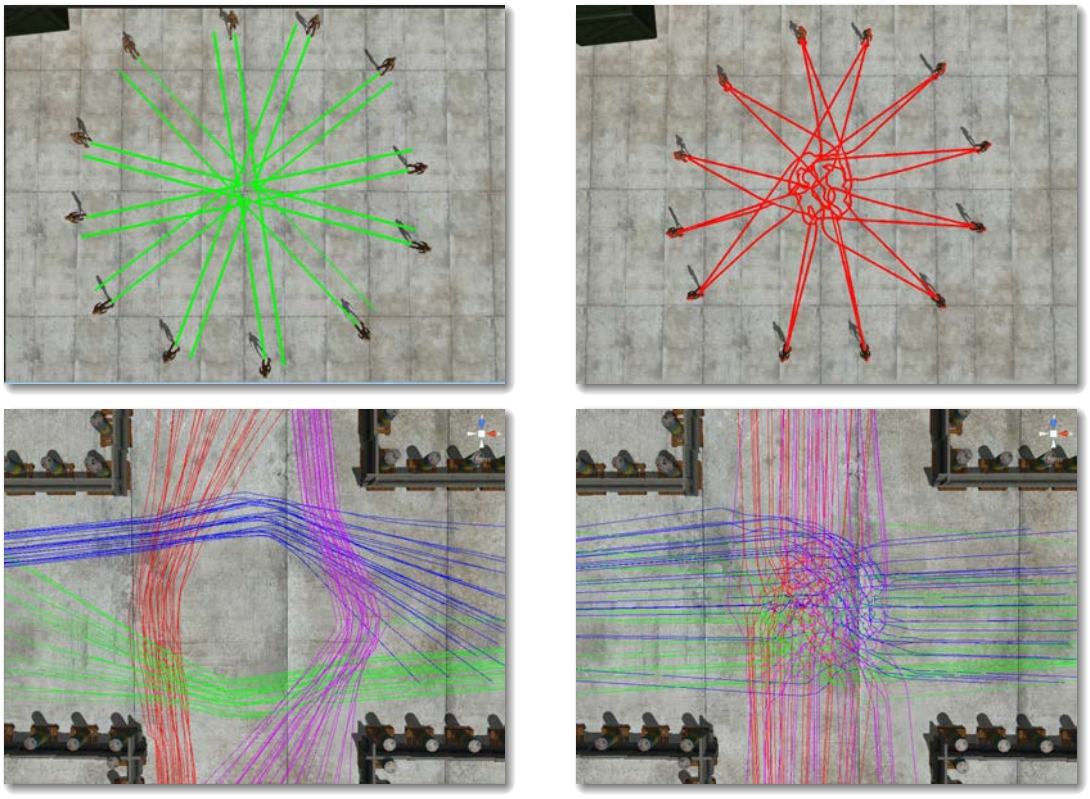


(a) (b)

(c) (d)

Figure 4.7: Different scenarios. (a) Agents crossing a highway with fast moving vehicles in both directions. (b) 4 agents solving a deadlock situation at a 4-way intersection. (c) 20 agents distributing themselves evenly in a narrow passage, to form lanes both in directions. (d) A complex environment requiring careful foot placement to obtain a solution.

Deadlocks. Multiple oncoming and crossing agents in narrow passageways cooperate with each other with space-time precision to prevent potential deadlocks. Agents observe the presence of dynamic entities at waypoints along their global path and refine their plan if they notice potentially blocked passageways or other high cost situations. Crowd simulators deadlock for these scenarios, while a space-time planner does not scale well for many agents.



Our framework | Unity navigation and steering

FIGURE 4.8: Trajectory comparison of our method with an off the shelf predictive steering algorithm in the Unity game engine. Our framework minimizes deviation and uses speed variations to avoid collisions in space-time.

Choke Points. This scenario shows our approach handling agents arriving at a common meeting point at the same time, producing collision-free straight trajectories. Figure 4.8 compares the trajectories produced using our method with an off the shelf navigation and predictive collision avoidance algorithm in the Unity game engine. Our framework produces considerably smoother trajectories and minimizes deviation by using subtle speed variations to avoid collisions in space-time.

Unpredictable Environment Change. Our method efficiently repairs solutions in the presence of unpredictable world events, such as the user-placement of obstacles or other agents, which may invalidate current paths.

Road Crossing. The road crossing scenario demonstrates 40 agents using space-time planning to avoid fast moving vehicles and other crossing agents.

Lane Selection for Bi-directional Traffic. This scenario requires agents to make a navigation decision in choosing one of 4 lanes created by the dividers. Agents distribute themselves among the lanes, while bi-directional traffic chooses different lanes to avoid deadlocks. This scenario requires non-deterministic dynamic information (other agents) to be accounted for while making global navigation decisions. This is different from emergent lane formation in crowd approaches, which bottlenecks at the lanes and cause deadlocks without a more robust navigation technique.

Four-way Crossing We simulate 100 oncoming and crossing agents in a four-way crossing. The initial global plans in Σ_1 take the minimum distance path through the center of the crossing. However, Σ_2 predicts a space-time collision between groups at the center and performs plan refinement so that agents deviate from their optimal trajectories to minimize group interactions. A predictive steering algorithm only accounts for imminent neighboring threats and is unable to avoid mingling with the other groups (second row of Figure 4.8).

Space-Time Goals. We demonstrate a complex scenario where 4 agents in focus (additional agents are also simulated) have a temporal goal constraint, defined as an interval $(40+/-1 second)$. Agents exhibit space-time precision while jumping across moving planes to reach their target and the temporal goal significantly impacts the decision making at all levels, where the space-time domain maybe unable to meet the temporal constraint and require plans to be modified in earlier domains. No other approach can solve this with real-time constraints.

Many of these scenarios *cannot* be solved by the current state of the art in multi-agent motion planning, which is able to either handle a single agent with great precision, or simulate many simple agents that exhibit reactive collision avoidance.

4.2 . Planning Using Footsteps

This work focuses on the computation of natural footsteps trajectories for groups of agents. There are some approaches that do focus on correct foot placement, but in most cases they are quite limited in the range of animations available or else can only deal with a small number of agents. Our work enforces foot placement constraints and uses motion capture data to produce natural animations, while still meeting real-time constraints for many interacting characters.

Figure 4.9 illustrates an example of four agents planning their footstep trajectory towards their goal while avoiding collision with other agents, and replanning when necessary. The resulting trajectories not only respect ground contact constraints, but also create more natural paths than traditional multi agent simulation methods.

This section is organized as follows. We first give an overview of our framework and then we explain in detail our pre-process step, planning algorithm and animation system. Finally we show some of our results and present a discussion about the strength of our method and its limitations along with conclusions and future work.

4.2.1 . Overview

Figure 4.10 illustrates the process of dynamic footstep planning for each character in real-time. The framework iterates over all characters in the simulation

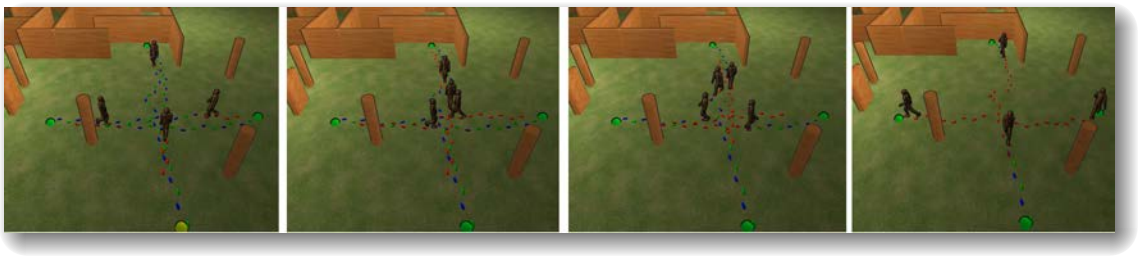


Figure 4.9: Footstep trajectories planning for four agents reaching goals in opposite directions

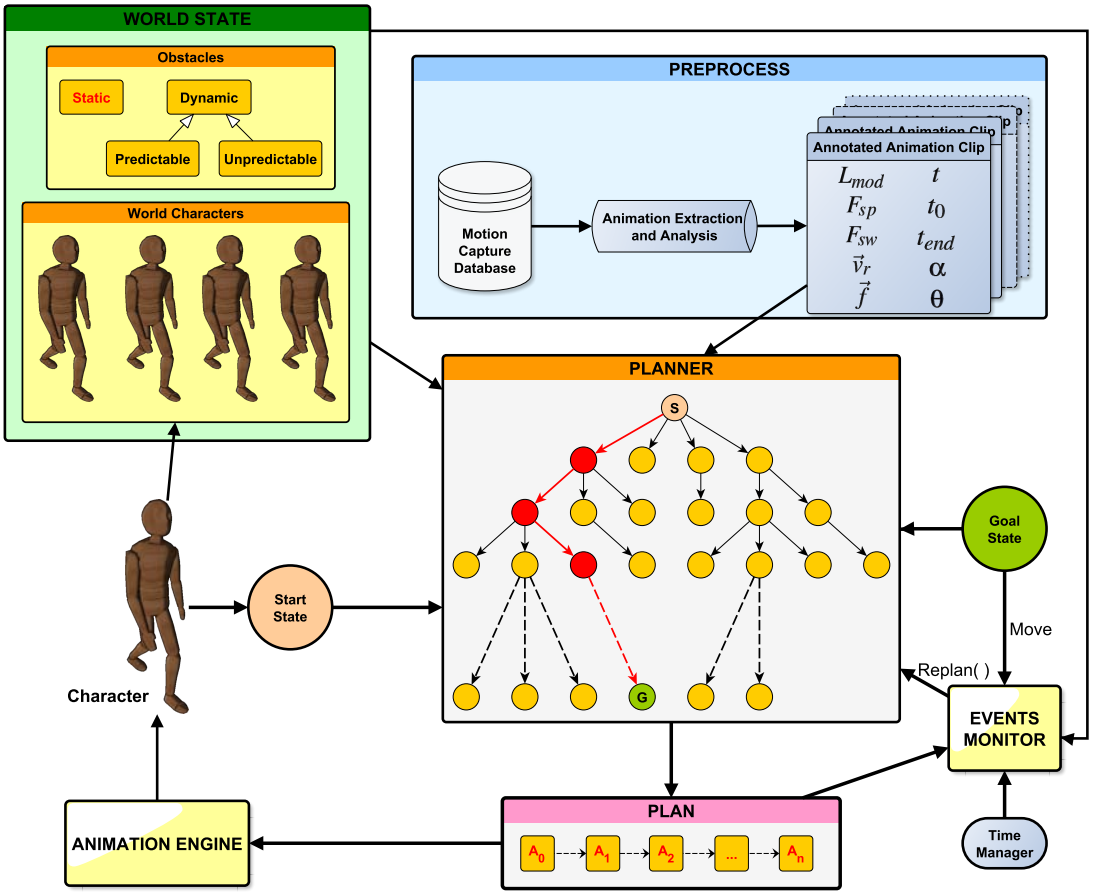


FIGURE 4.10: Diagram showing the process required for the dynamic footstep planning algorithm

to calculate each individual foot step trajectory considering obstacles in the environment as well as other agents' calculated trajectories.

The *Preprocess* phase is responsible for extracting annotated animation clips from a motion capture database. The real-time *Planner* uses the annotated animations as transitions between state nodes in order to perform a path planning task to go from an input *Start State* to a *Goal State*. The output of the planner is a *Plan* consisting of a sequence of actions $A_0, A_1, ..., A_n$, which are clips that the *Animation Engine* must play in order to move the *Character* along the computed path. Both state and plan of the *Character* are then input to the *World State* and thus exposed to other agents' planners, together with the nearby static or dynamic obstacles. The *World State* is used to prune and accelerate the search in order to predict and avoid potential collisions. The *Time Manager* is responsible for checking the elapsed time between frames to keep track of the expiration time of the current plan. Finally the *Events Monitor* is in charge of detecting events that will force the planner to recompute a new path. The *Events Monitor*

receives information from the *World State*, the *Time Manager*, *Goal State* and the character's current *Plan*. Events include: a possible invalid plan or the detection of a new dynamic obstacle or the goal position changing.

4.2.1 . Events Monitor

The events monitor is the module of the system in charge of deciding when a new path needs to be recomputed. Elements that will trigger an event are:

- *Goal state changed*: when the goal changes its position or a new goal is assigned for the current character.
- *New agent or deterministic dynamic obstacle nearby*: other agents or dynamic obstacles enter the surrounding area of our character. A new path needs to be calculated to take into account the potential collision.
- *Collision against non-deterministic obstacle*: sometimes an unpredictable dynamic obstacle could lead to a collision (for example: a dynamic obstacle moved by the user), so when the events monitor detects such situation it triggers an event in order to react to it.
- *Plan expiration*: a way to ensure that each agent is taking into account the latest plans of every other agent is to give every plan an expiration time and force re-planning if this is reached. A time manager helps monitoring this task, but instead of a time parameter this event can also be measured and launched by a maximum number of actions that we want to perform (play) before re-planning.

4.2.2 . Preprocess

During an offline stage, we analyze a set or a database of animation clips in order to extract the actions that our planner will then use as transitions between states. Each action consists of a sequence of skeleton configurations that perform a single animation step at a time, i.e., starting with one foot on the floor, until the other foot (swing foot) is completely resting on the floor. Our preprocess should work with any animation clip, since we tried both handmade and motion capture clips (from the CMU database [University, 2013]). After

analyzing each animation clip, we calculate mirrored animations. Mirroring animations is done in order to have each analyzed animation clip with either feet starting on the floor. The output of this stage is a set of annotated animations that can be used by the planner and the animation engine. This set can be easily serialized and stored to be reused for all instances of the same character type (same skeleton and the same scale, otherwise even if they share animations these could produce displacements of different magnitudes), reducing both preprocess time and the global memory consumption.

4.2.2 . Locomotion Modes

In order to give our characters a wider variety and agility of movements we define different locomotion modes that need to be treated differently. Each animation clip will be tagged with its locomotion mode. We thus have the following set of locomotion modes:

- Walking: these are the main actions that will be used by the planner and the agents since they represent the most common way to move. We therefore have a wide variety of walks going from very slow to fast and in different angles (not just forward and backwards).
- Running: these are going to be treated in the same way as the walking actions with an additional cost penalty (since running consumes more energy than walking). We have also noticed empirically that for running actions it is not necessary to have as many different displacement angles as for walking actions.
- Turns: turns are going to be clips of animation where the agent turns in place or with a very small root displacement. They are going to be defined by their turning angle and velocity.
- Platform Actions: in this group we will find actions like jumping or crouching in order to avoid some obstacles. Such actions should have a high energy cost and should only be used in case of an imminent danger of collision.

While turns and platform actions need to be performed completely from start to end, and they do not have any intrinsic pattern we can easily detect, walking

and running animations can be segmented by clips containing a single step. So animations of both walking and running locomotion modes will have a special treatment as we will need to extract the footsteps and keep only the frames of the animation covering a single step.

4.2.2 . Footsteps Extraction

As previously mentioned in this section, an action starts with one foot on the floor and ends when the other foot is planted on the floor. But animation clips, especially motion capture animations, do not always start and end in this very specific way. Therefore we need a foot plant extraction process to determine the beginning and end ending of each animation clip that will be used as an action.

Simply checking for the height of the feet in the motion capture data is not enough, since it usually contains noise and artifacts due to targeting. In most cases, when swinging the foot forwards while walking, the foot can come very close to the ground, or even traverse it.

Other techniques also incorporate the velocity of the foot during foot plant, which should be small. However this solution can also fail, since foot skating can introduce a large velocity. We detect foot plants using a height and velocity based detector similar to the method described in [van Basten and Egges, 2009], where foot plant detection is based on both height and time. First, the height-based test provides a set of foot plants, but only those where the foot plant occurs in a group of adjacent frames, are kept.

Our method combines this idea with changes on velocity for more accurate results, so we detect a foot plant when for a discretisized set of frames the foot is close to the ground for a few adjacent frames and with a change in velocity (deceleration, followed by being still for a few frames, and finishing with an acceleration). Notice that this method works for any kind of locomotion ranging from slow walking to running including turns in any direction.

4.2.2 . Clip Annotation

An analysis is performed by computing some variables over the whole duration of the animation. Each analyzed animation clip is annotated with the following

information:

L_{mod}	Locomotion mode
F_{sp}	Supporting Foot
F_{sw}	Swing Foot
$\vec{v_r}$	Root velocity vector
$\vec{f}$	Foot displacement
t	Time duration
t_0	Initial time
t_{end}	End time
α	Movement angle
θ	Rotation angle
$\mathbb{P}$	Set of Sampled positions

TABLE 4.2: Information stored in each annotated animation clip.

Locomotion mode, indicates the type of animation (walk short step, walk long step, run, walk jump, climb, turn, etc). *Supporting foot* is the foot that is initially in contact with the floor, and the *swing foot* corresponds to the foot that is moving in the air towards the next footstep. The *supporting foot* is calculated automatically based on its height and velocity vector from frame to frame.

The *root velocity vector* indicates, taking the starting frame of the extracted clip as reference, the total local displacement vector of the root during the whole step. We therefore know the magnitude, the speed in *m/s* and the angle of its movement. Similarly, *foot displacement* tracks the movement of the swing foot.

Movement angle in degrees indicates the angle between the swing foot displacement vector and the initial root orientation. Therefore an angle equal to 0 means an action moving forward and 180 means it is a backward action. An Angle equal to 90 means an action moving to the left if the swing foot is the left one, or the right if the swing foot is the right one. Finally the *rotation angle* is the angle between the root orientation vector in the first and last frame of the clip.

t indicates the total time duration of the extracted clip, with t_0 and t_{end} storing the start and end point of the original animation that the extracted clip covers. These values will be used by the animation engine to play the extracted clip.

$\mathbb{P}$ corresponds to a set of sampled positions for certain joints of the character within an animation clip, and it is used for collision detection (see section 4.2.3.5)

4.2.3 . Planning Footstep Trajectories

In this section, we first present the high level path planning on the navigation mesh. Then we define the problem domain we are dealing with when planning footsteps trajectories. Next we give details of the real-time search algorithm that we use as well as the pruning carried out to accelerate the search. Finally we explain how the collision detection and prediction is performed.

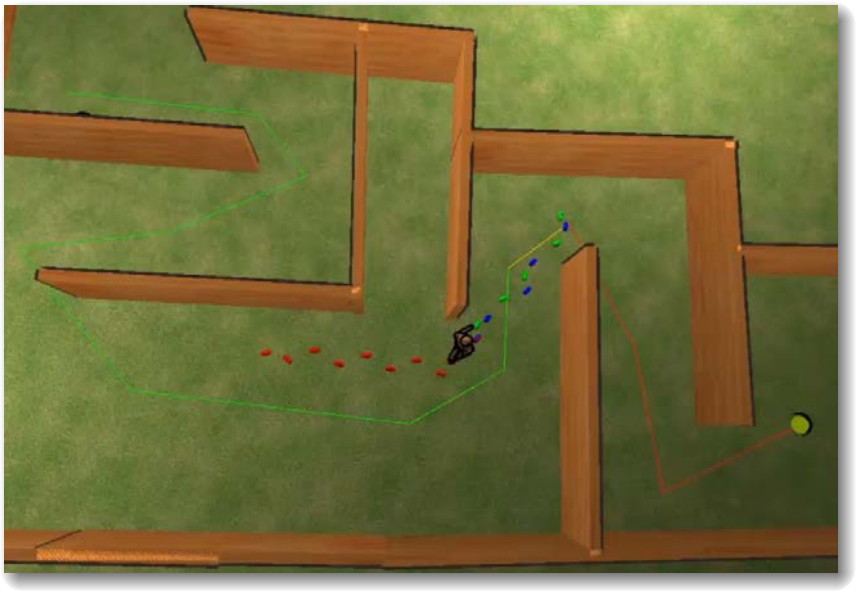


Figure 4.11: High level path with local footstep trajectory between consecutive visible waypoints.

4.2.3 . High Level Path Planning

Footstep trajectories are calculated between waypoints of the high level path (see Figure 4.11). This path is calculated over the navigation mesh using Recast [Mononen, 2009]. An A* algorithm is used to compute the high level path, and then footstep trajectories are calculated between consecutive visible waypoints. So given a sequence of waypoints $\{w_i, w_{i+1}, w_{i+2}, ..., w_{i+n}\}$), if there is a collision-free straight line between w_i and w_{i+n}, then the footstep trajectory is calculated between those two waypoints, and any other intermediate point is ignored. This

provides more natural trajectories as it avoids zig-zagging over unnecessary waypoints. Waypoints are considered by the planner as goal states, and each time that we change a waypoint the change of goal is detected by the events monitor, thus forcing a new path to be computed.

4.2.3 . Problem Definition

The algorithm for planning footstep trajectories needs to calculate the sequence of actions that each agent needs to follow in order to go from their start position to their goal position. This means solving the problem of moving in a footstep domain between two given positions in a specific amount of time. Therefore, characters calculate the best trajectory based on their current state, the cost of moving to their destination and a given heuristic. The cost associated with each action is given by the bio-mechanical effort required to move (i.e: walking has a smaller cost than running, stopping for a few seconds may have a lower cost than wandering around a moving obstacle). The problem domain that we are dealing with is thus defined as:

$$\Omega = \left(\mathbb{S}, \mathbb{A}, c(s, s'), h(s, s_{goal})\right)$$

Where $\mathbb{S}$ is the state space and is defined as the set of states composed of the character's own state *self*, the world composition *environment*, and the *other agents* state. The action space $\mathbb{A}$ indicates the set of possible transitions in the state space and thus will have an impact on the branching factor of the planner. Each transition is an action, so we will have as many transitions as extracted clips times the possible speed variations we allow to introduce (we can for example reproduce a clip at half speed to obtain its displacement two times slower). Actions are then going to be defined by their corresponding annotated animation. $c(s, s')$ is the cost associated with moving from state s to state s'. Finally $h(s, s_{goal})$ is the heuristic function estimating the cost to go from s to s_{goal}.

4.2.3 . Real-Time Planning Algorithm

Planning footsteps trajectories in real time requires finding a solution in the problem domain Ω described earlier. The planner solution consists of a sequence $A_0, A_1, ..., A_n$ of actions. Our planner interleaves planning with execution, because we want to be able to replan while consuming (playing) the action. For this purpose, we use a best-first search technique (e.g., A*) in the footstep problem domain, defined as follows:

- $\mathbb{S}$: the state space will be composed of the character's own state (defined by position, velocity, and the collision model chosen), the state of the other agents plus their plan, and the state and trajectory of the deterministic dynamic obstacles. For more details about collision models and obstacles avoidance see the following section **Collision Prediction**.
- $\mathbb{A}$: the action space will consist of every possible action that can be concatenated with the current one without leading to a collision, so before adding an action we will perform all necessary collision checks.
- $c(s, s')$: the cost of going from one state to another will be given by the energy effort necessary to perform the animation:

$$c(s, s') = M \int_{t=0}^{t=T} e_s + e_w |v|^2 \, dt$$

  where M is the agent mass, T is the total time of the animation or action being calculated, v the speed of the agent in the animation, and e_s and e_w are per agent constants (for an average human, $e_s = 2.23 \frac{J}{Kg.s}$ and $e_w = 1.26 \frac{J.s}{Kg.m^2}$) [Kapadia et al., 2011a].

- $h(s, s_{goal})$: the heuristic to reach the goal comes from the optimal effort formulation:

$$h(s, s_{goal}) = 2M c^{opt}(s, s_{goal}) \sqrt{e_s e_w}$$

  where $c^{opt}(s, s_{goal})$ is the cost of the optimal path to go from s to s_{goal}, in our case we chose the euclidian distance between s and s_{goal}[Kapadia et al., 2011a]. The optimal effort for an agent in a scenario is defined as the energy consumed in taking the optimal route to the target while traveling at the average walking speed: $v_{av} = \sqrt{\frac{e_s}{e_w}} = 1.33 m/s$

Taking all these components into consideration the planner can search for the path with least cost and output the footstep position with their time marks that the animation engine will follow by playing the sequence of actions planned (see figure 4.12).

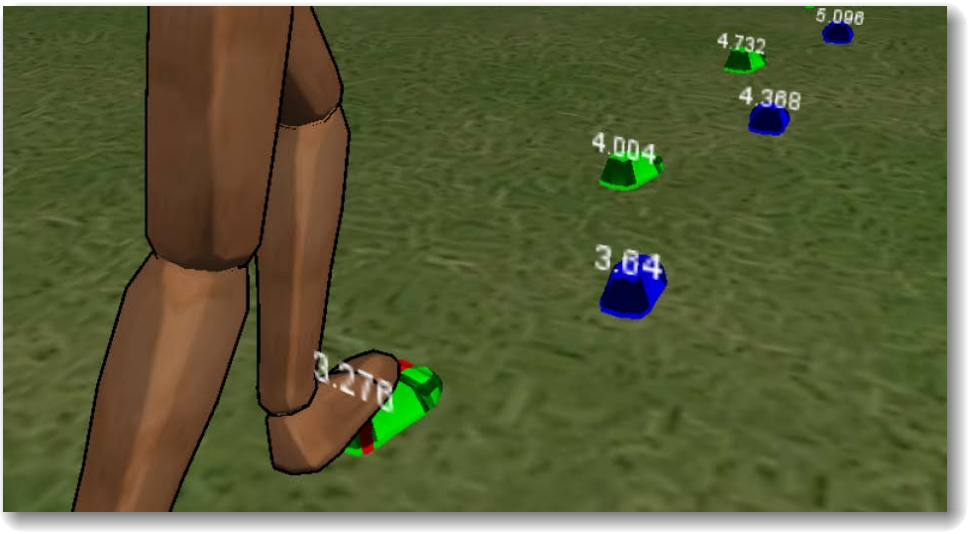


FIGURE 4.12: Footsteps trajectory with time constraints that need to be followed by the animation controller.

4.2.3 . PRUNING RULES

In order to accelerate the search we can add simple rules to help prune the tree and reduce the branching factor. A straight forward way to halve the size of the tree consists of considering only consecutive actions starting with the opposite foot. So given a current node with a supporting foot, expand the node only for transitions that have that same foot as the swing foot. Actions which are not possible due to locomotion constraints on speed or rate of turning are also pruned to ensure natural character motion (so after a staying still animation, we will not allow a fast running animation). The next pruning applied is based on collision prediction as we will see in the following section. The idea is that when a node is expanded and a collision is detected, the whole graph that could be expanded from it gets automatically pruned. The pruning process reduces the branching factor of the search, and also ensures natural footstep selection

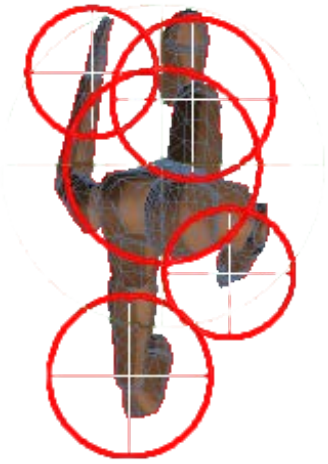


Figure 4.13: Collision model of 5 cylinders around the head, the left and right hands, and the left and right foot.

4.2.3 . Collision Prediction

While expanding nodes the planning algorithm must check for each expanded node whether the future state is collision free or not. If it is collision free, then it maintains that node and continues expanding it. Otherwise, it will be discarded. In order to have large simulations in complex environments we need to perform this pruning process in a very fast manner.

In order to predict collisions against other agents or obstacles (both dynamic or static), we introduce a multi-resolution collision detection scheme which performs collision checks for two resolution levels. Our lowest resolution collision detection model is a simple cylinder centered at the root of the agent with a fixed radius. The higher resolution model consists of five cylinders around the end joints (head, hands and feet) that are used to make finer collision tests Figure 4.13.

We could introduce more collision models, where high resolution ones will be executed only in case of detecting collisions using the coarser ones. At the highest complexity mode we could have the full mesh collision check, but for the purpose of our simulation the 5 cylinders model gives us enough precision to avoid agents walking with their arms intersecting against other agents as they swing back and forth. Compared against simpler approaches that only consider obstacle detection against a cylinder, our method gives better results since it allows us to have closer interactions between agents. All obstacles have

simple colliders (boxes, spheres, capsules) to accelerate the collision checks by using a fast physics ray casting test.

It is also important to mention that collision tests are not only performed using the initial and end positions of the expanded node, but also with sub-sampled positions inside the animation (for the 5 cylinder positions). For example, an agent facing a thin wall as a start position and the other side of the wall as end position of its current walk forward step. If we only check for possible collisions with those start and end positions we would not detect that the agent is actually going through the wall.

The sub-sample for each animation is performed off-line and stored in the annotated animation. To save memory, this sampling is performed at low frequencies and then in real time intermediate positions can be estimated by linear interpolation.

Finally, we provide the characters with a surrounding view area to maintain a list of obstacles and agents that are potential threats to our path (see figure 4.14). For each agent, we are only interested in those obstacles/agents that fall within the view area in order to avoid running unnecessary collision tests.

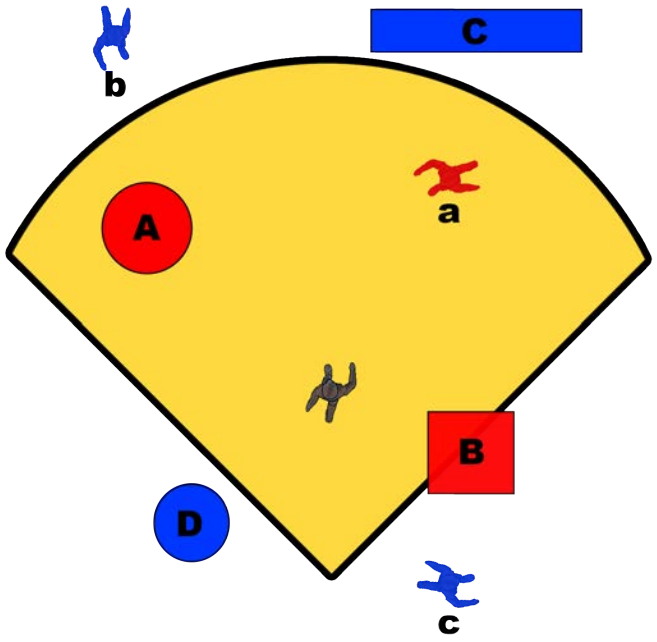


FIGURE 4.14: When planning we only consider obstacles and agents that are inside the view area. Obstacles A, B and agent a are inside it and the agent will try to avoid them, while it will ignore obstacles C, D and agents b and c .

Static World

Static obstacles are part of the same static world that is used to compute the navigation mesh with Recast [Mononen, 2009]. They do not need to have a special treatment since the high-level path produces waypoints that avoid collisions with static obstacles..

Deterministic Dynamic Obstacles and Other Agents

Deterministic obstacles move with a predefined trajectory. Other agents have precomputed paths which can be queried to predict their future state. To avoid interfering with those paths we allow access to their temporal trajectories. So, for each expanded node with state time t we check for collisions with every obstacle and agent that falls inside his view area at their trajectory positions at time t. Figure 4.15 shows an example of an agent avoiding two dynamic obstacles.

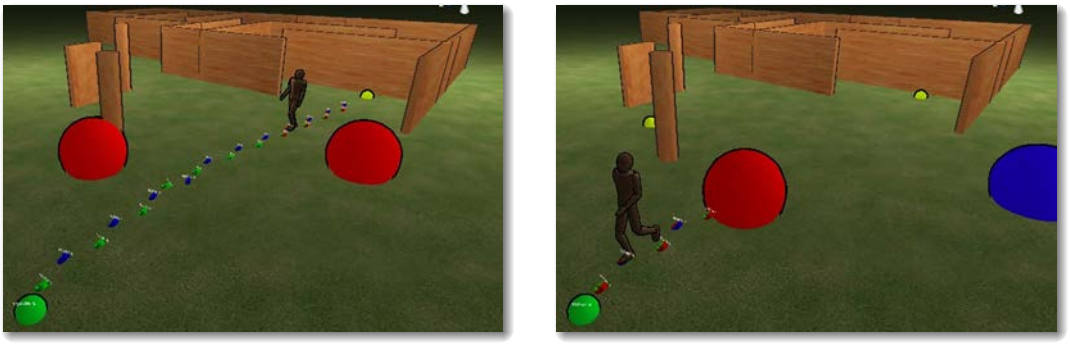


Figure 4.15: An agent planning with two dynamic obstacles in front of him (top). After executing some steps the path is re-planned. The blue obstacle indicates that it is not in his nearby area anymore, so that obstacle is not considered in the collision check of this new plan. (bottom)

Unpredictable Dynamic Obstacles

Unlike deterministic dynamic obstacles and other agents, unpredictable dynamic obstacles are impossible to be accounted for while planning. Therefore they can be ignored when expanding nodes, but we need a fast way to react to them. This is the reason why we need the events monitor to detect immediate collisions and force re-planning. Figure 4.16 shows an example where a wall is arbitrarily moved by the user and the agent needs to continuously re-plan its trajectory.

FIGURE 4.16: An agent reacting to a non-deterministic obstacle by re-planning his path.

4.2.4 . ANIMATION ENGINE

The animation engine is in charge of playing the output sequence of actions given by the planner. These actions contain all the data in the annotated animation. When a new action is played it sets t_0 as the initial time of the animation. When the current animation reaches t_{end} the animation engine blends the current animation with the next one in the queue.

The Animation Engine also tracks the global root position and orientation, and applied rotation corrections by rotating the whole character using the rotation values of the annotated animation (rotation angle θ). The blending time between actions can be user defined within a short time (for example 0.5s).

4.2.5 . RESULTS

The presented framework has been implemented using the ADAPT simulation platform [Shoulson et al., 2013] which works with Unity Game Engine [Unity, 2014] and C# scripts. Our current framework can simulate around 20 agents at approximately 59-164 frames per second (depends on the maximum planning time allowed), and 40 agents at 22-61 frames per second (Intel Core i7-2600k CPU @ 3.40GHz and 16GB RAM). Figure 4.17 shows the frame rates achieved on average for an increasing number of agents. The black line corresponds to a maximum planning time of 0.01s, and the red line corresponds to 0.05s. Additionally, by setting planner parameters such as the horizon of the search, we can achieve significant speedup at the expense of solution fidelity. For example,

we can produce purely reactive simulations where the character only plans one footstep ahead by reducing the search horizon to 1.

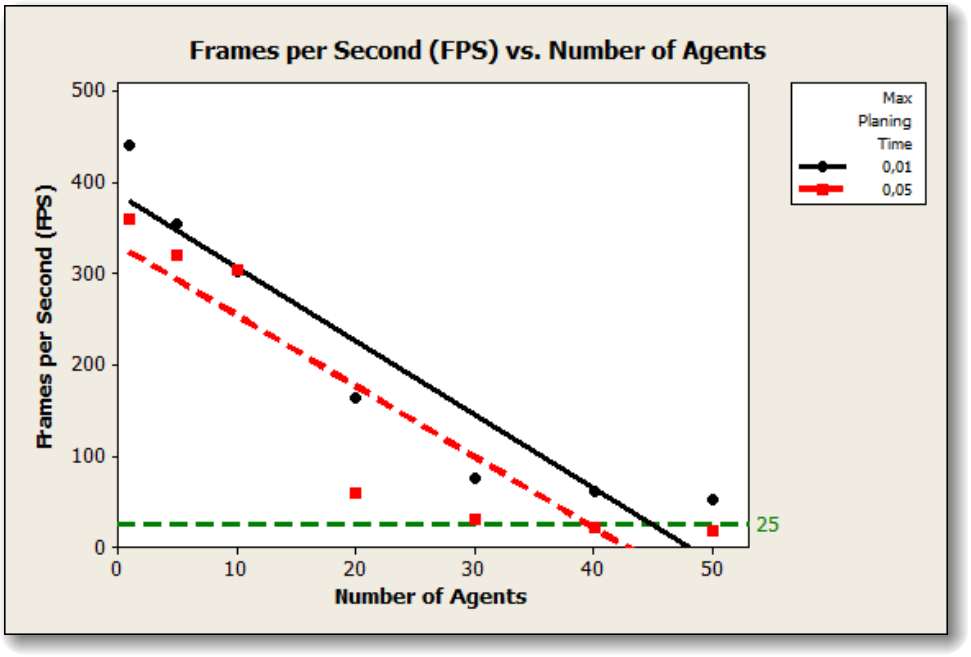


FIGURE 4.17: This graph shows the frames per second on average for different simulations with increasing number of agents. We have used two values for the maximum planning time: 0.01 resulting in higher frame rates, and 0.05 resulting in lower frame rates but better quality paths

The results showed have been made with a database of 28 motion captured animations. This is a small number compared to approaches based on motion graphs (generally having around 400 animation clips), but a large number compared with techniques based on handmade animation (such as pre-computed search trees). This decision allows us to achieve results that look natural and yet can be used for real time applications.

Our approach solves different scenarios where several agents are simulated in real-time achieving natural looking paths while avoiding other obstacles and characters. The quality of the results in terms of natural paths and collision avoidance depends on the planner. The planner will be given a specific amount of time to find a solution (which translates in how many nodes of the graph are expanded). Obviously when we allow larger search times (larger number of nodes to expand) the resulting trajectory looks more natural and is collision free, but at the expense of being more computationally expensive. Alternatively, if we drastically reduce the search time (smaller number of nodes to expand) we may end up having collisions as we can see in Figure 4.18.

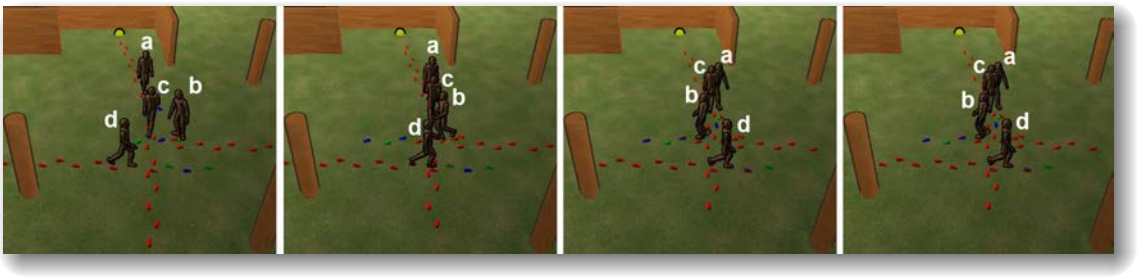


FIGURE 4.18: Example with four agents crossing paths with a drastically reduced search time resulting in agents **a** and **c** not being able to avoid intersection as seen in the last two images of this secuence. Also notice how agent **b**, walks straight towards **c**, steps back and then continues, instead of following a smooth curve around **c**.

Interleaving planning with execution provides smooth animations, since not all the characters plan their paths simulateneously. At any time, the new plan is calculated with the start position being the end position of the current action.

We have also shown how the Events Monitor can successfully plan routes when deterministic obstacle invalidate a character's plan, as well as efficiently react to non deterministic obstacles (see Figures 4.15 and 4.16)

4.3 . Conclusions on Crowd Simulation

Our first contribution is a framework for real-time, multi-agent navigation in large-scale, complex, dynamic environments, with space-time precision. We use multiple problem domains to provide a balance between control fidelity and computational complexity by accounting for dynamic aspects of the environment at all stages of decision-making. The original navigation problem is decomposed into a set of smaller problems that are distributed across planning tasks working in these different domains. An anytime dynamic planner is used to efficiently compute and repair plans for each of these tasks, and the use of tunnel based search is particularly useful for working in complex domains such as Σ_4 where the plan from Σ_3 is used to focus its search, thereby greatly reducing the number of nodes expanded.

The domains described in this work represent popular solutions that are used in both academia and industry. Navigation meshes (Σ_1) are a standard solution [Mononen, 2009] for representing free spaces in arbitrarily large, complex, static environments with recent proposed extensions [van Toll et al., 2012] that account for dynamic information (Σ_2). A grid-based representation (Σ_3) provides a uniform discretization of the environment, and is widely used in robot motion planning [Koenig and Likhachev, 2002, Likhachev et al., 2005]. The introduction of time as a third dimension (Σ_4) enables collision checks in the future, facilitating more robust collision resolution.

These domains provide a nice balance between global navigation and space-time planning, enabling us to showcase the strength of our framework: the ability to use multiple domains of control, and leverage solutions across domains to accelerate computations while still providing a high degree of control fidelity. Additional domains can be easily integrated (e.g., the footstep domain we have presented in our second contribution) to meet application-specific needs, or solve more challenging motion planning problems.

Domains can be connected by using the plan from one domain as a tunnel for the other, or by using successive waypoints along the plan as start and goal pair for multiple planning tasks in a more complex domain. We evaluated both domain relationships based on computational efficiency and coverage, as shown

in Table 4.1. Using waypoints from the navigation mesh domain as start, goal pairs for planning tasks in the grid and space-time domain keeps the search depth for Σ_3 and Σ_4 within reasonable bounds. The tradeoff is that a space-time plan is never generated at a global level from an agent's start position to its target, thus sacrificing completeness guarantees. This design choice worked well for our experiments where the reduction in success rate of our framework when using this scheme was within reasonable bounds, while providing a considerable performance boost, making it suitable for practical game-like applications. Users may wish to opt for different domain relationships depending on the application.

Our second contribution is a multi-agent simulation approach where planning is done in the action space of available animations. Animation clips are analyzed and actions are extracted and annotated, in order to be used in real time to expand a search tree. Nodes are only expanded if they are collision free. To predict collisions we sample animations and use a new collision model with colliders for each end joint (head, hands and feet). This way we are able to simulate agents avoiding more detailed collisions. The presented framework handles both deterministic and non-deterministic obstacles, since the former can be taken into consideration when planning, while the later needs a completely reactive behavior.

Unlike pre-computed search trees our set of transitions is composed of actions, and mainly footsteps, which allows us to build online the search tree and to dynamically prune it, considering not only start and goal positions, but also departure and arrival times. An events monitor can help us to decide when to re-plan the path, based on the environment situation such as obstacle proximity or velocity.

As Illustrated in 4.17, the computational complexity of our framework scales linearly with the number of agents. By reducing the search depth and maximum planning time, we can simulate a larger crowd of characters at interactive rates. Notice that memory is required per animation (to store sub-sampled animations) and not per agent in the simulation, therefore increasing the size of the simulated group of agents would not have an impact on the memory requirements of our system. If we wanted to simulate crowds of characters we would need more CPU power, but not memory as long as we had more instances of characters sharing the same skeleton and animations.

Publications

Our work on simulation has yielded the following two publications ([Beacco et al., 2013b, Kapadia et al., 2013]):

- M. Kapadia, A. Beacco, F. Garcia, V. Reddy, N. Pelechano and N.I. Badler. *Multi-Domain Real-time Planning in Dynamic Environments*. ACM SIGGRAPH/EUROGRAPHICS Symposium on Computer Animation 2013 (SCA 2013), Anaheim, CA, U.S.A., 2013
- A. Beacco, N. Pelechano and M. Kapadia. *Dynamic Footsteps Planning for Multiple Characters*. EUROGRAPHICS Spanish Conference of Computer Graphics 2013 (EGse CEIG 2013), Madrid, Spain., 2013

Both publications were done in collaboration with the *Human Modeling and Simulation Lab*, at the *University of Pennsylvania*, in Philadelphia, where I did a 4 months stay as part of my PhD. Special thanks to Professor Norman I. Badler, and to Doctor Mubbasir Kapadia, who oriented this work during that period and afterward.
Furthermore, we plan for a journal submission extending our work on multi-domain planning and including the footstep domain of our second contribution.

5 . Contributions to Crowd Animation

Synthesizing accurate human motion whilst keeping within constraints is not an easy task, and although many techniques have been developed for synthesizing the motion of one agent, they cannot be easily extended to large numbers of agents simulated in real time. In this chapter we first introduce a method to synthesize animations using only the root motion as input, while avoiding artifacts such as the foot-sliding effect. Then we present a method that allows to synthesize animations to accurately follow a path composed of footsteps, just like the one obtained by our footsteps planner in our previous contribution.

5.1 . Reflecting the Root Motion

In this section we present an Animation Planning Mediator (APM); a highly efficient animation synthesizer that can be seamlessly integrated with a crowd simulation system. The APM selects the parameters to feed a motion synthesizer while it feeds back to the crowd simulation module the required updates to guarantee consistency. Even when we only have a small set of animation clips, our technique allows a large and continuous variety of movement. It can be used with any crowd simulation software, since it is the crowd simulation module which drives the movement of the virtual agents and our module limits its work to adjusting the root displacement and skeletal state.

5.1.1 . Framework

The framework employed for this work performs a feedback loop where the APM acts as the communication channel between a crowd simulation module and a character animation and rendering module. The outline of this framework is shown in Figure 5.1.

For each frame, the crowd simulation module, CS, calculates the position p, velocity v, and desired orientation of the torso Θ. This information is then passed to the APM in order to select the parameters to be sent to the character animation module, CA, which will provide the next pose of the character, P. Each pose is a vector specifying all joint positions in a kinematic skeleton.

The APM calculates the next synthesized animation S_i which is described by the tuple A_i , dt , b, P , v, Θ (see Table 5.1).

The APM may need to slightly adjust the position and velocity direction of the agent in order to guarantee that the animations rendered are smooth and continuous. It is thus essential that the crowd simulation model employed works in continuous space and allows for updates of the position and velocity of each agent at any given time in order to guarantee consistency with the requirements dictated by the animation module. We have used the crowd simulation (CS) model HiDAC [Pelechano et al., 2007].

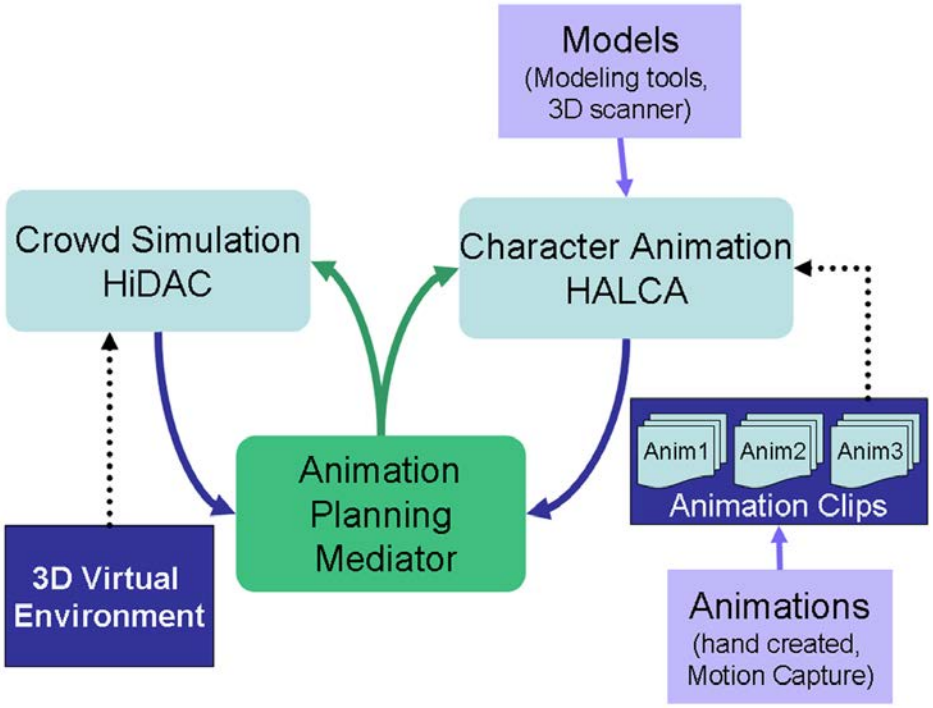


FIGURE 5.1: Framework

P	Agent root position.
$\mathbf{V}$	Agent velocity.
Θ	Angle indicating torso orientation in x-z plane.
A_i	Animation clip for agent i.
dt	Differential of time for blending animation clip A_i
B	Blending factor when changing animation clip, i.e. when $A_i(t) \neq A_i(t\text{-}1)$, where t indicates time.
$\mathcal{P}$	Pose given by the skeletal state of the agent.

TABLE 5.1: Input/Output variables of the APM

The torso orientation at time t, w_t, is obtained from the velocity vector v_t after applying a filter so that it will not be unnaturally affected by abrupt changes.

$$w_t = \omega_o w_{t-1} + v_t \tag{5.1}$$

where ω_o is the orientation weight introduced by the user and ω_{t-1} is the direction of the orientation vector at time $t-1$. The orientation angle Θ of the vector w is measured relative to the positive x-axis (since either the angle or the vector can be calculated from the other).

This orientation filter is applied as we want the position of the character to be able to react quickly to changes in the environment such as moving agents and obstacles, but we need the torso to exhibit only smooth changes, as without this the result will be unnatural animations where the rendered characters appear to twist constantly. Through filtering we can simulate an agent that moves with a slight zigzag effect, while the torso of the rendered character moves in a constant direction.

For the animation and visualization of avatars (CA) we are using a hardware accelerated library for character animation (HALCA [Spanlang, 2009]). The core consists of a motion mixer and an avatar visualization engine. Direct access to properties such as the duration, frame rate, and the skeletal states of an animation are provided to the hosting application. Such information can be useful to compute, for example, the actual walking speed of a character when animated. Among the functionalities provided are: blending, morphing, and efficient access and manipulation of the whole skeletal state.

The CA contains a motion synthesizer which can provide a large variety of continuous motion from a small set of animations by allowing direct manipulation of the skeleton configuration. To create the library of animations, we decided to use hand created animations, although motion capture data could also be used after some pre-processing.

5.1.2 . Animation Planning Mediator

To achieve realistic animation for large crowds from a small set of animation clips, we need to synthesize new motions.

The APM is used to find the best animation available while satisfying a set of constraints. On the one hand it needs to guarantee that the next pose of the animation will reflect as closely as possible the parameters given by the crowd simulation module (p, v, Θ), and on the other hand, it needs to guarantee smooth and continuous animations. Therefore, the selection of the best next pose of the character needs to take into account the current skeletal state, the available animations, the maximum rotation physically possible for the upper body of a human, and whether there are any contact points to respect between the limbs of the skeleton and the environment (such as contact between a foot and the floor). Once the APM determines the best set of parameters for the next pose and passes this information to the CA for animation and rendering, it will also provide feedback to the CS in the cases where the parameters sent needed to be slightly adjusted to guarantee natural looking animations with the available set of animation clips and transitions.

During pre-processing the APM will calculate, for each animation clip average velocity v_{anim} in m/s by computing the total distance traveled by the character through the animation clip divided by the total duration of the animation clip, T, as well as the angle α between the torso orientation, Θ_{anim}, and the velocity vector, v_{anim}, in the animation clip. The i^{th} animation clip A_i is defined by the tuple $\{v_{anim}, i, \alpha_i, T_i\}$.

During the simulation the APM takes the input parameters from the CS and proceeds through the five steps shown in Figure 5.2 to obtain the output tuple $\{A_i, dt, b, P', p', \Theta\}$ that will be sent to the CA. We explain this in detail next.

5.1.2 . Animation Clip Selection

Instead of achieving different walking speeds by using hundreds of different walk animations, the algorithm can be effective with a limited number of animation clips by blending within an animation. This is given by the parameter

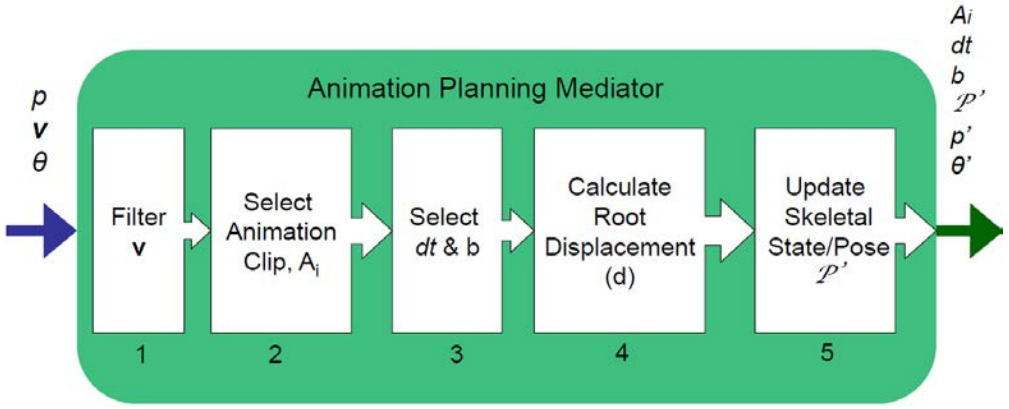


FIGURE 5.2: The Animation Planning Mediator steps

dt which defines the time elapsed between two consecutive frames. Each animation clip covers a subset of speeds going from the minimum speed of 0 m/s (dt=0) to the original speed of the animation clip (dt=1).

An animation clip of walking forward can also be used to have the agent turn, as we can reorient the foot on the floor, thus reorienting the entire figure. This provides natural looking results for high walking velocities, but for slower velocities, using several turning animation clips and blending between them results in more natural looking motion.

If we consider α being the angle between the direction of movement v_{anim} and the torso orientation Θ_{anim}, of an animation A_i, we can classify animations based on α and the velocity. For example, for an animation of walking sideways $\alpha = 90$ degrees and for walking forwards $\alpha = 0$ degrees.

To determine when to use each animation we classify them in a circle defined by tracks and sectors. A track is the area contained between two concentric circles. Each concentric circle has as radius the velocity of an animation, v_{anim}. Once the tracks are defined we divide them into sectors, each of which maps to a clip.

All the animations used must satisfy the following requirements:

- Must be time aligned
- v and α must be approximately the same throughout the animation clip (within a small threshold defined by the user).
- Animation clips must be cyclical.

Each animation clip could be used when the velocity of the agent $v \leq v_{anim}$, therefore we decide on which animation to assign to each sector depending on the α value. The decision points of when to switch from one animation sector to the next as the angle increases is defined as being halfway between the α values of two neighboring animations. Figure 5.3 graphically represents the decision framework, where colors are used to represent the animation clips assigned per sector. We have chosen some animation clips with similar velocities or angles ($v_3 \approx v_4$, $v_6 \approx v_7 \approx v_8$, and $\alpha_1 \approx \alpha_2 \approx \alpha_5$) to graphically show the splitting of tracks into sectors. Only half of the circle of animation clips is shown due to symmetry. At any time during the simulation we can run any animation backwards by using a negative dt and selecting the clip by flipping vertically over the x axis and mirroring in the y axis.

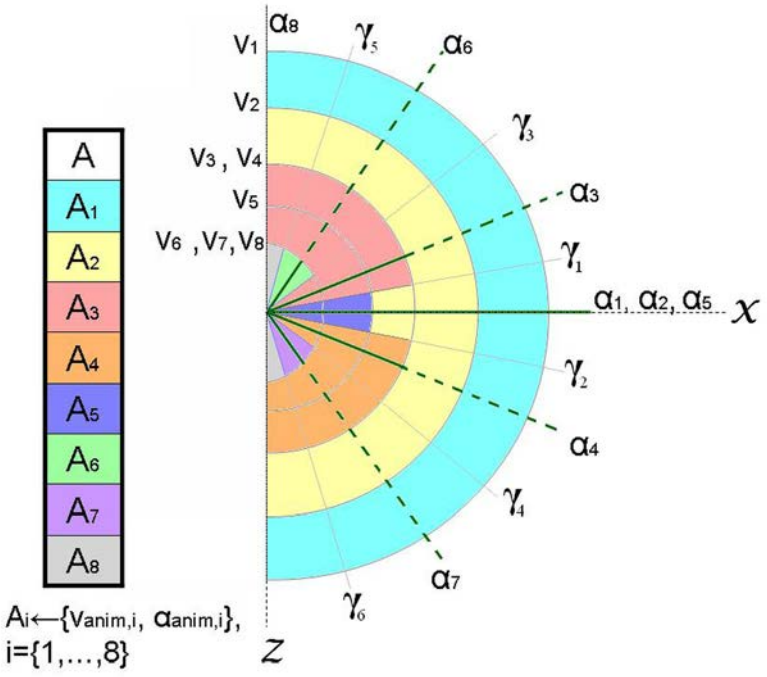


FIGURE 5.3: The Animation Clip Decision Graph: an example with 8 animations.

Classifying Motion Clips

Initially the algorithm starts by dividing the circle into tracks T_1, T_2, ..., T_m, where each track is defined by the velocity of an animation clip from the library (v_{anim}) and $v_1 > v_2 > ... > v_n$. Note that $m \leq n$ as two animations A_i and A_j with similar velocity (i.e. $|v_i - v_j| < \epsilon_v$) will be assigned to the same track. Each track T_i is defined by its maximum and minimum velocity, $T_i = \left\{v_i^{min}, v_i^{max}\right\}$. Starting

from the outer track T_1 (highest velocity), the algorithm proceeds by splitting each track into sectors based on the γ values defined as being halfway between the α values of two animations.

Each sector $S_{i,k}$ inside track T_i is defined by the tuple: $\left\{v_i^{min}, v_i^{max}, \gamma_{i,k}^{min}, \gamma_{i,k}^{max}\right\}$ which corresponds to the minimum and maximum velocity, and minimum and maximum decision angles.

For each step of the algorithm, a track T_{i+1} will be split into at least as many sectors as contained within T_i. For each sector $S_{i,k} = \left\{v_i^{min}, v_i^{max}, \gamma_{i,k}^{min}, \gamma_{i,k}^{max}\right\}$ there will be a new sector $S_{i+1,k} = \left\{v_{i+1}^{min}, v_{i+1}^{max}, \gamma_{i,k}^{min}, \gamma_{i,k}^{max}\right\}$ where $v_{i+1}^{max} = v_i^{min}$, with the same animation being assigned. Then further splitting of those sectors will occur for each of the new animations with depending on the α values of the remaining animations. If we let α_p be the angle of a new animation A_p and α_q be the angle of the previous animation A_q that is assigned to a new sector $S_{i+1,k}$, then if $\left|\alpha_p - \alpha_q\right| < \epsilon_\alpha$, the new animation A_p replaces A_q for that sector. For any other case a new sector is created and the angle limits γ between sectors are recalculated.

The variety of movements that can be synthesized with this method will depend upon the number of animation clips. The more clips we have, the more sectors we can define.

During run time the APM will select the best animation clip based on the current velocity, v, of the agent, and the angle, φ, between the velocity and the torso orientation, Θ, provided by the CS. If a change of animation is required, then the APM will determine the weight, b, necessary for blending between animations.

5.1.2 . Blending Factors

At every frame the CS calculates the new position for each agent based on the desired path to move between an initial position to a destination, while interacting with other agents, walls and obstacles. However, since the CS is not aware of the type of animation being used for the rendering, if we take that new position to translate the root of the skinned avatar and the angle Θ to reorient it, we will observe that the figures appear to "skate", and also that there

is no coherence between the orientation of the avatar and the actual animation movement.

To avoid foot-sliding and guarantee that the animation satisfies the constraints given by v and Θ, we need to ensure that the figure appears to move according to v while the foot currently in contact with the floor stays in place, and the torso faces the direction given by Θ. This could be done using inverse kinematics, but since we are simulating large crowds, we need a method that can be quickly calculated and applied to hundreds of 3D animated figures in real-time. Also, to avoid the time and cost of generating a large number of animation clips, we are interested in a limited number of clips that can be combined to achieve as many realistic animations as possible. We have a trade-off between the accuracy of our animations and the simplicity, and therefore speed, of our calculations. Knowing the velocity of the agent v and the animation velocity v_{anim} from the selected animation A_i, we can calculate the blending factor τ:

$$\tau = \frac{v}{v_{anim}}, \tau \in [0,1] \tag{5.2}$$

The dt needed by the CA module to blend between poses is:

$$dt = \tau \Delta t \tag{5.3}$$

where Δt is the elapsed time between consecutive frames. At this point of our algorithm we have the new pose of the agent and thus can obtain the local coordinates of the root and the feet position.

Knowing that the root movement is driven by the foot that is on the floor during two consecutive poses, we update the root position in the global coordinates of the environment.

5.1.2 . Calculation of Root Displacement

For the current pose P_t, we calculate the vector u_t that goes from the foot on the floor to the root of the skeleton. Likewise for the pose P_{t-1} in the previous frame we calculate u_{t-1} (see Figure 5.4). The vector u_t contains all the rotations that happen at the ankle and knee level and thus provides sufficient information about the leg movement.

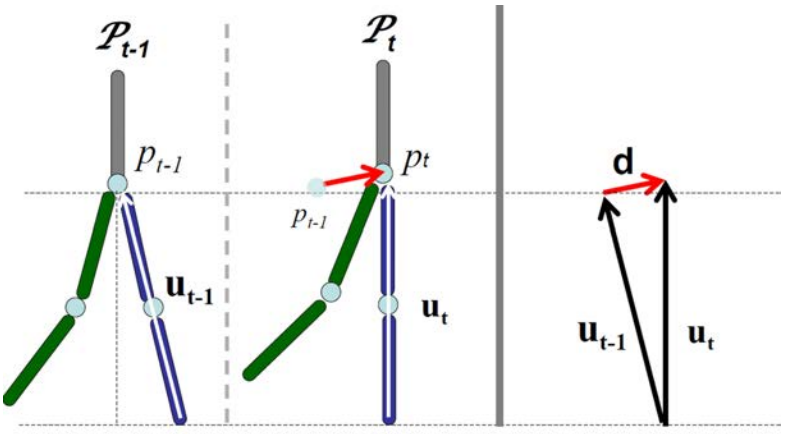


Figure 5.4: Root displacement

By subtraction of vectors we can calculate root displacement d between frames. The vector of displacement d, shown in red is:

$$d = u_t - u_{t-1}$$

And the new position is thus:

$$p_t = p_{t+1} + d$$

The method is efficient enough to allow for fast calculation and extension to 3D is straight forward. From the results shown in Figure 5.5 we can observe that it avoids foot sliding and preserves the vertical root movement that appears when we walk fast. In the figure we have rendered the root path in black.

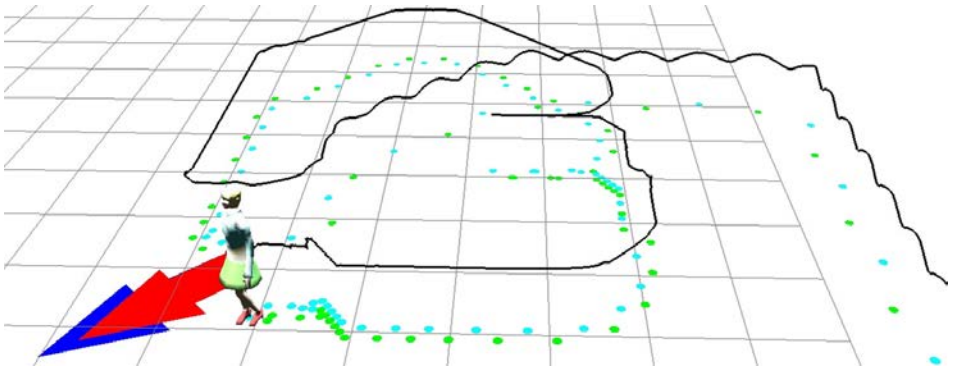


Figure 5.5: Path followed by one agent.

5.1.2 . Updating the Skeletal State

Turns in the environment can be achieved by reorienting the figure according to the velocity vector v and adjusting the torso orientation according to Θ by modifying the skeletal configuration. This will also orient the displacement vector to move the character in the direction indicated by the CS module.

The visual effect of this is that the foot on the floor will slightly rotate in place. This is almost unnoticeable to the human eye, especially at high velocities, and thus is a trade-off worth considering as we can achieve turns in any direction without the requirement of having a large database of animation clips.

For slower velocities, we can achieve higher realism by having a number of simulations where the torso orientation does not necessarily need to be aligned with v. To calculate the rotations we use the following variables:

w	torso orientation vector.
β	angle in the x-z plane of the velocity vector **v** relative to the positive x-axis.
φ	angle of the velocity vector, **v**, relative to the orientation vector, **w**.
α	angle in the x-z plane of the velocity vector $\mathbf{v}_{anim}$ of the animation clip A_i relative to its orientation vector $\mathbf{w}_{anim}$.

Table 5.2: Variables required for upper body rotation.

To achieve movement in the direction given by v, the avatar is rotated by $(\beta - \alpha)$ so that the velocity vector of the animation v_{anim} matches v. Then the torso is oriented according to w: the angle ψ is calculated as the difference between φ and α (Figure 5.6) and propagated across the spine of the character by modifying the current skeletal state of the pose of the character. This allows the agent to move the root in the direction indicated by v while the torso is facing the direction indicated by w.

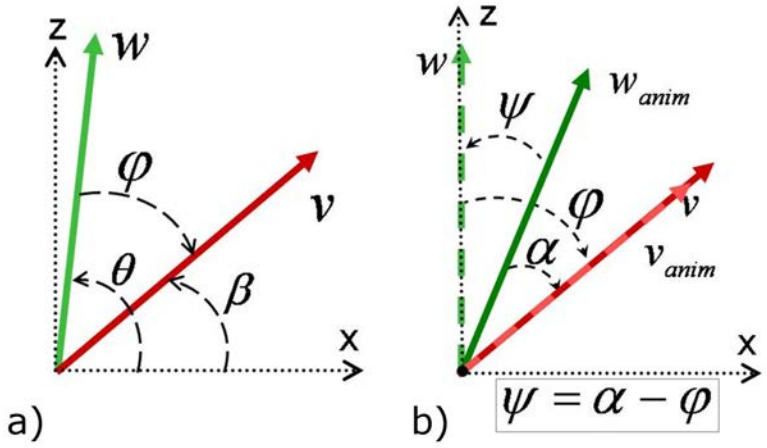


Figure 5.6: Torso Correction. Angles described in table 5.2.

5.1.2 . The Algorithm

Table 5.3 summarizes the APM algorithm with references to the sections where each step was explained.

Section	Algorithm: APM
5.1.2.1	φ := AngleBetween(w, v); A_i := SelectAnimation(v, φ);
5.1.2.2	if $(A_i \neq A_{i-1})$ then b := AgentBlendingFactor(); dt := CalculateDT(v, A_i);
5.1.2.3	P_t := getNextPose(A_i, P_{t-1}, dt); d := CalculateDisplacement(P_t, P_{t-1});
5.1.2.4	α := GetAngleAnim(A_i); β := CalculateAngle(v); $\psi := \alpha - \varphi$; AgentCA.PropagateAngleSpine(P_t, φ);

Table 5.3: The APM algorithm and the sections where each step is explained.

With the parameters calculated by the APM, we apply an absolute rotation of $(\beta - \alpha)$ and add the displacement, d, to the previous position to render our character satisfying the requirement given by the CS.

5.1.3 . Results

The method presented only needs a small set of animation clips to obtain visually plausible results. By increasing the number of animations and/or the quality of those animations (i.e. using motion capture data), we would obtain improved results with no additional cost during simulation time.

The animation library in our examples consists of four walking forward animations, two side-step animations and four "walking on an angle" animations. Per agent and per frame, on a Intel Dual Core 3GHz with 4GB of RAM, the root displacement computation is less than 0.8μs. The whole APM algorithm requires less than 0.021ms. Therefore, we could incorporate the APM into any crowd simulation module, with an additional linear cost per frame of 0.021 ms times the crowd size. Since typically not all the agents in a crowd are visible simultaneously, we could apply the APM algorithm to only the few hundred agents closest to the camera.

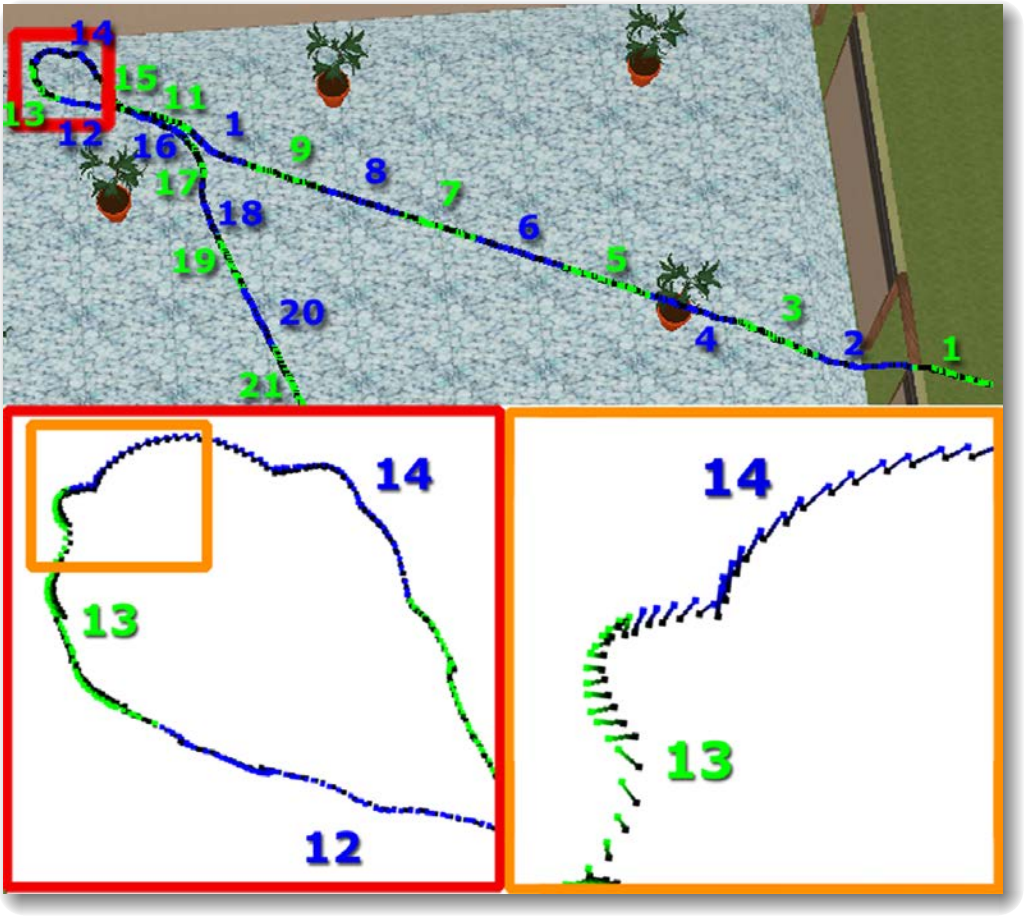


Figure 5.7: Path followed by an agent with close-ups of a turn.

In order to satisfy the constraints given by the *CA*, our APM may need to slightly modify the position of the root given by the *CS*. Figure 5.7 shows the path followed by an agent, with a zoomed view of a sharp. Each blue/-green segment corresponds to 75 frames of the animation (3 seconds). We have represented the deviation introduced by the algorithm as a segment with a green/blue ending indicating the position given by the *CS* and a black one indicating the corrected position calculated by the APM.

Deviation is bigger where there are abrupt turns and blending simultaneously. We have calculated the average deviation between the *CS* position and the APM corrected position in order to determine the impact of our algorithm on the final path. Running at 25 frames per second, on average we obtain a deviation of less than 7.78mm per frame. Larger deviations correspond to segments 13 and 14 which are when the agent is turning sharply (Figure 5.8).

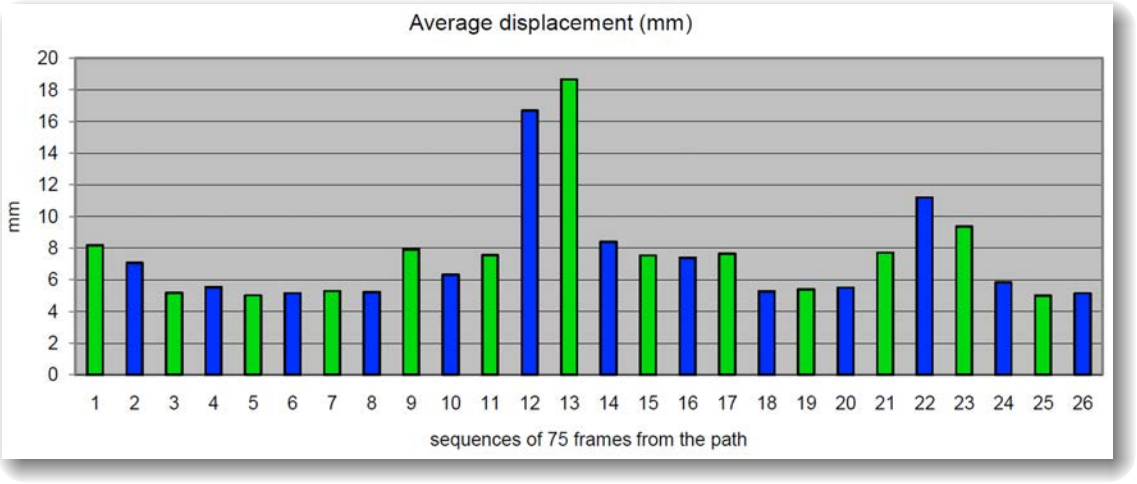


Figure 5.8: Deviations in mm for each segment of the path shown in Figure 5.7

In Figure 5.9 we can see that deviations are larger at turns and reach maximum values when there are repulsion forces between agents (e.g. at the doors where agents congregate). In most of the frames the deviation stays under 1cm.

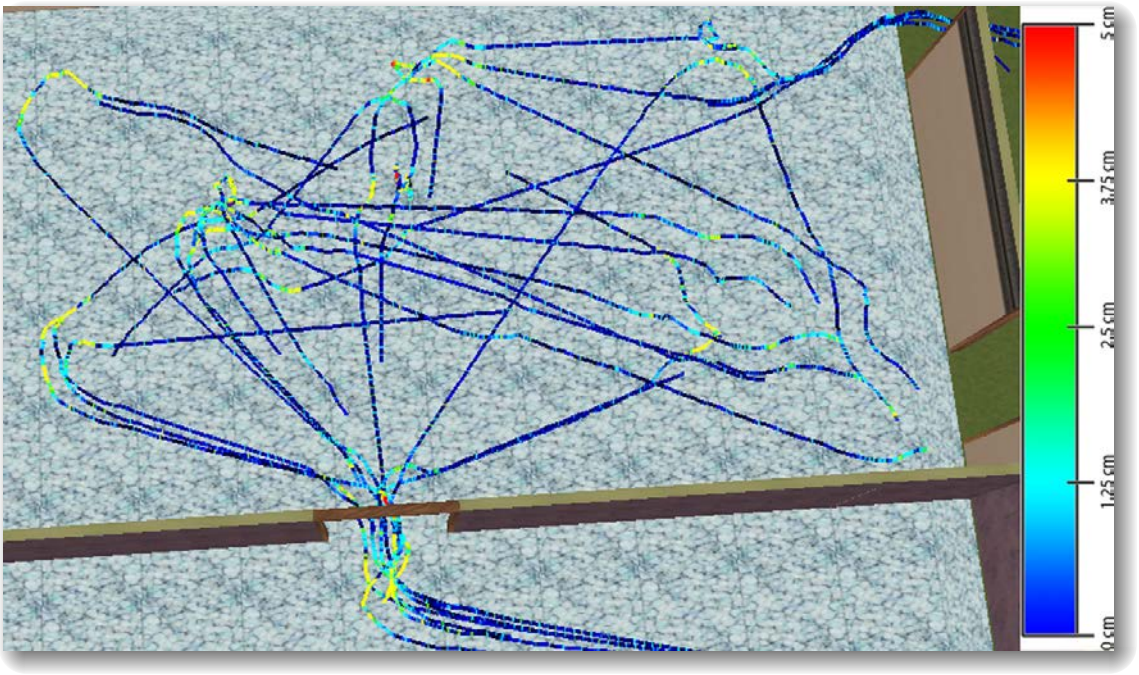


FIGURE 5.9: Paths for 20 agents showing the deviation with a color gradient scale.

5.2 . Synthesizing Motion Following Footsteps

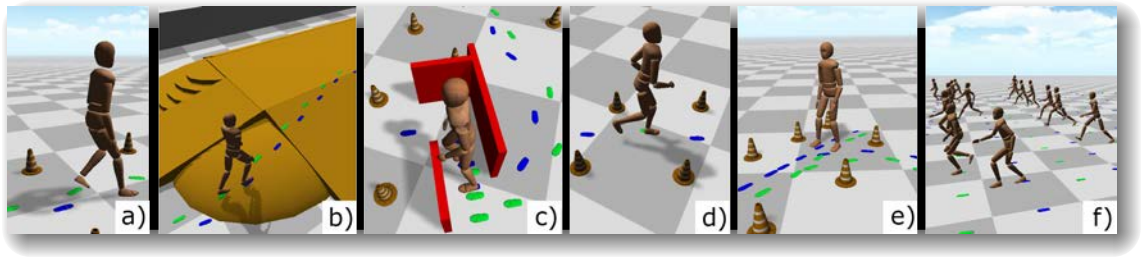


Figure 5.10: An autonomous virtual human navigating a challenging obstacle course (a), walking over a slope (b), exercising careful foot placement constraints including side-stepping (c), speed variations (d), and stepping back (e). The system can handle multiple agents in real time (f).

In this section we present an online animation synthesis technique for fully embodied virtual humans that satisfies foot placement constraints for a large variety of locomotion speeds and styles (see Fig. 5.10). Given a database of motion clips, we precompute multiple parametric spaces based on the motion of the root and the feet. A root parametric space is used to compute a weight for each available animation based on root velocity. Two foot parametric spaces are based on a Delaunay triangulation of the graph of possible foot landing positions. For each foot parametric space, blending weights are calculated as the barycentric coordinates of the next footstep position for the triangle in the graph that contains it. These weights are used for synthesizing animations that accurately follow the footstep trajectory while respecting the singularities of the different walking styles captured.

Blending weights calculated as barycentric coordinates are used to reach the desired foot landing by interpolating between several proximal animations, and IK is used to adjust the final position of the support foot to correct minor offsets, foot step orientation and angle of the underlying floor.

Since foot parametric space only considers final landing positions of the feet without taking into account root velocity, this may lead to the selection of animations that satisfy position constraints but introduce discontinuities in root velocity. To incorporate root velocity fidelity we present a method that can integrate both foot positioning and root velocity fidelity. Our method also allows the system to recover nicely when the input foot trajectory contains steps that

are not possible to perform with the given set on animations (for example, due to extreme distance between steps).

The presented method is evaluated on a variety of test cases and error measurements are calculated to offer a quantitative analysis of the results achieved. Our framework can efficiently animate over sixty agents in real time (25 FPS) and over a hundred characters at 13 FPS, without compromising motion fidelity or character control, and can be easily integrated into existing crowd simulation packages. We also provide the user with control over the trade-off between footstep accuracy and root velocity.

5.2.1 . Framework Overview

Animating characters in real time animations has different requirements depending on the application. In many applications, the user only wants to control the direction of movement and speed of the root, but there are other situation where a finer control of the foot positioning is required. For example the user may want to respect different walking gaits depending on the terrain, to make the character step over stones to cross a river, or walk through some space full of holes whilst avoiding falling. For this purpose we have developed a framework to animate virtual characters following footstep trajectories, while still being able to follow trajectories based on the movement of the COM when necessary. The main issue we address in this work is to provide an animation system that is able to accurately follow footstep trajectories while meeting real-time constraints, and that can be scaled up to handle large groups of animated characters.

For this purpose, we introduce two parametric spaces based on position of each foot: Ω_{f_L} and Ω_{f_R}, and switch between the two depending on the swing foot, as well as a parametric space based on the root movement Ω_{f_R}. Our technique takes into account both displacement (from Ω_{f_L} and Ω_{f_R}) and speed (from Ω_r) to ensure the satisfaction of both spatial and temporal constraints. Our system provides the user with the flexibility to choose between different control granularities ranging from exact foot positioning to exact root velocity trajectories. Fig. 5.11 shows our framework.

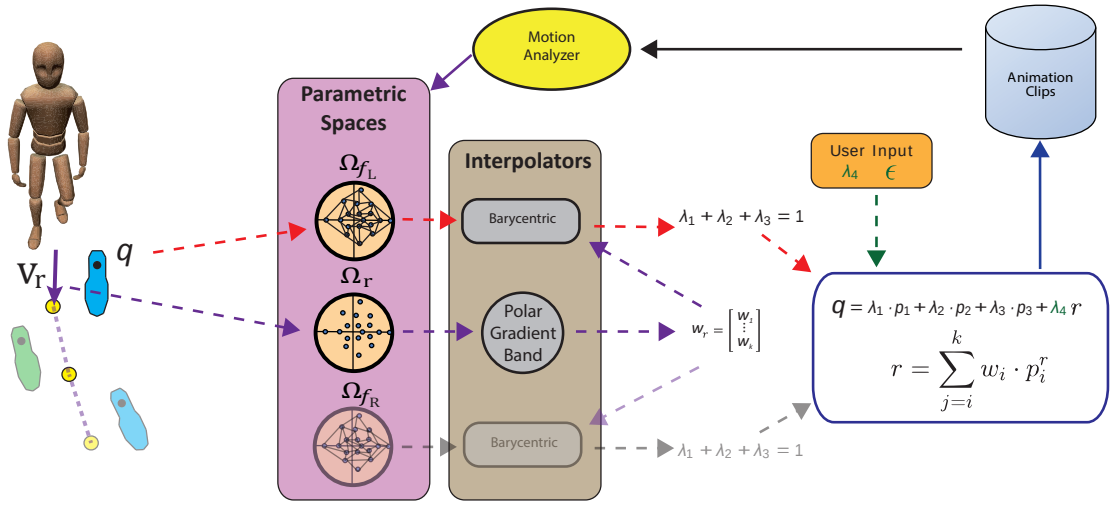


FIGURE 5.11: Online selection of the blend weights to accurately follow a footstep trajectory. Ω_r uses a gradient band polar based interpolator [Johansen, 2009] to give a set of weights w_j, which are then used by the barycentric coordinates interpolator to tradeoff between footstep and COM accuracy.

5.2.2 . FOOTSTEP-BASED LOCOMOTION

The main goal of the Footstep-based Locomotion Controller is to accurately follow a footstep trajectory, i.e., to animate a fully articulated virtual human to step over a series of footplants with space and velocity constraints. The system must meet real-time constraints for a group of characters, should be robust to sparse motion clips, and produce synthesized results that are void of artifacts such as foot sliding and collisions.

5.2.2 . MOTION CLIP ANALYSIS

From a collection of cyclic motion clips [1], we need to extract individual foot steps. Each motion clip contains two steps, one starting with the left foot on the floor, and one starting with the right one foot. A step, is defined as the action where one foot of the character starts to lift-off the ground, moves in the air and finishes when it is again planted on the floor. We say that a footstep corresponds to one foot when that foot is the one performing the action previously described. The foot that stays in contact with the floor for most of the duration of the footstep is called the supporting foot, since it supports the weight of the

[1] Although cyclic animations are not strictly required by our method, they help find smoother transitions between consecutive footsteps and are preferred by most standard animation systems [Johansen, 2009].

body. This applies even for running motions, where the support foot goes into fly mode for a short phase of the footstep, but it is still the one supporting the weight during most of the footstep.

During an offline analysis, each motion clip m_i is annotated with the following information: (1) $\mathbf{v}_i^r$: Root velocity vector. (2) $\mathbf{d}_i^L$: Displacement vector of the left foot. (3) $\mathbf{d}_i^R$: Displacement vector of the right foot.

Similar to [Johansen, 2009], animations are analyzed in place, that is, we ignore the original root forward displacement, but keep the vertical and lateral deviations of the motion. This allows an automatic detection of foot events, such as lifting, landing or planting, from which we can deduce the displacement vector of each foot. For example, the displacement vector of the left foot $\mathbf{d}_i^L$ is obtained by subtracting the right foot position at the instant of time when the left foot lands, from the right foot position at the instant of time when the left foot is lifting off. These displacements will be later used to move the whole character, eliminating any foot sliding. By adding $\mathbf{d}_i^L$ to $\mathbf{d}_i^R$ and knowing the time duration of the clip, we can calculate the average root velocity vector $\mathbf{v}_i^r$ of the clip m_i.

This average velocity is used to classify and identify animations, by providing an example point which is the input for the polar gradient band interpolator (where each example point represents a velocity in a 2D parametric space). Gradient band interpolation specifies an influence function associated with each example, which creates gradient bands between the example point and each of the other example points. These influence functions are normalized to get the weight functions associated with each example. However the standard gradient band interpolation is not well suited for interpolation of examples based on velocities. The polar gradient band interpolation method is based on reasoning that in order to get more desirable behavior for the weight functions of example points that represent velocities, the space in which the interpolation takes place should take on some of the properties of a polar coordinate system. It allows for dealing with differences in direction and magnitude rather than differences in the Cartesian vector coordinate components. For more details we refer the reader to [Johansen, 2009].

Each motion clip is then split into two animation steps A_i^L for the left foot and A_i^R for the right foot. For each foot, we need to calculate all the possible positions that can be reached based on the set of animation steps available. Since

the same analysis is performed for both feet separately, from now on we will not differentiate between left and right for the ease of exposition. For each individual animation step A_i and given an initial root position, we want to extract the foot landing position p_i, if the corresponding section of its original clip was played. This is calculated by summing the root displacement during the section of the animation with the distance vector between the root projection over the floor and the foot position in the last frame.

The set $\{p_i | \forall i \in (1,n)\}$ where n is the number of step animations, provides a point cloud. Fig. 5.12 shows the Delaunay triangulation that is calculated for the point cloud of landing positions. This triangulation is queried in real time to determine the simplex that contains the next footstep in the input trajectory. Once the triangle is selected, we will use its three vertices p_1, p_2 and p_3 to compute the blending weights for each of the corresponding animations A_1, A_2 and A_3.

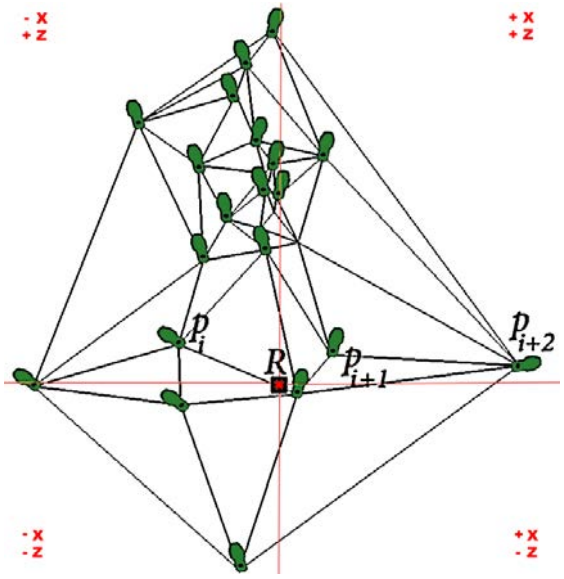


Figure 5.12: Delaunay triangulation for the vertices representing the landing positions (p_i, p_{i+1}, p_{i+2},...) of the left foot when the root, R is kept in place.

5.2.2 . Footstep and Root Trajectories

Our system can work with both footstep trajectories and COM trajectories. A footstep trajectory will be given as an ordered list of space-time positions with orientations, whether it is precomputed or generated on-the-fly.

The input footstep trajectory may be accompanied by its associated root trajectory (a space-time curve, rather than a list of points, and an orientation curve), or else we can automatically compute it from the input footsteps by interpolation. This is achieved by computing the projection of the root on the ground plane, as the midpoint of the line segment joining two consecutive footsteps. The root orientation is then computed as the average between the orientation vectors of each set of consecutive steps. This provides us with a sequence of root positions and orientations which can be interpolated to approximate the motion of the root over the course of the footstep trajectory.

5.2.2 . Online Selection

During run time, the system animates the character towards the current target footstep. If the target is reached, the next footstep along the trajectory is chosen as the next target. For each footstep q_j in the input trajectory $\{q_1, q_2, q_3, ..., q_m\}$ we need to align the Delaunay triangulation graph with the current root position and orientation. Then the triangle containing the next foot position is selected as the best match to calculate the weights required to nicely blend between the three animations in order to achieve a footstep that will land as close as possible to the desired destination position q_j (Fig. 5.13). Notice that these weights are applied equally to all the joints in the skeleton, which means that at this stage we cannot accurately adjust the specific foot orientation required by each footstep in the input trajectory.

5.2.2 . Interpolation

Footstep parameters change between successive footplants, remaining constant during the course of a single footstep (several frames of motion). Therefore we need to compute the best interpolation for each footstep, blend smoothly between consecutive steps, and apply the right transformation to the root in order to avoid foot-sliding or intersections with the ground.

To meet these requirements, we use a barycentric coordinates based interpolator in Ω_{f_L} and Ω_{f_R}, and constrain the solution based on the weights computed in Ω_r. This allows us to animate a character at the granularity of footsteps, while simultaneously accounting for the global motion of the full body.

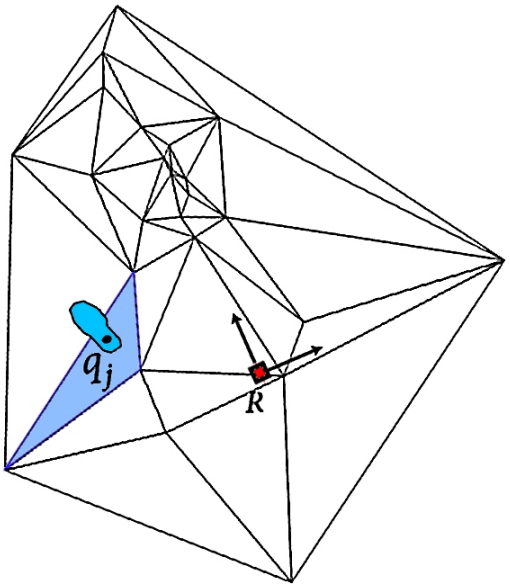


FIGURE 5.13: By matching root position and orientation, we can determine the triangle containing the destination position for the landing position q_j.

If we only consider the footstep parametric space, then the vertices of the selected triangle are the ones that can provide the best match for the desired foot position. The barycentric coordinates of the desired footstep are calculated for the selected triangle as the coordinates that satisfy:

$$q_j = \lambda_1 \cdot p_1 + \lambda_2 \cdot p_2 + \lambda_3 \cdot p_3, \qquad (5.4)$$
$$\lambda_1 + \lambda_2 + \lambda_3 = 1$$

where p_1, p_2 and p_3 are the positions of the foot landing if we run animation steps A_1, A_2 and A_3 respectively. The calculated barycentric coordinates are then used as weights for the blending between animations. A nice property of the barycentric coordinates is that the sum equals 1, which is a requirement for our blending. Finally in order to move the character towards the next position, we need to displace the root of the character adequately to avoid foot sliding. The final root displacement vector, $\mathbf{d}_j^r$ is calculated as the weighed sum of the root's displacement of the three selected animation steps (Eq. 5.5), and changes in orientation of the input root trajectory are applied as rotations over the ball of the supporting foot.

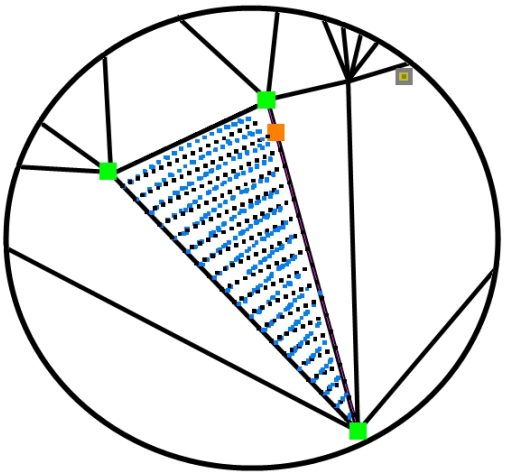


FIGURE 5.14: Offsets for different landing positions in a triangle, between barycentric coordinates interpolation (black dots) and blending the whole skeleton using SLERP (blue dots).

$$\mathbf{d}_j^r = \lambda_1 \cdot \mathbf{d}_1^r + \lambda_2 \cdot \mathbf{d}_2^r + \lambda_3 \cdot \mathbf{d}_3^r \tag{5.5}$$

This provides a final root displacement that is the result of interpolating between the three root displacements in order to avoid any foot sliding. It is important to notice that the barycentric coordinates provide the linear interpolation required between three points in 2D space to obtain the position q_j . This is an approximation of the real landing position that our character will reach, as the result of blending the different poses of the three animation clips, using spherical linear interpolation (SLERP) with a simple iterative approach as described in [Shoulson et al., 2013]. Therefore there will be an offset between the desired position q_j and the position reached after interpolating the three animations. To illustrate this offset, Fig. 5.14 shows the points sampled to compute barycentric coordinates in black, and in blue the real landing positions achieved after applying the barycentric weights to the animation engine and performing blending using SLERP. In order to correct this small offset at the same time that we adjust the feet to the elevation of the terrain and orient the footstep correctly, we incorporate a fast and simple IK solver.

5.2.2 . Inverse Kinematics

An analytical IK solver modifies the leg joints in order to reach the desired position at the right time with a pose as close as possible to the original motion

capture data. For footstep-based control, the desired foot position is already encoded in the footstep trajectory, and for COM trajectories the final position is calculated by projecting the current position of the foot over the terrain. The controller feeds the IK system with the end position and orientation for each footstep. This allows the system to handle footsteps on uneven terrain.

5.2.3 . Incorporating Root Movement Fidelity

In some scenarios the user may be more interested in following root velocities than in placing the feet at exact footsteps or with specific walking styles. We present a solution to include root movement based interpolation in our current barycentric coordinates based interpolator through a user controlled parameter λ_4.

For this purpose, we incorporate the locomotion system presented by Johansen [Johansen, 2009] to produce synthesized motions that follow a COM trajectory with correction for uneven terrain. During offline analysis, a parametric space is defined using all the root velocity vectors extracted from the clips in the motion database. For example, a walk forward clip at 1.5 m/s, and a left step clip at 0.5 m/s produces a parametric space using the root velocity vectors going from the forward direction to the 90^o direction, and with speeds from 0.5 m/s to 1.5 m/s.

Given a desired root velocity we define a parametric space Ω_r, and a gradient band interpolator in polar space [Johansen, 2009] is created to compute the weights for each animation clip to produce the final blended result. The gradient band interpolator does not ensure accuracy of the produced parameter values but it does ensure smooth interpolation under dynamically and continuously changing parameter values, as with a player-controlled character. Once the different clips are blended with the computed weights, the system predicts the support foot position at the end of the cycle and projects it on the ground to find the exact position where it should land.

The root movement based interpolator will select a set of k animations A_1^r to A_k^r with their corresponding weights: $w_1,...,w_k$. Each of those animations provides a landing position $p_1^r,...,p_k^r$, and if we only interpolated these animations we would obtain the landing point r.

In order to incorporate the output of the polar gradient band interpolator in the barycentric coordinates based interpolator we proceed as indicated in Algorithm 1.

Algorithm 1 Incorporating root movement fidelity

Input:

- The target position q_j,
- The current triangle $\langle p_1, p_2, p_3 \rangle$,
- Root landing positions $\langle p_1^r, ..., p_k^r \rangle$,
- Animation weights $\langle w_1, ..., w_k \rangle | w_1 \geq ... \geq w_k$,
- A user input threshold ϵ,
- A user input weight parameter λ_4

Output: $\lambda_1, \lambda_2, \lambda_3$

1: **for** $i = 1$ **to** 3 **do**
2: $u \leftarrow (i+1) \mod 3$
3: $v \leftarrow (i+2) \mod 3$
4: $j \leftarrow 1$
5: $replaced \leftarrow$ false
6: **while** $j \leq 3 \wedge \neg replaced$ **do**
7: **if** $\left\| p_i - p_j^r \right\| \leq \epsilon \wedge IsInTriangle\left(q_j, \langle p_j^r, p_u, p_v \rangle\right)$ **then**
8: $p_i \leftarrow p_j^r$
9: $replaced \leftarrow$ true
10: **end if**
11: $j \leftarrow j + 1$
12: **end while**
13: **end for**
14: $r \leftarrow CalculateRootLanding\left(\langle p_1^r, ..., p_k^r \rangle, \langle w_1, ..., w_k \rangle\right)$
15: $\langle \lambda_1, \lambda_2, \lambda_3 \rangle \leftarrow ComputeWeights(\langle p_1, p_2, p_3 \rangle, \lambda_4, r)$

The algorithm first checks whether a vertex of the current triangle $\langle p_1, p_2, p_3 \rangle$ can be replaced by any of the three vertices with highest weights selected by the polar band interpolator, p_j^r, $j \in [1, k]$ (lines 1-13 in the algorithm). This replacement takes place if the distance between the two landing positions p_i and p_j^r is within a user input threshold ϵ (line 7), and the resulting triangle still contains the desired landing position q_j (function *IsInTriangle* returns true is q_j is inside the new triangle). This means that there is another animation that also provides a valid triangle and has a root velocity that is closer to the input root velocity.

Next, function *CalculateRootLanding* computes the landing position reached after blending the animations given by the root movement interpolator (Eq. 5.6).

$$r = \sum_{i=1}^{k} w_i \cdot p_i^r \tag{5.6}$$

Finally, *ComputeWeights* calculates the three λ_i for the next footstep q_j by incorporating a user provided λ_4 and the result of the polar band interpolator r (Eq. 5.7).

$$q_j = \lambda_1 \cdot p_1 + \lambda_2 \cdot p_2 + \lambda_3 \cdot p_3 + \lambda_4 \cdot r \tag{5.7}$$

and λ_i are defined using the following relationship:

$$\lambda_1 + \lambda_2 + \lambda_3 + \lambda_4 \cdot \sum_{i=1}^{k} w_i = 1 \tag{5.8}$$

Since w_i and p_i^r are known $\forall i \in \{1,...,k\}$, and λ_4 is a user input, we have a linear system, where λ_4 determines the trade-off between following footsteps accurately (if $\lambda_4 = 0$), and simply following root movement (if $\lambda_4 = 1$).

Given any set of animation clips, if we allow the user to increase λ_4 until it reaches 1, there will be a maximum value $\beta \in [0,1]$, for which one of the barycentric coordinates of the vertices of the triangle will give a negative value. Since we do not want to obtain negative weights, we determine β as the maximum value for which our system can accurately follow footstep trajectories. If we further increase λ_4 beyond the value β then the algorithm will provide the blending values that correspond to a new point q' which slowly moves along the line joining the desired landing position q and the point p^r. Therefore when $\lambda_4 = 1$ the resulting blending will be exclusively the one provided by the root movement trajectory since $\lambda_1 = \lambda_2 = \lambda_3 = 0$. Fig. 5.15 illustrate this situation.

Time Warping. Incorporating root velocity in the interpolation, does not always guarantee that the time constraints assigned per footstep will be satisfied. Therefore once we have the final set of animations to interpolate between, with their corresponding weights λ_i, $i \in \{1,2,3\}$ and w_j, $j \in [1,k]$, we need to apply time warping. Each input footstep f_m has a time stamp τ_m indicating the time at which position q_m should be reached (where $m \in [1,M]$ and M is the number of footsteps in the input trajectory). The total time of the current motion, T

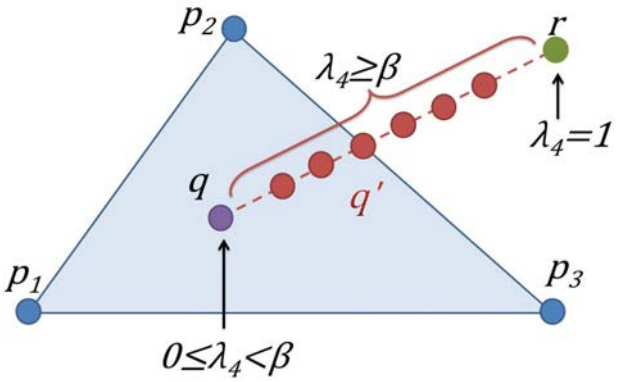


FIGURE 5.15: The barycentric coordinates of q for $\lambda_4 \geq \beta$ will make λ_1, λ_2 or λ_3 negative. Therefore we need to move the landing location q' from q to r as the user increases the value of λ_4 between β and 1.

can be calculated as the weighted sum of the time of the animation steps being interpolated: $T = \sum_{i=1}^{3}(\lambda_i \cdot t(A_i)) + \sum_{j=1}^{k}(w_j \cdot t(A_j))$. Therefore the time warping factor that needs to be applied can be calculated as: $warp_m = (\tau_m - \tau_{m-1})/T$.

Outside the Convex-Hull. The foot step parametric space defines a convex-hull delimiting the area where our character can land its feet. When our target footstep position falls inside this area, clips can be interpolated to reach that desired position. But if it falls outside this convex-hull, and we want the system to still consider it and try to reach it. Our solution to handle this problem, consists on projecting orthogonally the input landing position q over the convex-hull to a new position q_{proj}. Our system gives then the blending weights for q_{proj} and applies IK to adjust the final position. We include a parameter to define a maximum distance for the IK to set an upper limit on the correction on the landing position. It is important to notice that even if the input trajectory has some footsteps that are unreachable with the current data base of animation clips, our system will provide a synthesized animation that will follow the input trajectory as closely as possible, until it recovers and catches up with future steps in the input trajectory. This situation is similar to the scenarios where the user increases λ_4 and then reduces it again.

5.2.4 . Results

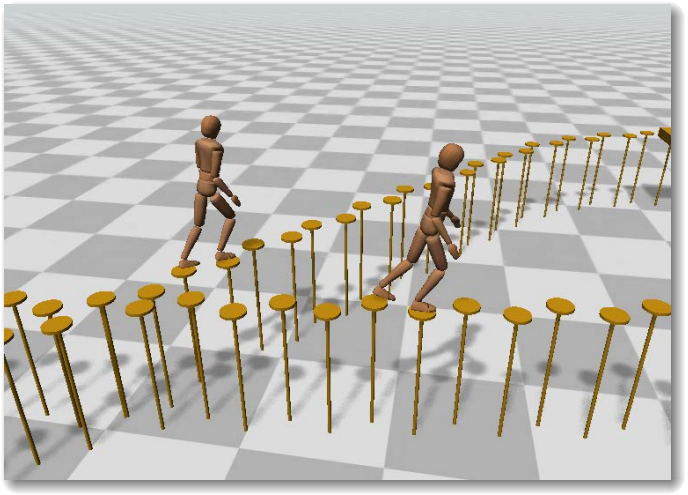
The animation system described in this section is implemented in C# using the Unity 3D Engine [Unity, 2014]. The footstep trajectories used to animate the characters are generated using the method described in [Singh et al., 2011] or are created by the user. Some difficult scenarios, exercising careful footstep selection are shown in Fig. 5.10 and Fig. 5.16. Agents carefully plant their feet over pillars (Fig. 5.16-a), use stepping stones to avoid falling into the water (Fig. 5.16-b), and even scale to handle over a hundred agents at 13 FPS (Fig. 5.16-c and Fig. 5.18).

Obstacle Course. We exercise the locomotion dexterity of a single animated character in an obstacle course. The character follows a footstep trajectory with different walking gaits , alternating running and walking phases (Fig. 5.10-a,b), and including sidesteps (Fig. 5.10-c) and backward motion (Fig. 5.10-e).

Stepping Stone Problem. Stepping stone problems (Fig. 5.16-b) require careful footstep level precision where the environment constraints require the character to place their feet exactly on top of the stones in order to successfully navigate the environment. Our framework can be coupled with footstep-based controllers to solve these challenging benchmarks.

Integration with Crowd Simulator. We integrate our animation system with footstep-based simulators [Singh et al., 2011]; our character follow the simulated trajectories without compromising its motion fidelity while scaling to handle large crowds of characters (Fig. 5.16-c).

It is important to mention that the quality of the results depends strongly on the quality of the clips available from the motion capture library. As can be seen in the video, the least precise movements in our results are side steps and back steps. This is due to two reasons: (1) we had a small number of animations compared to other walking gaits, and thus triangles covering that space have larger areas, and (2) interpolation artifacts appear when blending between animations that move in opposite directions (for example a backwards step with a forward step). We believe that having a better and denser sampling in these areas will improve the results. For steps falling in triangles of smaller areas, and with all the vertices in the same quartile we have obtained results of high quality even for difficult animations such as running or performing small jumps.



(a)

(b)

(c)

FIGURE 5.16: (a) Agents accurately following a footstep trajectory and avoiding falls by carefully stepping over pillars. (b) The stepping stone problem is solved with characters avoiding falls into the water. (c) A crowd of over 100 agents simulated at interactive rates.

5.2.4 . Foot Placement Accuracy

The presented barycentric coordinates interpolator assumes a small offset between the results of linearly interpolating landing positions from the set of animations being blended, and the actual landing position when calculating spherical linear interpolation over the set of quaternions. This small offset depends on the area of the triangle, so as we incorporate more animations into our data base, we obtain a denser sampling of landing positions and thus reduce both the area of the triangles and the offset. We believe this is a convenient trade off since such a small offset can be eliminated with a simple analytical solver but the efficiency of computing barycentric coordinates offers great performance. It is also important to notice, that if exact foot location is not necessary, and the user only needs to indicate small areas for stepping as in the watter scenario, then it is not necessary to apply the IK correction. Fig. 5.17 shows the offset between the landing position and the footstep. The magnitude of the error is illustrated as the height of the red cylinders that are located at the exact location where the foot first strikes.

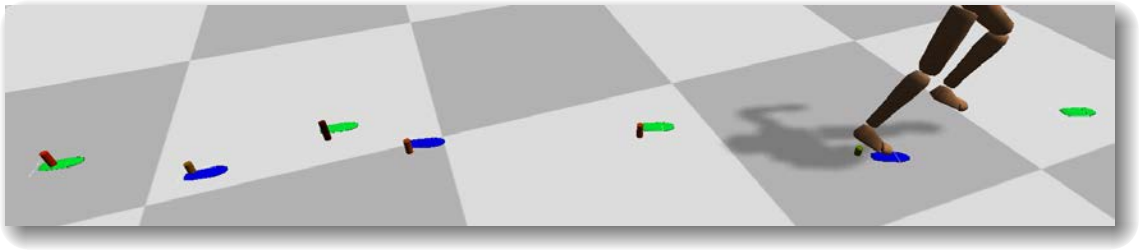


Figure 5.17: The red columns show the small offset between landing position and the footstep when the IK corrections are not being applied.

5.2.4 . Performance

Fig. 5.18 shows the frame rate we obtain as we double the number of agents. It is important to notice that increasing the number of animations would enhance the quality and accuracy of the results, with just a small overhead on the performance.

The average time of the locomotion controller is 0.43ms, this process includes blending animations, IK, the polar band interpolator and our barycentric coordinates based interpolator. The computational cost of our footstep interpolator is 0.2 ms, which is amortized over several frames as the interpolation in

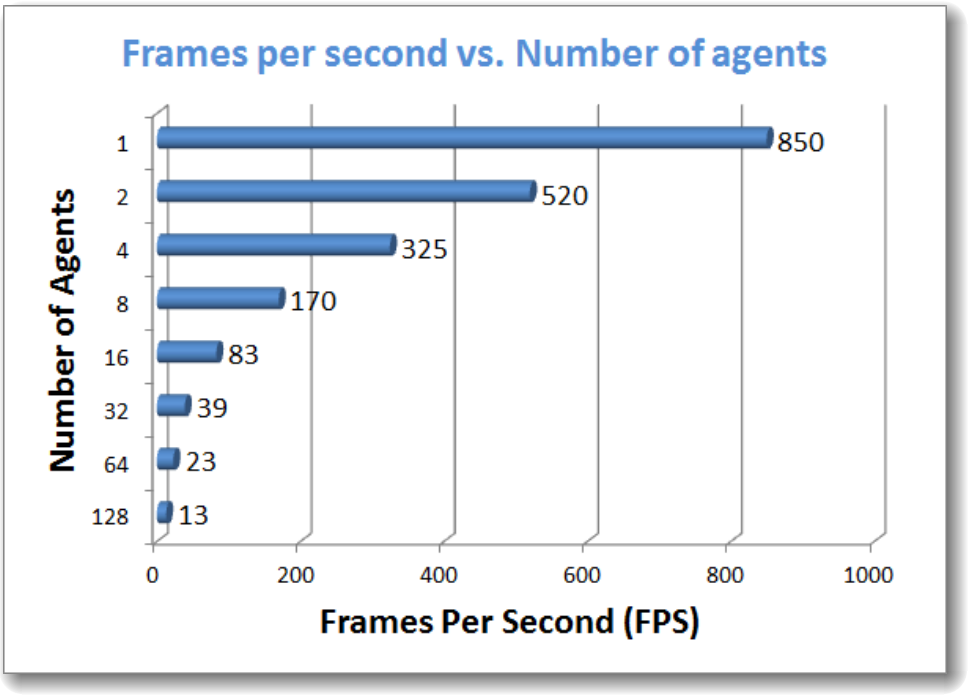


FIGURE 5.18: Performance of the Footstep Locomotion System in frames per second as the number of agents increases.

Ω_{f_L} or Ω_{f_R} only need to be performed once per footstep. This time is divided between computing the root movement polar band interpolator which takes 0.155ms and our barycentric coordinates interpolator which takes 0.045ms. Performance results were measured on an Intel Core i7-2600k CPU at 3.40GHz with 16GB RAM.

5.3 . Conclusions on Crowd Animation

Our first contribution in crowd animation has been a realistic, yet computationally inexpensive, method to achieve natural animation without foot-sliding for crowds. Our goal was to free the crowd simulation module from the computational work of achieving natural looking animations and focus instead on developing crowd behavior that looks realistic. We proved that natural looking results can be obtained with a minimal library of animation clips and that this method can be integrated with any crowd simulation software, at the expense of slightly modifying the positions of the agents.

We can classify the APM as a motion parametrization approach within the example based methods of locomotion synthesis. Using only root velocities and positions of the agents it synthesizes locomotion in a natural way. The advantage is that it is a robust and efficient method that does not require a large amount of motions in our database. These were manually created in our experiments, although we believe results would be much better with motion capture animations. The low computational cost of this approach also guarantees that it can easily been scaled up to crowds of hundreds or thousands of agents.

With the APM we also have a presented a simple yet effective method to remove foot sliding. By using the real displacement of the root within the current synthesized motion, we avoid any possible skating. The major drawbacks are the need of slightly modifying the end position of the agent in the simulation, and the need to detect which foot is on the floor.

We have also presented a system that uses multiple parameter spaces to animate fully embodied virtual humans to accurately follow a footstep trajectory respecting root velocities, using a relatively small number of animation clips (24 in our examples). Our method is fast enough to be used with tens of characters in real time (25 FPS) and over a hundred characters at 13 FPS. The method can handle uneven terrain, and can be easily extended to introduce additional locomotion behaviors by grouping new sets of animation clips and generating different parametric spaces. For example, walking and running motions can be blended together, but if we wanted to add crawling motions or jumping motions, it would be better to separate them in different parametric spaces for each

style. This will avoid unnatural interpolations that can appear when blending between very different styles. Having different parametric spaces requires some sort of classification, which could initially be done manually but it could also be based on the characteristics of the motion, such as changes in acceleration, maximum heights of the root, length of fly phase, etc.

We do not run physical or biomechanical simulations, and use interpolation and blending between motion capture animations. Our method accuracy depends on the variety of animation clips, while its quality and efficiency depends on the number of clips. A trade-off between efficiency and accuracy is therefore necessary, for which we have found a good equilibrium.

One limitations is that in order to reduce the dimensionality of the problem, we have not included in our parametric space the orientation of the previous footstep. Ignoring the final orientation of the character at the end of the previous step can induce some discontinuities between footsteps. We mitigate this effect by blending between footsteps automatically for a small amount of time (about 0.2 seconds) at the advantage of reducing the computational time and thus making our method suitable for large groups of agents in real time.

Regarding the selection of animation at the end of each foot step, notice that in our database, left and right animation steps are extracted from complete animation cycles that are usually consistent in parameters such as velocity, acceleration and walking gait. Therefore for a given sequence of steps, the most likely animation steps to be chosen will be those extracted from the same set of animation cycles, thus resulting in smooth and natural transitions between very similar steps. When the characteristics of the steps change drastically, then our method needs to blend between steps from very different animation cycles. So in general, alternating left/right steps results in natural transitions with smooth continuity when blending animations, and only when the input step trajectory changes drastically between each pair of steps, we may observe transitions between animations that feel unnatural. This can happen if the step trajectory is done manually with artifacts due to the user lack of experience creating footstep trajectories, or for example when the input trajectory forces the character to walk over artificially located steps, like crossing a river by stepping over stones. We would like to empathize that this situation would also look awkward in the real world and thus the result of our synthesize animation may be the desired one.

These two approaches have a common goal which is generating natural animations for crowds from a small library of motion clips, but they work with different input, root trajectories for the former, and foot-step trajectories for the later. Recalling our contributions in crowd simulation, these two kind of outputs could correspond to different granularities of control, that could for example be applied for different levels of detail. For example closer agents could be simulated using footsteps while agents in the background could just use root trajectories. Another possibility could be to replace the polar gradient band interpolation of our second approach, used to switch to root control, by our APM approach.

Publications

Our work on animation has produced the following publications ([Beacco et al., 2010b, Pelechano et al., 2009, 2011]):

- N. Pelechano, B. Spanlang and A. Beacco. *A framework for rendering, simulation and animation of crowds*. EUROGRAPHICS Spanish Conference of Computer Graphics (EGse CEIG 2009), Donostia (San Sebastian), Spain. 9-11 September 2009.
- A. Beacco, B. Spanlang, and N. Pelechano. *Efficient elimination of foot sliding for crowds*. In Posters Proceedings, The ACM SIGGRAPH/EUROGRAPHICS Symposium on Computer Animation (SCA 2010), pages 19-20, Madrid, Spain 2010
- N. Pelechano, B. Spanlang, and A. Beacco. *Avatar locomotion in crowd simulation*. In International Conference on Computer Animation and Social Agents (CASA 2011), Chengdu, China, 2011

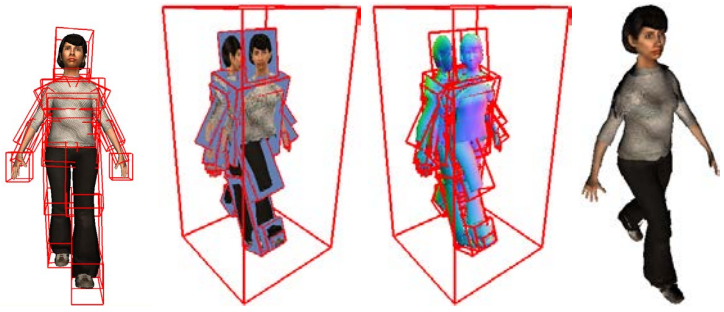
Furthermore, we have submitted to a journal our work on synthesizing motion accurately following footsteps (Section 5.2). Thanks to Professor Norman I. Badler and Doctor Mubbasir Kapadia, for his collaboration in that work. Also, thanks to Doctor Bernhard Spanlang, for his support and collaboration when working with Halca [Spanlang, 2009].

6 . Contributions to Crowd Rendering

Rendering detailed animated characters is a major limiting factor in crowd simulation. In this chapter we present two methods for rendering thousands of animated characters in real-time. We maximize rendering performance by using a collection of pre-computed impostors sampled from a discrete set of view directions. The first method is based on relief impostors and the second one in flat impostors. Our work differs from previous approaches on view-dependent impostors in that we use per-joint rather than per-character impostors. Characters are animated by applying the joint rotations directly to the impostors, instead of choosing a single impostor for the whole character from a set of predefined poses. This representation supports any arbitrary pose and thus the agent behavior is not constrained to a small collection of predefined clips. To the best of our knowledge, this was the first time a crowd rendering algorithm encompassing image-based performance, small GPU footprint and animation-independence was proposed.

6.1 . Relief Per-Joint Impostors

Relief mapping [Oliveira et al., 2000] has been proven to be a powerful tool to encode detailed geometry and appearance information. Most importantly, since relief maps support efficient random-access, impostors based on relief mapping are output sensitive, i.e. their rendering cost is roughly proportional to the area of their screen projection. This feature makes relief impostors especially suitable for accelerating the rendering of scenes involving a huge number of objects.



(a) (b) (c) (d)

(e)

Figure 6.1: Overview of our approach: A bounding box is created for each articulated part of an animated character (a). Color (b), normal (c) and depth information is projected onto the box faces, which are rendered through relief mapping (d). Image (e) shows a crowd with about 5,000 agents, all of them rendered with our relief impostors.

In this section, we present a new representation for animated characters (Figure 6.1) which uses relief impostors to represent the different body parts of the character delimited by the bones of the skeleton. Each character is encoded through a collection of oriented bounding boxes, where each box represents the geometry influenced by a particular bone. Textures are projected orthogonally onto the six faces of each box. For each face we store two textures encoding color, normal and depth values. During animation the bounding boxes are transformed rigidly by a vertex shader according to the transformation of the associated bone in the animated skeleton. A fragment shader efficiently recovers the details of the avatar's skin and clothing using an adapted version of relief mapping.

Unlike competing output-sensitive approaches, our compact representation has no preprocessing requirements (construction can be performed at load time) and does not require us to predefine the animation sequences nor to select a subset of discrete views. Our performance experiments show a significant improvement with respect to geometry rendering. We have also conducted user perception tests validating our technique for rendering agents at middle and far distances from the observer.

Relief Mapping

Among image-based techniques, relief mapping [Policarpo et al., 2005] has proven to be useful for recovering high-frequency geometric and appearance details. Relief maps store surface details in the form of a height field. Typically the RGB channels encode a normal or a color map, while the alpha channel stores quantized depth values. The programmability of modern GPUs allows us to recover the original geometry by a simple ray-heightfield intersection algorithm executed in the fragment shader [Policarpo et al., 2005]. Acceleration techniques for computing the ray-heightfield intersection include, among others, linear search plus binary search refinement [Policarpo et al., 2005], varying sampling rates [Tatarchuk, 2006], precomputed distance maps [Baboud and Décoret, 2006], cone maps [Policarpo and Oliveira, 2007] and quadtree relief-mapping [Schroders and Gulik, 2006].

Other recent techniques adopt a relief mapping approach to encode details in arbitrary 3D models with minimal supporting geometry [Andujar et al., 2007, Baboud and Décoret, 2006]. Unfortunately, these output-sensitive approaches are limited to static geometry.

Only a few works attempt to animate geometry encoded as relief impostors. In [Pamplona et al., 2008] the animator is requested to create an animation by manually defining and moving a few control points in texture space. Radial basis functions are used to warp the original image by texture coordinate modification. The above method suffers from two major limitations: control points defining the animation are just moved in 2D, providing only image-warp animation, and it does not support standard skeletal animation.

6.1.1 . Our Approach

6.1.1 . Overview

We aim at increasing the number of simulated agents in real-time crowd simulations by reducing the rendering cost of individual agents. This involves using a simple representation for animated characters supporting output-sensitive rendering, so that rendering times are roughly proportional to the number of rendered fragments, instead of depending on the complexity of the underlying surface. Therefore only characters that are very close to the observer are rendered as polygonal meshes, while the rest of the agents are rendered using our new relief impostor method. We assume the input character conforms to the de facto standard in character animation and thus consists of a textured polygonal mesh (skin), a hierarchical set of bones (skeleton) and vertex weights. We assume that both the skin and the skeleton have been designed in a reference pose. The nodes of the skeleton represent joints and the edges represent the bones. Since each bone can be easily identified by its origin, we can use the term joint interchangeably. The transformations affecting joints in the hierarchy are assumed to be rigid. The vertex weights describe the amount of influence of each joint on each vertex.

Since we want to keep preprocessing and memory costs at a minimum while still supporting real-time mixing of animation sequences, we use a separate relief impostor for each animated part of the articulated character. Our representation for distant characters consists of a collection of oriented bounding boxes (OBB), one for each bone in the skeleton, along with a collection of textures projected into the OBB faces, encoding color, normal and depth values (Figure 6.1). The OBB will be transformed in the same way as the bones of the skeleton, giving the impression that our impostor character is animated.

Our approach differs from previous work in several aspects. First, we do not attempt to animate a single relief impostor representing a whole character [Pamplona et al., 2008], but to provide relief impostors representing an already animated character. Second, we require much less memory than competing image-based approaches which require prerendering the character for every possible animation frame for a large set of view angles.

Third, compared to previous image-based techniques, the cost of adding new characters is drastically lower as new animations can be added at no cost at all. Furthermore, our technique allows blending animations and also running animations at arbitrary speeds (including slow-motion) since we are not limited to a discrete set of animation frames.

Finally, our method provides a detailed rendering for any character, viewpoint, and animation sequence.

Our implementation relies on the Halca animation library [Spanlang, 2009] to draw the animated characters from which we create our impostors. Halca is a hardware-accelerated library for character animation which is based on the Cal3D XML file format [Cal3D, 2014] to describe skeleton weighted meshes, animations, and materials. Our current implementation works with any animated avatar and any animation that can be exported to the Cal3D format.

6.1.1 . Construction

The construction of our relief impostors from a given 3D character proceeds through the following steps, described in detail below:

1. Associate mesh triangles with impostors.
2. Select a suitable pose for capturing the impostors.
3. Compute the bounding boxes with the chosen pose.
4. Capture the textures of each bounding box.

Step 1. We start by assigning mesh triangles with impostors, where each impostor corresponds to a joint of the articulated character. We assume that each input vertex v_i is attached to joints $J_1,\dots J_n$ with weights $w = (w_1,\dots w_n)$. Now the problem is, given a triangle with vertices v_1, v_2, v_3, to decide which impostors the triangle will be attached to. This determines which triangles will be captured by the impostor. Since we want to keep preprocessing tasks at a minimum, we only tested simple, automatic solutions.

One extreme option is to allocate mesh triangles to joints, attaching each triangle to a single joint. More specifically, each triangle is attached to the joint

having the largest influence over the triangle (the influence of a joint over a triangle is computed as the sum of the joint weights over the triangle's vertices). It turns out that this partition tends to produce visible gaps in the joint boundaries during animation; the higher the deviation with respect to the reference pose, the larger the resulting gaps.

The opposite approach would consist of attaching each triangle to a bone if at least one of its vertices is influenced by the bone, regardless of the corresponding weight. Therefore some triangles (those around joints) would be attached to a variable number of impostors, resulting in overlapping parts among joints. These overlapping parts produce protruding geometry when the character is rendered in a pose other than the capture pose.

Therefore we propose an optimized strategy where triangles are assigned to joints based on a given user-defined threshold ϵ. In this approach a triangle is attached to a joint if any of its vertices has a weight above ϵ. This approach assigns triangles to one or more joints, except for triangles where none of the weights are above the threshold. In this particular case we fall back to the first approach, i.e. the triangle is assigned to the joint with the highest influence. On the one hand high values for ϵ result in less protruding artifacts, but on the other hand low values for ϵ result in less cracks. Experimentally, we found that a threshold $\epsilon = 0.5$ worked well on all our test characters, minimizing both cracks and protruding artifacts (Figure 6.5).

Notice that all the strategies above only use vertex weights and thus are pose-independent.

Step 2. The second step is to choose a suitable pose for capturing the impostors. Triangles are captured according to the chosen pose, i.e. after mesh vertices have been blended according to the pose (using linear blend skinning). This choice of pose affects the captured geometry. Ideally, we should select a pose representing a somewhat average pose of the animation sequence.

For example, if the animation sequence shows a character walking with the arms in a rest position, it is better to capture the triangles around the shoulder with the arms in such a position rather than when stretching arms out sideways. Since impostors will undergo only a rigid transformation, choosing a pose corresponding to a walking animation keyframe tends to minimize

artifacts around joints. Our current implementation uses a pose from a walking animation sequence, rather than the reference pose. Notice that the above choice only affects triangles influenced by multiple joints; triangles influenced by a single joint will be reconstructed in their exact position regardless of the selected pose.

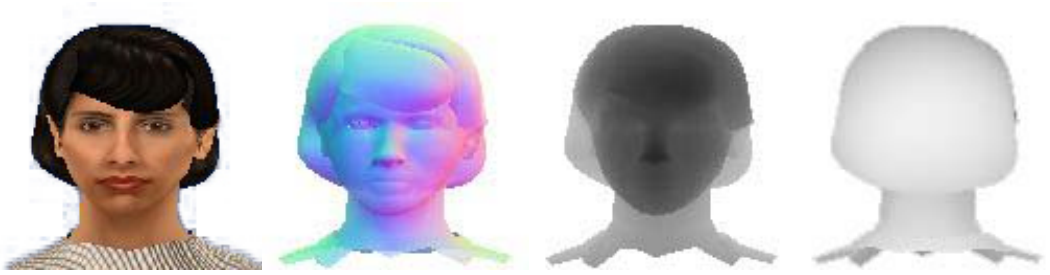
Step 3. Once a suitable pose has been chosen, we deform the mesh accordingly by applying linear blend skinning to the mesh vertices, i.e. the transformed vertex v' is computed as $v' = \sum w_i M_{Ji} v$, where M_{Ji} is the rigid transformation matrix from the reference-pose of joint J_i to its actual position in the chosen posture. The bounding box of each impostor is then computed as the oriented bounding box (OBB) of the (transformed) triangles attached to the impostor.

Step 4. The last step is to render the deformed mesh to capture the relief maps corresponding to each one of the six faces of its bounding box. For each bounding box face, we set up an orthographic camera with its viewing direction aligned with the face's normal vector, and then render the triangles attached to the corresponding impostor. We capture the following RGBA textures (Figure 6.2):

- Color map: the RGB channels encode the color, and the alpha channel encodes the minimum (front) depth value z_f.
- Normal map: the RGB channels encode the normal vector, and the alpha channel encodes the maximum (back) depth value z_b.

Front depth values are captured by rendering the attached triangles with the default GL_LESS depth comparison function. Likewise, back depth values are captured by clearing the depth buffer with a zero value (instead of the default unit value) and switching depth comparison to GL_GREATER. Although storing both depth values is redundant (front depth values of a face equal one minus back depth values of the opposing face), we have chosen this option to improve the locality of texture fetches during rendering.

In order to speed up rendering we reorganized the textures to have both depth values in the same texture. We still keep only two textures, but now the four



(A) (B) (C) (D)

FIGURE 6.2: Color (a), normal (b), front depth (c) and back depth (d) values are encoded as two RGBA textures.

channels of the first texture encode (R,G,z_f,z_b) and the second texture encodes (B, n_x, n_y, n_z). This reduces to one half the number of texture fetches during the ray-heightfield intersection step of relief mapping.

The vertices of all bounding boxes of a character are stored in a single Vertex Buffer Object (VBO), which is shared by all the instances of the same character.

Color and normal maps of each character are stored in texture arrays to avoid texture switching while rendering the instances of the same articulated character.

Since a typical animated character for crowd simulation consists of about 21 bones, this accounts for storing $21 \times 6 \times 2 = 252$ RGBA textures per character. This is quite reasonable, considering that competing output-sensitive approaches need to capture the character for each view angle (typically 136 discrete view directions are sampled) and for each animation frame (typically sampled at 10Hz). So for one second of animation, 64×64 textures (which provides a resolution of about 1cm/texel for geometry, colors and normals) and 4 bytes per pixel, it would require about $136 \times 64 \times 64 \times 10 \times 4 = 22$ MB approximately. With our technique each character requires only about 4 MB of storage.

6.1.1 . LOD FOR RELIEF IMPOSTORS

As the characters move away from the camera, we can further speed up rendering by having a hierarchical representation of our relief impostors. We construct a new representation with fewer boxes by merging boxes, i.e. a father node absorbs its child nodes. The OBB associated with the father node is recomputed to include the geometry of the child nodes. New textures are captured for

the new OBBs and for this representation all the geometry included in an OBB will undergo the rigid transformation applied to the father. For instance, if we enclose the hand, fore-arm and upper-arm inside a single OBB, then the hand will not move other than following the upper-arm transformations. Notice that once the user selects the target number of boxes for each LOD and the bones associated with each of them, the task of creating OBBs and capturing textures is fully automatic. For the experiments we used relief impostors with 21, 7, and 1 boxes (see Figure 6.4). The 1-bone LOD has obviously no deformations, which is appropriate only for characters very far away from the camera.

6.1.1 . Real-Time Rendering

Our current prototype uses multiple level-of-detail representations for each character type; a textured polygonal mesh which is used for agents close to the viewpoint, and the multiresolution impostor set described above for the rest of agents. We first render nearby polygonal agents (grouped by character type to minimize rendering state changes) and then the rest of the agents as impostors (again grouped by character type and LOD).

Each character is rendered through an adapted version of relief mapping over the fragments produced by the rasterization of the transformed bounding boxes. The CPU-based part of the rendering algorithm proceeds through the following steps:

1. Bind the corresponding texture arrays (color and normal maps) into different texture units, and bind also the vertex buffer object with the geometry of the bounding boxes in the pose used to capture the impostors. These steps are performed only once per character type.

2. Draw the bounding box associated with each bone, to ensure that a fragment will be created for any viewing ray intersecting the underlying geometry.

The vertex shader multiplies the incoming vertices of the bounding boxes by the corresponding rigid transformation matrix so that they follow the original skeleton animation. The vertex shader also transforms the variables encoding

the location and orientation of each relief map, as these will be used in the fragment shader.

The most relevant part of the rendering relies on the fragment shader, which uses the depth values stored in the color and normal maps to find the intersection P of the fragment's viewing ray with the underlying geometry. For this particular task any ray-heightfield intersection algorithm can be adopted. Pyramidal displacement mapping [Oh et al., 2006] is particularly suitable as it guarantees finding the correct intersection on any heightfield and viewing condition. Our current prototype though is based on the simpler relief mapping algorithm described in [Policarpo et al., 2005].

The fragment shader receives as input the following information:

- World space viewpoint coordinates E.
- World space fragment coordinates C.
- The origin o of the face, i.e. the vertex whose texture coordinates are $(0,0)$.
- An orthonormal basis of the bounding box face, consisting of a normal vector n and two vectors (u,v) aligned along the horizontal and vertical sides of the transformed face.

The fragment shader computes the intersection of the fragment's viewing ray $r = (C - E)$ with the height field encoded by the displacement values stored in the relief map. If no intersection is found, the fragment is discarded. As in [Policarpo et al., 2005], we use first a linear search by sampling the ray r at regular intervals to find a ray sample inside the object, and then a binary search to find the intersection point. This allows us to retrieve the diffuse color of the fragment being processed, along with a normal vector to compute per-fragment lighting. Unlike classic relief mapping, we use two depth values z_f and z_b per texel. During the search process, a sample along the ray with depth z is classified as interior to the object iff $z_f \leq z \leq z_b$.

6.1.2 . Results

We have implemented the construction and rendering algorithms described above in C++ and OpenGL 3.2.

The algorithms have been tested on a collection of detailed human characters from the aXYZ Design's Metropoly 2 data set (Figure 6.3). When converted to Cal3D format, the triangle meshes had 4K to 6K triangles, and used 2048×2048 texture atlases for diffuse color and normal data. All models were initially rigged to 67-bone skeletons. Reported results have been measured on an Intel Core2 Quad Q6600 PC equipped with a GeForce 8800 GT.

Figure 6.3: Test data set. Each mesh contains between 4K and 6K polygons.

6.1.2 . Impostor Creation

The conversion of the input character meshes into a multiresolution collection of relief impostors took on average 859 ms on the test hardware, including all steps detailed in Section 6.1.1. We created three LOD representations with 21, 7 and 1 boxes, respectively (Figure 6.4).

All relief textures (diffuse, normal and depth maps) were captured at 64×64 pixels and stored in a single texture array shared by all the instances of the same character. This resulted in $21 \times 6 \times 64^2 \times 8 = 3.95$ MB for the finest LOD, 1.31 MB for the intermediate LOD, and 192 KB for the coarsest LOD, i.e. about 5.25 MB per character type.

Although our image-based representation is a bit redundant, both within a LOD level (a single surface point is often captured by 1-3 box faces) and across levels (each LOD has its own collection of relief maps), it is still several orders of magnitude more efficient, in terms of memory space, than competing image-based approaches requiring a separate image for each view direction and animation frame.

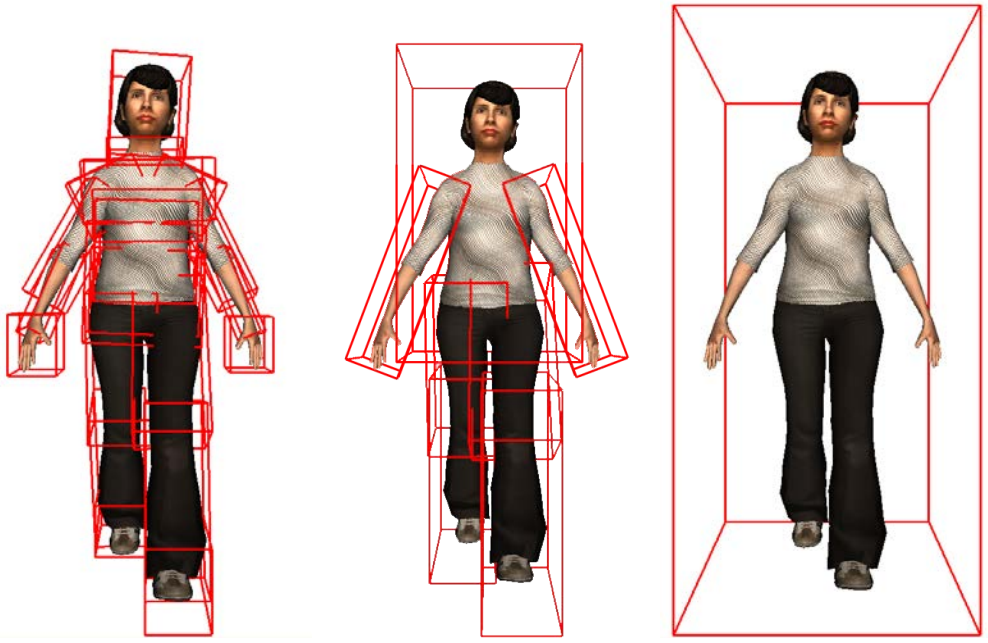


FIGURE 6.4: Relief impostors consisting of 21, 7 and 1 boxes.

(A) (B) (C) (D)

FIGURE 6.5: Artifacts due to lack of geometric skinning: (a) original mesh, (b) impostors created by assigning each triangle to a single joint, (c) impostors created by assigning to each joint all the triangles influenced by the joint, and (d) impostors created by our threshold-based strategy.

Figure 6.6: An animated character rendered using our 21-box relief impostor representation.

6.1.2 . Image Quality

Our relief impostor representation aims at accelerating the rendering of animated characters at the expense of some image quality loss. Image artifacts in the resulting images may fall into the following categories (the surface associated to a particular bone will be referred to as a surface patch):

Surface undersampling due to non-height field patches. We represent each surface patch with six orthogonal relief maps, where each relief map stores a single depth value per texel. Therefore we assume that surface patches look like a height-field when seen from any of these six directions. More formally, we assume that for any axis-aligned ray there is at most one frontface and at most one backface intersection with the surface patch. If

this assumption fails for some axis-aligned direction, the extra intersections between the nearest one at z_n and furthest one at z_f will be ignored, causing the relief mapping algorithm to reconstruct the surface as if the whole segment from z_n and z_f were interior to the object. This could be dealt with by storing multiple depth values per texel, as in [Policarpo and Oliveira, 2006]. Fortunately, our surface patches for the 21-bone skeleton correspond to simple body parts such as upper and lower leg, upper and lower arm, chest, and head, for which the height-field condition above typically holds. Therefore the assumption above produces no artifacts without requiring the storage of additional depth values.

Texel-to-pixel ratio. Since our impostors are image-based, the accuracy of the geometric and appearance details is obviously limited by the texel-to-pixel ratio [Dobbyn et al., 2005]. Therefore we must ensure that textures are large enough to keep the texel-to-pixel ratio above 1:1 for all the viewing distances associated to the textures. Since our 64x64 textures guarantee the above ratio, no image undersampling artifacts appear in the final images.

Depth quantization. The relief mapping algorithm relies on depth values to find the intersection of per-fragment viewing rays with the underlying height-field. We quantize such depth values (which are relative to the box dimensions) using 8-bit integers. Since our boxes have moderate sizes (with the longest edge typically below 50 cm), 8-bit quantization results in (at least) 0.2mm accuracy, which is sufficient to prevent any visible quantization artifact.

Missing ray-surface intersections. Some relief mapping implementations do guarantee that correct ray-surface intersections are always found [Oh et al., 2006, Schroders and Gulik, 2006] whereas others do not [Policarpo et al., 2005]. This issue has been extensively discussed in the literature and can be dealt with in multiple ways (see e.g. [Schroders and Gulik, 2006]).

Lack of geometric skinning. This issue is by far the major limiting factor when considering the valid distance range for our relief impostors. Recall that we animate each relief impostor using the rigid animation of the associated bone. This contrasts with geometric skinning techniques (such as linear blending and dual-quaternion blending) typically applied when animating geometry-based characters, where some vertices are influenced by

more than one bone. In our case, each surface patch is fully influenced by its corresponding bone. This obviously results in some artifacts around joints (triangles influenced by a single bone are reconstructed correctly though). These artifacts might include cracks (e.g. if mesh triangles are assigned to a single bone) or overlapping parts (e.g. if each mesh triangle is assigned to all the bones influencing the triangle, regardless of the weights). Fortunately, our optimized construction results in much less artifacts around joints (Figure 6.5).

Figure 6.6 shows multiple views of one character rendered with our 21-bone impostors. Note that these artifacts are hardly noticeable for moderate viewing distances. Figure 6.7 shows several animation frames of the characters in the test dataset rendered with our 21-bone impostors. Although the images show that artifacts may appear around the joints, these are very hard to perceive in the context of a crowd simulation. Figure 6.8 compares renders using 21-bone and 7-bone representations, respectively. The 1-bone LOD obviously supports no deformations and thus it is reserved for characters very far away from the camera.

One of the features of our approach is the joint handling of geometric and appearance details, encoded through displacement, diffuse and normal maps. The effects of reducing the size of the texture maps is illustrated in Figure 6.9. A side benefit of this approach is that we can use a mipmap pyramid for better minification filtering with no color bleeding artifacts. This is a feature often lacking in polygonal characters, which typically use texture atlases with multiple disconnected patches, thus hindering mipmapping rendering.

6.1.2 . Mesh vs Impostor Rendering

Comparing the performance of our impostors with that of the full-resolution mesh is clearly unfair, as in a real-world application, each character instance would be rendered using an appropriate LOD chosen according to, among other factors, its distance to the viewpoint. We thus compare our approach with a discrete collection of LOD meshes. We use the following notation. LOD representations using *relief impostors* will be denoted as R_j, j being the number of bones/boxes. As stated above, we constructed three representations R_{21}, R_7

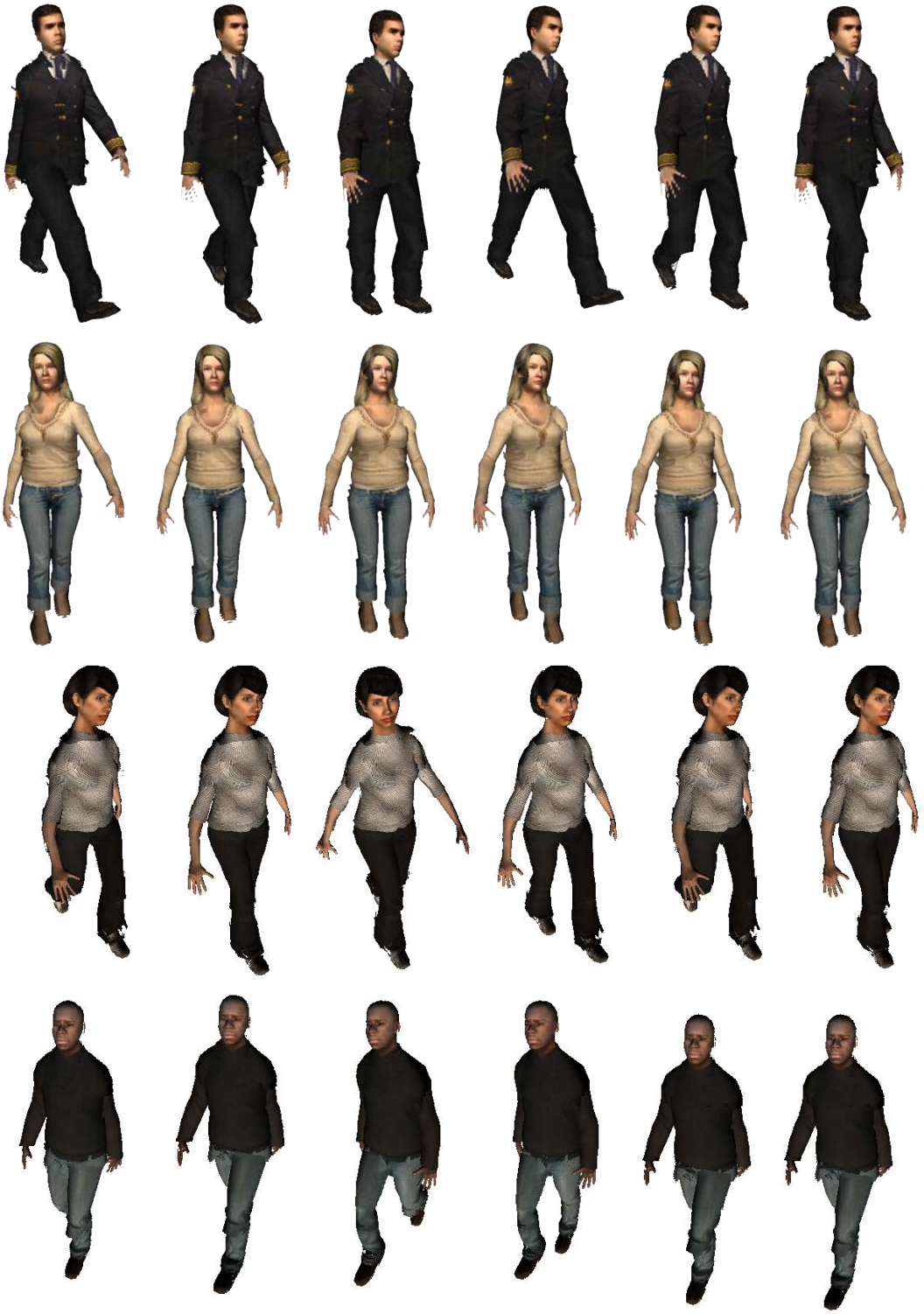


Figure 6.7: Relief impostors corresponding to the test dataset.

and R_1 with 21, 7 and 1 bones, respectively. LOD representations using textured polygonal meshes will be denoted as M_i, i being the percentage of original polygons, M_{100} denoting the full-resolution mesh. We simplified the input mesh to generate LODs M_{90}, M_{85}, $\ldots M_5$ and $M_{2.5}$ (Figure 6.10). Mesh simplification was accomplished using the *Optimize* filter of Autodesk 3DS MAX 2010.

Figure 6.8: Rendering relief impostors with 21 bones (top) and 7 bones (bottom). Note that in the 7-bone representation the head bone has been collapsed with the trunk and thus both undergo the same transformation.

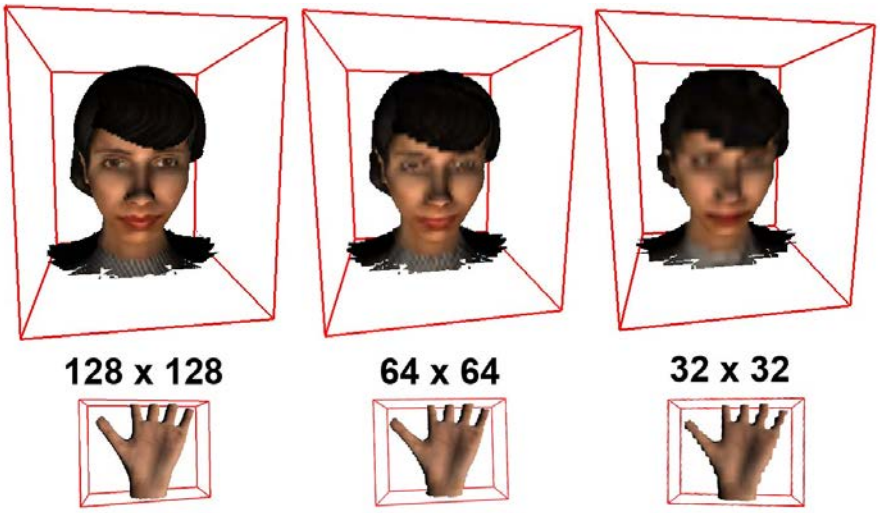


Figure 6.9: Relief impostors with decreasing texture sizes.

FIGURE 6.10: From left to right we show the original mesh M_{100} and some examples of the simplified meshes M_{75}, M_{55}, M_{35}, M_{20} and $M_{2.5}$

We are now interested in a criterion to measure the quality of each representation, which will be used both to compare mesh-based and impostor-based representations, and to select the appropriate LOD according to the distance to the viewer. Note that a measure of the geometric approximation error only makes sense for static polygonal meshes, and that it would ignore visual errors due to distortions of the diffuse and normal maps. We thus adopt an image-space error metric, computed using multiple animation frames and view directions.

Let L be a particular LOD representation of an animated character, using either mesh geometry or relief impostors (i.e. $L \in \{M_{100} \dots M_{2.5}, R_{21}, R_7, R_1\}$). Let $\phi(L,d)$ be the average image difference resulting from rendering a character at distance d using the representation L instead of the full-resolution mesh M_{100}.

screen-projected area or subtended solid angle, provided that we fix some viewing conditions. For all the discussion below, we used a 21' LCD monitor with a 1024×1024 viewport. The field of view of the camera was set to 60 degrees, and the viewing distance from the LCD monitor was set to 60 cm. Note that these typical viewing conditions for a desktop user can be used to determine both the viewing angle subtended by an animated character at some particular

distance from the camera, as well as its screen-projected area. Thus from now on we will refer only to character-to-camera distances.

Since the image difference obviously depends on both the animation frame and the viewing angle, we can compute $\phi(L,d)$ by selecting a representative set of animation frames and a sufficiently dense discretization of the view directions in the Gauss sphere, and averaging the resulting image differences. We chose the root mean square (RMS) error as an objective measure to compare the image differences. We could have adopted perceptual-based image metrics [Yee and Newman, 2004], which integrate factors of the human visual system that reduce the sensitivity to errors, but these metrics are more appropriate for comparing two final, complete images rather than renders of individual agents. This is because high-level HVS models go beyond simple models of brightness and contrast and consider for example masking effects, i.e. decreased visibility of a signal due to background contrast. These effects can be measured only on complete images, where each pixel has a well-defined context. In our case we aim at comparing the rendering of individual characters (thus only a part of the final image) without prior knowledge on the context/background. This is why we discarded HVS-based metrics for comparing the renderings of single, isolated characters, and we chose the broadly-adopted RMS error for this purpose. We thus computed $\phi(L,d)$ by averaging (in the L^2 norm sense) the RMS image differences along 4 equally spaced frames and 10 view directions uniformly distributed on the upper half-sphere surrounding the character. This amounts to 40 samples for computing $\phi(L,d)$, which gives quite reliable results.

Given a certain distance d, we are interested in the simplest mesh level M_i and the simplest relief level R_j the rendering of which produces an image error below some threshold ε. In other words, we want to compute $\min_i\{\phi(M_i,d) < \varepsilon\}$ and $\min_j\{\phi(R_j,d) < \varepsilon\}$. We call the resulting pair M_i, R_j the *optimal* representation for distance d.

We conducted an informal user study to decide a proper error threshold, considering the viewing conditions detailed above. Nine users (aged 23-35) participated in the experiment. Users were presented a video showing an animated character side-by-side, one side rendered using detailed polygonal meshes, and the other side using impostors. Every 10 seconds we doubled the distance from the character to the camera, thus decreasing its screen-projected area. Users

were requested to stop the movie as soon as they perceived no difference between both sides of the image. We recorded the resulting RMS error. We observed that the average RMS error was 0.004, and that none of the users were able to find any difference for a RMS error below 0.003. Therefore we set ε=0.003 to prevent users from perceiving any visual difference between the original and the simplified representation.

The resulting number of primitives for this error threshold are shown in Figure 6.11. Note that triangle mesh rendering requires drawing about one order of magnitude more primitives than impostor rendering, under matching quality conditions. For example, for a character at 15 m, a 4k triangle mesh (8k vertices) is needed to keep the RMS error below ε=0.003, whereas the matching impostor has 126 quads (168 vertices). In terms of per-vertex processing, the polygonal mesh requires about 33k matrix operations for skinning, whereas our impostors requires just 168 operations. For a character at 40 m, the mesh requires 916 matrix operations whereas the impostors only 48. For close-up characters (d < 15 m), the relief-based representation leads to an error above the threshold and thus we fall back to polygonal rendering.

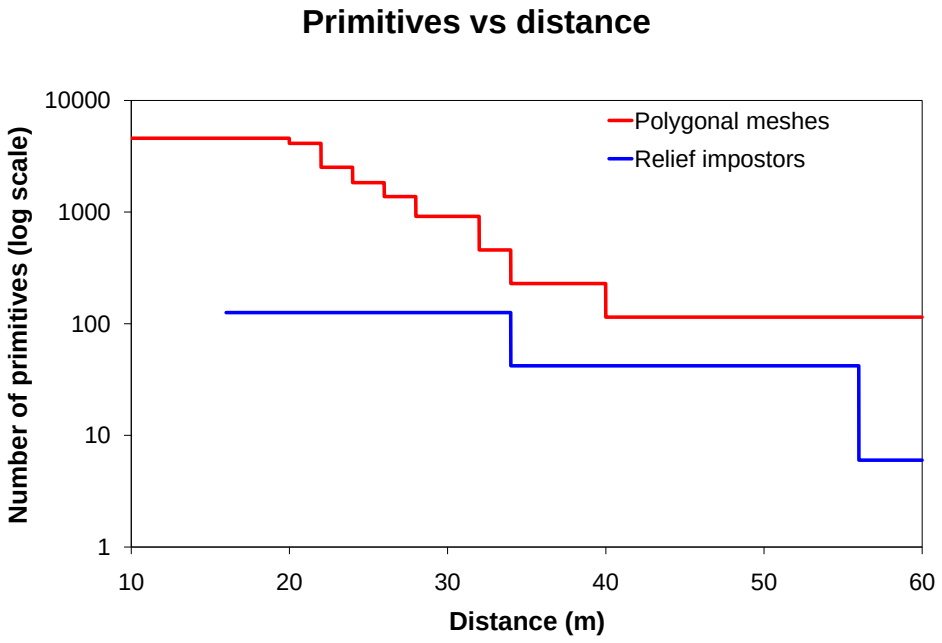


FIGURE 6.11: Minimum number of primitives (triangles/quads) to be drawn (per character instance) to keep the RMS error below 0.003 for one of the test characters. Triangle mesh rendering requires one order of magnitude more primitives (notice the log-scale).

6.1.2 . Choosing the Fastest Representation

Beyond a given distance (15 m for the chosen threshold), we can choose to render the characters using either the optimal mesh-based or the optimal relief-based representation. Since both provide images with similar quality, it makes sense to choose the appropriate representation according to its performance.

For each distance value d, we measured render times for the optimal mesh-based and the optimal relief-based representations computed above. Render times were measured using OpenGL's timer queries, which provide accurate timings. Each query block enclosed the OpenGL drawing commands that need to be executed for each character instance.

For mesh rendering, we used the hardware-accelerated Halca animation library [Spanlang, 2009]. Render times are shown in Figure 6.12. As with image differences, times were averaged for multiple animation frames and view directions, taking 40 samples per distance value.

When rendering polygonal meshes, the bottleneck is likely to be in the vertex processing stage due to the large amount of matrix operations needed to implement skinning. This makes rendering times quite insensitive to the number of fragments produced. However, since for increasing distances we use a more simplified mesh (see Figure 6.11), the rendering times decrease accordingly.

Relief impostors achieve a drastic reduction in the number of primitives to be drawn, and each vertex is influenced by a single bone. This results in a very small number of per-vertex computations when compared to the equivalent level using mesh geometry. On the downside, fragment processing is more involved due to relief mapping computations. As a consequence, rendering times for impostors also decrease with increasing distances, but this time the shape of the resulting curve can be attributed more to the smaller number of fragments rather than to the reduced number of primitives. Note that for characters beyond 15 m, using the relief-based representation results in a performance gain.

6.1.2 . Crowd Rendering Performance

The results above provide optimal switch distances for individual agents, but do not show actual frame rates when rendering a complete crowd. Therefore

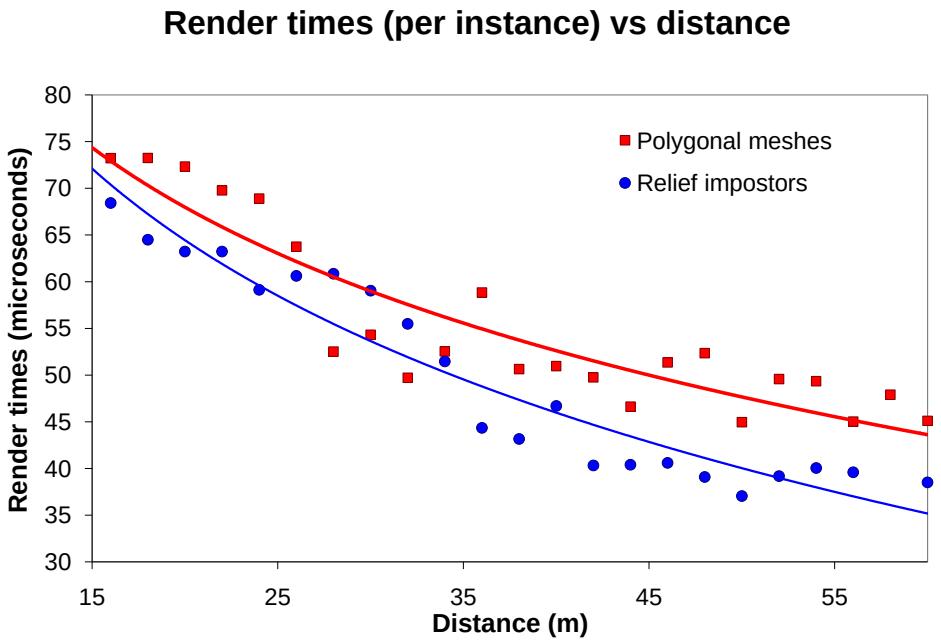


FIGURE 6.12: Render times for the polygonal-based and relief-based LODs that keep the RMS error below 0.003.

we also measured the frame rate using two different strategies for selecting the appropriate LOD: (a) using only mesh-based levels M_{100}, M_{95}, ... $M_{2.5}$, chosen according to Figure 6.11, and (b) using the full-resolution mesh M_{100} or the optimal relief-based level, whichever is fastest. According to the results discussed above, we used the following criteria in option (b) to choose the appropriate representation for each agent: full-resolution mesh for $d < 15$, R_{21} for $15 \leq d < 34$, R_7 for $34 \leq d < 56$ and R_1 for $d > 56$.

Besides the hardware and the number of simulated agents, the actual framerate depends on many factors, including population density (the higher the density, the higher the number of instances requiring fine LOD levels and thus the lower the framerate), camera field of view (the higher the fov, the higher the perspective distortion, thus allowing coarser LOD levels), screen resolution, and number of agents actually visible.

For the following comparison we used a crowd with a varying number of agents rendered into a 1024×600 viewport (see accompanying movies). Table 6.1 shows the resulting frame rates for the different crowd scenarios shown in Figure 6.13. Note that our approach provides a speed up between 2× and 4×.

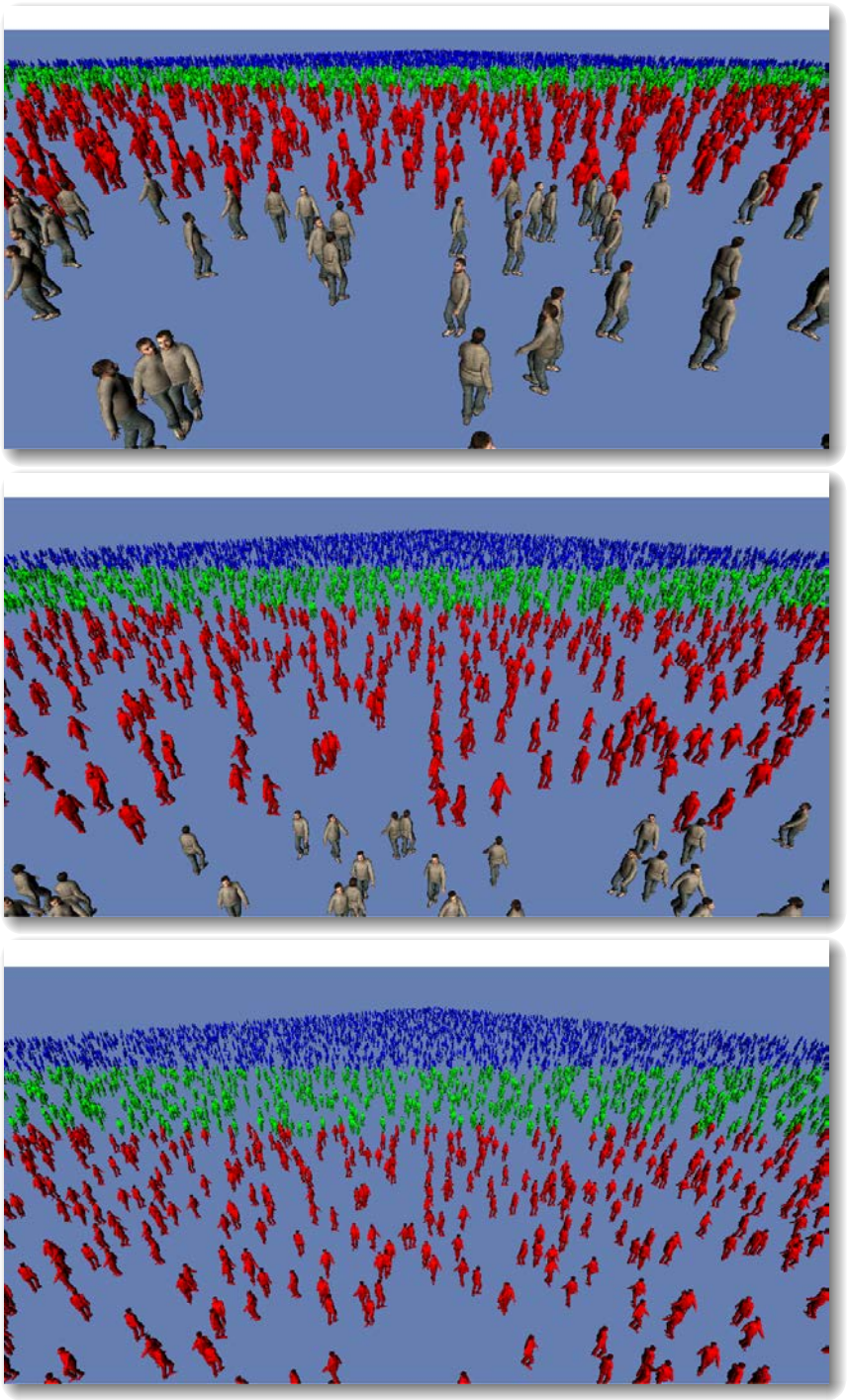


Figure 6.13: Camera settings for the performance test.

Setting	Agents	Polygonal mesh	Our approach	Speed up
Camera 1	2,000	21 fps	50 fps	2.4×
Camera 2	2,000	23 fps	47 fps	2.0×
Camera 3	2,000	25 fps	49 fps	2.0×
Camera 1	4,000	10 fps	38 fps	3.8×
Camera 2	4,000	10 fps	36 fps	3.6×
Camera 3	4,000	12 fps	34 fps	2.8×
Camera 1	10,000	4 fps	16 fps	4.0×
Camera 2	10,000	4 fps	15 fps	3.7×
Camera 3	10,000	4 fps	14 fps	3.5×

TABLE 6.1: Frame rates for different crowds and camera settings.

6.1.2 . USER STUDY

We conducted a user study to validate our impostor-based approach in terms of image quality. The main goal of the experiment was to evaluate whether users perceive any image quality loss when using our impostors instead of polygonal meshes. For this purpose, we rendered an animated crowd with two different strategies: (a) using the full-resolution mesh for all characters, and (b) using for each character the optimal representation (mesh or relief impostors), chosen according to the criteria discussed in Section 6.1.2. We produced a 25 s movie for different crowd settings (Figure 6.13). We used exactly the viewing conditions detailed in Section 6.1.2. In order to assess image quality with respect to the reference image, we grouped these movies in pairs, stacking horizontally the movie using impostors with the one using full-resolution meshes. The movie using impostors was stacked on the left/right randomly.

Nine subjects (aged 23-35) participated in the experiment. Users were requested to watch five pairs of videos (which were presented in a random order) and to decide which of the two sides (left/right) had better image quality than the other, if any. This yield a total of $N = 45$ trials. Let f_i be the (relative) frequency of users choosing the side with impostors as the best movie. Likewise, let f_g be the frequency of users choosing the side with geometry, and f_u the frequency of users unable to decide which side is better. The absolute frequencies we observed from the 45 samples of our user study where $n_i = 15$, $n_g = 14$, and $n_u = 16$. We consider the null hypothesis to be 'giving the answers by chance' which implies that all conditions should be chosen with equal probability, i.e. $H_0 : f_i = f_g = f_u = 1/3$. The corresponding significance levels for a two-sided test against the null hypothesis that each proportion is 1/3 are 0.5, 0.7, 0.8,

therefore the null hypothesis cannot be rejected. This means that the choices of the subjects were equivalent to random choices, and thus our impostor-based technique can be used for rendering acceleration with negligible visual artifacts (see Figure 6.14).

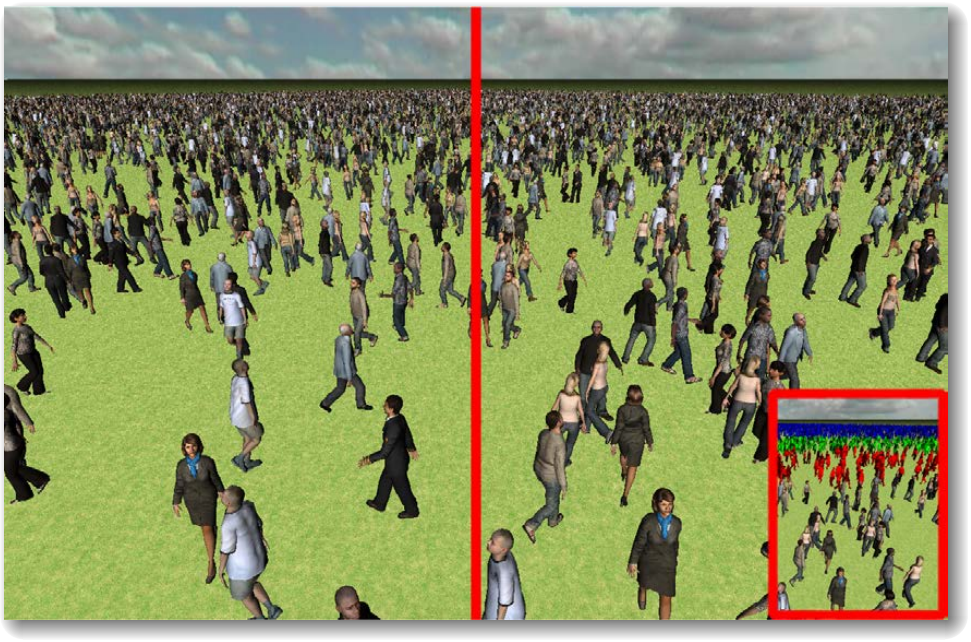


Figure 6.14: Image rendered with full-resolution polygonal meshes (left) and our approach (right).

6.2 . Flat Per-Joint Impostors

Our previous approach creates separate impostors for each body part. Each character is encoded as a collection of bounding boxes where each box represents the geometry influenced by each joint through six orthogonal displacement maps. At runtime, an adapted version of relief mapping is used to render the displaced geometry. Although this method provides a high-quality representation for distant meshes, its applicability is limited to relatively far-away characters due to the high per-fragment cost of the relief mapping shader.

We aim at encompassing the performance benefits of view-dependent impostors with the flexibility of animation-independent approaches. On the one hand, view-dependent impostors minimize the geometry to be transformed as well as per-fragment computations, and thus achieve the maximum performance while minimizing the use of programmable hardware. On the other hand, animation-independent approaches are more flexible in terms of animation clips and animation blending.

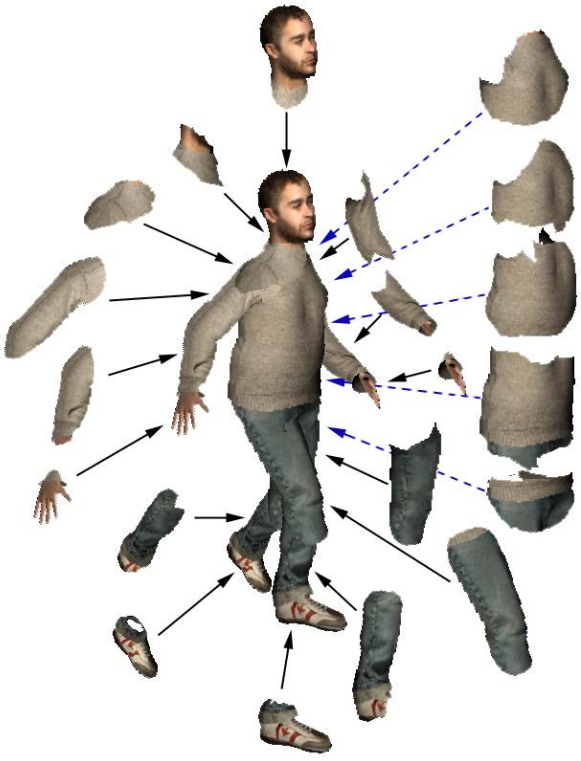


Figure 6.15: We combine per-joint impostors to render animated characters.

Our approach thus relies on view-dependent impostors but instead of having each impostor represent the whole character at a given pose from a given view direction, we use a different view-dependent impostor *for each joint* (see Figure 6.15). During rendering, each impostor undergoes the same rigid transformation as its corresponding joint. This way we do not need to generate nor store impostors for every animation frame, and thus our approach offers the flexibility of being able to add on the fly new animation clips.

We provide algorithms for creating automatically the most appropriate impostors during preprocessing as well as algorithms for their efficient rendering. Since our impostors are intended to be valid for any pose, a key issue is to properly define which part of the geometry influenced by each joint must be represented as opaque pixels in the corresponding impostor. The opaque portion of the impostor will be referred to as *mask*. We provide an efficient algorithm for computing optimized masks which considers how the geometry of each bone is affected by the transformation of neighboring joints. Our approach clearly outperforms competing animation-independent approaches for crowd rendering.

6.2.1 . Overview

We aim at increasing the number of simulated agents in real-time crowd simulations by reducing the rendering cost of individual agents. Therefore only characters that are very close to the observer are rendered as polygonal meshes, while the rest of the agents are rendered using our new per-joint impostor method.

Figure 6.16 outlines the main stages of our impostor construction algorithm. We assume the input character conforms to the de facto standard in character animation and thus consists of a textured polygonal mesh (skin), a hierarchical set of bones (skeleton) and vertex weights. The nodes of the skeleton represent joints and the edges represent the bones. Since each bone can be easily identified by its origin, we can use the terms bone and joint interchangeably. The transformations affecting joints in the hierarchy are assumed to be rotations. The vertex weights describe the amount of influence that each joint has over each vertex.

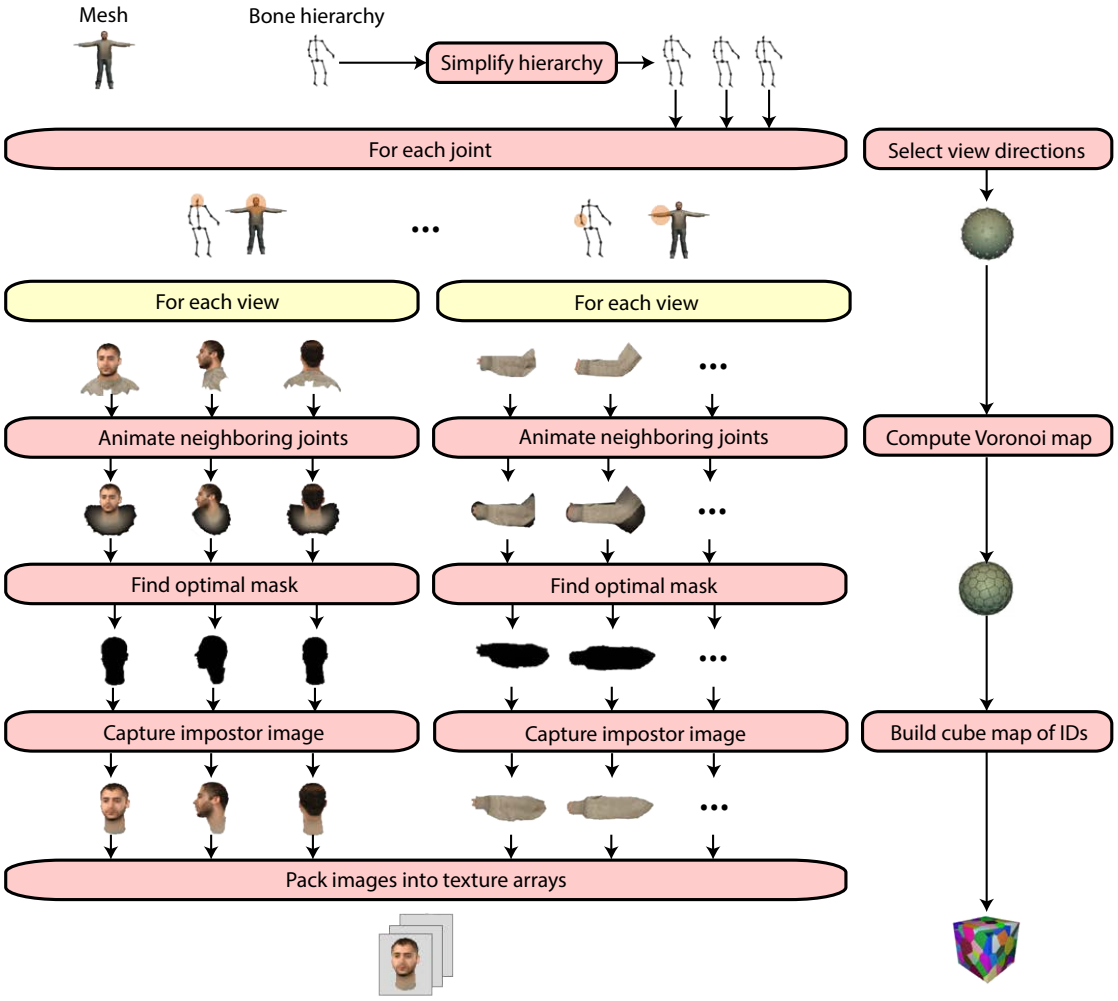


FIGURE 6.16: Overview of our algorithm for generating per-joint impostors

Since we want to support real-time blending of animation sequences, we create a separate impostor for each animated part of the articulated character (Figure 6.15). In this respect, our approach is similar to our previous one. However, instead of using six orthogonal relief maps for each joint, which requires multiple dependent texture accesses per fragment, we use flat impostors created by sampling each joint from multiple view directions. This results in a single texture lookup per fragment, which is one order of magnitude faster than relief mapping.

Our representation for distant characters consists of a collection of impostors (alpha-matted textured quads), per joint and view direction. At runtime, the impostors are transformed in the same way as the bones of the skeleton, giving the impression that our impostor character is animated.

The quality of our representation depends on four major factors, namely (a) the

resolution of the impostor images, (b) the number of view samples (i.e. impostors) per joint, (c) the distribution of the view samples on the Gauss sphere S^2, and (d) the portion of the polygonal mesh influenced by the joint represented by each impostor.

Factors (a) and (b) clearly control the trade-off between memory and quality of the representation, and thus we consider texture resolution and number of view samples as user-defined parameters. Concerning the distribution of view samples on S^2, both uniform and adaptive sampling schemes can be adopted. The pros and cons of both approaches will be discussed below.

According to our experiments though, the portion of the polygonal mesh represented by each impostor is by far the most crucial factor for minimizing animation artifacts. Since our per-joint impostors are intended to be valid for any arbitrary pose, they must be defined so as to blend with neighboring impostors as seamlessly as possible. Failing to define a proper mask (the *mask* is the opaque portion of the impostor, see examples in Figure 6.16)

would result in animations showing visible cracks around joints or duplicated geometry appearing as protruding parts. We provide a formal statement for this problem, and present an efficient algorithm for computing the impostor masks.

6.2.2 . Preprocessing

The construction of per-joint impostors from a given 3D character proceeds through the following steps, described in detail below (see Figure 6.16):

1. Simplify the bone hierarchy, creating multiple level-of-detail skeletons.
2. Choose a suitable set of samples from S^2, compute the spherical Voronoi map of these samples, and build a cube map by projecting the Voronoi cells onto the cube faces.
3. For each joint and for each view, compute a proper mask and capture the impostor.
4. Pack all textures in a texture array of texture atlases.

6.2.2 . Bone Hierarchy Simplification

Since we use per-joint rather than per-character impostors, we simplify the input bone hierarchy by grouping joints, letting some parent nodes absorb small child nodes.

The resulting impostors will obviously undergo the rigid transformation applied to the parent node. For instance, if we group the hand, fore-arm and upper-arm into a single joint, the hand will not move other than following the upper-arm rotations. For the experiments we created the same bone hierarchies as in the previous approach, with 21, 7, and 1 joint (see Figure 6.4). The 1-bone LOD supports no deformations and thus cannot be animated, which makes it appropriate only for characters very far away from the camera.

6.2.2 . Choosing View Samples From S^2

Our approach supports any strategy for sampling views from the unit sphere S^2. We considered both uniform and non-uniform (i.e. adaptive) sampling schemes. The main advantage of the latter, in the context of crowd rendering, is that we can sample S^2 more densely around views more likely to occur when animating the character, and around views capturing the more salient parts of the character. For example, views around the south pole of S^2 are unlikely to be needed when rendering a crowd, and thus can be sampled more sparsely. Likewise, we might want to sample front and side views of the head more densely than top and back views [McDonnell et al., 2009].

Since memory requirements is not an important concern for our approach (as we do not need to capture impostors *for each pose*), our current implementation samples S^2 using a uniform sampling strategy. For the sake of simplicity, we subdivide once an icosahedron and take its face normals as the view samples. The subdivided triangles are taken as the Voronoi regions defined by these samples, see Figure 6.17.

At runtime, we need to solve a nearest-neighbor problem for associating each element in S^2 with one of the discrete samples. Since this computation has to be performed for each joint of each agent in the crowd, we build a cube map by projecting the color-coded subdivided triangles onto the faces of a unit

(A) (B) (C)

FIGURE 6.17: Choosing samples from S^2: (a) subdivided icosahedron; triangle normals correspond to view directions for capturing the impostors, (b) spherical Voronoi regions defined by the triangle normals, and (c) cube map build by casting the triangles onto a cube. The cube map is used to find the nearest precomputed view for an arbitrary direction in S^2.

cube, Figure 6.17(c). Since the texel of the cube map representing a direction ω stores the ID of its nearest discrete sample, the nearest-neighbor problem is solved using a single texture lookup. As we use a uniform sampling scheme for all joints, a single cube map suffices for all joints.

6.2.2 . IMPOSTOR GENERATION

Problem statement

Let B be a given joint from the bone hierarchy, and let $N_1,\dots N_j$ be its neighboring joints. Each joint is controlled by a rotation matrix which changes during animation according to the character's pose. Suppose that we need to generate an impostor for the part of the mesh influenced by joint B. Since rotations are invertible, we can fix joint B (as if it were the root node) and represent the rotation matrices of its neighboring joints as relative to B.

Let T be the set of triangles influenced by B, i.e. triangles with at least one vertex being influenced by B, see Figure 6.18a. Since mesh triangles can be influenced by multiple joints, the final geometry of the triangles in T is determined by the rotation matrices attached to its neighboring joints (along with the procedure for implementing mesh skinning). Given a joint B and a collection of rotation matrices, the result of deforming the triangles in T according to these matrices is called a *realization* of B, and it will be denoted as R.

Let ω be some view sample from S^2. We can orthogonally project all the possible realizations of B onto an image plane Π perpendicular to ω. The portion

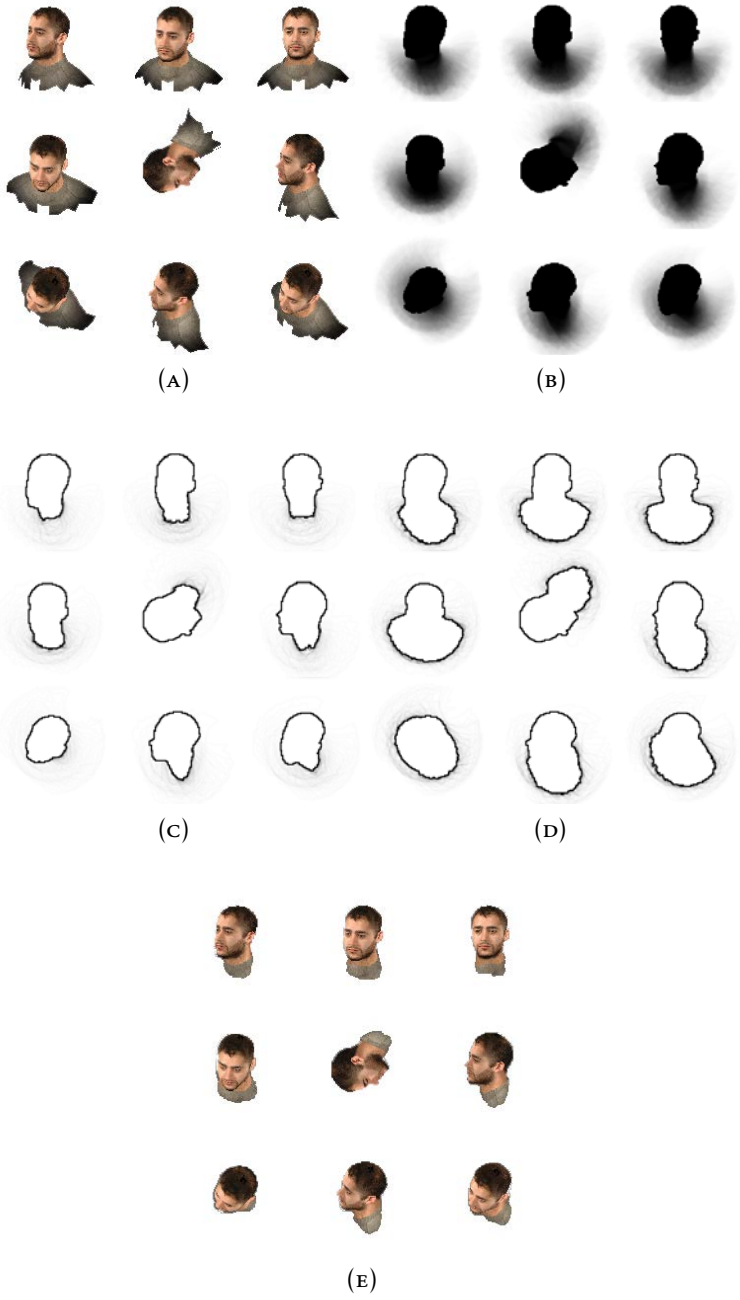


FIGURE 6.18: Defining impostor extents for multiple view directions: (a) triangles T influenced by the head bone, shown in a reference pose; (b) area swept by T when neighboring joints are rotated; the gray level indicates the number of realizations projecting onto each pixel; (c) and (d) two possible mask boundaries defining the extents of the impostors; (e) impostor captured using our optimized mask.

of the image plane Π filled by these projections will be referred to as the *swept area* S, see Figure 6.18b. Now the key problem is to decide which portion of S must be captured by the impostor representing B from ω.

For the sake of simplicity, let us assume that the geometry attached to a given joint forms a single connected component, and that the portion to be included in the impostor image (i.e. the impostor mask M) is simply connected and thus has a single outer boundary (the discrete heuristic algorithm proposed below handles the general case of multiple connected components with arbitrary genus).

Let $u : [0,1] \rightarrow \mathbb{R}^2$ be the parameterization of a closed loop on the image plane Π, defining the boundary of the impostor mask. Some sample mask boundaries are shown in Figures 6.18c and 6.18d.

For each bone B and for a collection of S^2 samples, we could generate the corresponding impostor using some collection of mask boundaries u, see Figure 6.18e. The animated characters can be rendered as a collection of rigidly-transformed impostors, choosing for each bone B the precomputed impostor whose view better matches its final orientation. The resulting image might exhibit some error E when compared to that produced by rendering the original polygonal mesh. The image error depends on multiple factors, including the actual camera position and orientation with respect to the character, the actual pose, and the criterion adopted to generate the mask boundaries. We pay special attention to the latter, as mask boundaries play a key role on the final image quality and it is a factor subject to optimization during preprocessing.

In order to better illustrate the effect of the criterion for generating the mask boundary, let us consider first two extreme cases. The *undercoverage* criterion would define the mask as the region containing just the triangles uniquely influenced by B (i.e. triangles with vertices having unit weight for this joint, and null weight for the rest of joints). It turns out that this criterion would tend to produce visible gaps during animation, in particular around joint boundaries, as the corresponding blending triangles wound not be represented in any impostor. See Figures 6.21a and 6.21b.

The opposite *overcoverage* criterion would define the mask as the swept area S. This obviously would result in overly large, overlapping impostors, causing protruding geometry artifacts around joint boundaries.

Therefore the problem can be stated as the following optimization problem over the set U of all mask boundaries:

$$\min_{u \in U} f(u) = \int_{S^2} \int_{P} E^2(\omega, p, u) d\omega \, dp$$

where u here is the collection of mask boundaries, ω represents view directions, $p \in P$ represents poses in a space of poses, and E is a measure of the error produced by rendering, instead of the polygonal mesh, the impostors masked with u at pose p from a camera aligned with ω.

In other words, we aim at choosing the mask boundaries minimizing the image error across all possible views and animations frames. Note that, even if a basic error metric is used for computing E, the variational problem above cannot be solved analytically. In the next section we provide an efficient GPU-based algorithm for computing an approximate solution to this problem.

Mask generation and impostor capture

For each joint B and view v, we need to generate a mask defining the portion of the swept area S that must be captured by the corresponding impostor. We can restrict ourselves to mask boundaries within the swept area S (which can be seen as the *union* of the orthogonally-projected realizations) because points outside S do not belong to the projection of the joint's geometry, no matter the character pose, and thus should never belong to the mask. Another key observation is that mask boundaries should enclose at least the triangles influenced exclusively by the joint. These triangles belong to all possible projections of the joint's geometry, and thus their inclusion causes no protruding geometry and excluding them would result in visible gaps.

A key contribution of our approach is to define the mask as the *intersection* of the orthogonally-projected realizations of the joint. Since points within such a mask belong to all possible projections of the geometry influenced by the joint, this is likely to be the largest mask we can generate while guaranteeing no protruding parts, regardless of the character pose. The resulting masks exhibit rounded boundaries near joints due to the blending effect of neighboring joint rotations on the deformed geometry. These shapes are somewhat similar to the rounded joints used in traditional art mannequins and articulated figures, see Figure 6.18e.

In order to obtain rounded endings for each mask, we obtain the realization of a joint by rotating each DoF within the range $[-\pi/4, \pi/4]$ with respect to a standing reference pose. We could obtain masks closer to the range of movements of each joint, if we had in advance the skeleton information regarding the DoFs available per joint, and their real limits. However, since we aim at producing animation-independent impostors, our implementation uses a fixed set of angle limits.

We did not attempt to compute the exact intersection of the projected realizations analytically, although for linear blend skinning this should not be a difficult task. Instead, we use an approximate but simpler algorithm based on rendering a discrete subset of realizations by sampling rotation angles within the per-joint limits at equally-spaced intervals.

Each realization is rendered onto a frame buffer object (FBO) which plays the role of an accumulation buffer (Figure 6.18b). For a given joint B and view ω, the realizations are generated by applying linear blend skinning to the set of triangles influenced by B (see Algorithm 1 below).

Algorithm 2 Generate impostors

```
for each joint B do
    for each view ω do
        compute the set T of triangles influenced by B
        clear FBO
        for each neighboring joint N_j of B do
            for each free axis a⃗_jk of N_j do
                for each discrete angle α_jk in [−π/4, π/4] do
                    deform T using the rotation angles α_jk
                    render T using an orthographic camera and accumulate into FBO
                end for
            end for
        end for
        mask M = set of pixels in the FBO with maximum accumulated value
        capture impostor using mask M
    end for
end for
```

The algorithm above clearly runs in $O(bvna)$ time, where b is the number of joints in the skeleton (21+7 in our experiments), v is the number of views (80 in our implementation), n is the number of neighboring joints (typically less than 4), and a is the number of samples in the $[-\pi/4, \pi/4]$ range (we use 10 samples). Since the number of joint DoFs is at most three, in the worst case scenario we need to render about $28 \cdot 80 \cdot 4 \cdot 3 \cdot 10 = 268\,\mathrm{K}$ joints. Assuming

each joint can be rendered in 1ms, the full impostor set for a character can be generated in about 5 minutes.

Since our algorithm only considers a discrete set of rotation angles, the resulting masks are slightly smaller than the true intersection. We thus create a slightly enlarged mask that encloses all the pixels in the swept area that belong to a large percentage r of realizations. We have found empirically that using $r = 80\%$ produces appropriate results. Therefore we modify Line 13 in the algorithm above to compute the mask M as the set of pixels in the FBO whose normalized accumulated value is greater than 0.8. Although most body parts and views result in masks with a single genus-0 component, the algorithm handles equally well masks with arbitrary topology.

6.2.3 . Real-Time Rendering

Our current prototype uses multiple level-of-detail representations for each character type; a textured polygonal mesh which is used for agents close to the viewpoint, and the multiresolution impostor set described above for the rest of agents. We first render nearby polygonal agents (grouped by character type to minimize rendering state changes) and then the rest of the agents as impostors (again grouped by character type and LOD). Impostor characters are rendered through the following algorithm.

6.2.3 . CPU Stage

The CPU-based part of the algorithm proceeds through the following two steps:

1. Bind the corresponding texture arrays (color and optionally normal maps) and the cube map texture that maps directions in S^2 to precomputed view samples. This step is performed only once per character type, not per instance.

2. Draw a vertex buffer object (VBO) encoding each character type (all instances of a character type share the same VBO).

The VBO contains one vertex for each per-joint impostor, corresponding to the center of its bounding cube. The rest of the impostor information is encoded as vertex attributes. These attributes include the joint ID, the orthonormal basis of the impostor quad (in the capture pose), and the layer of the texture array containing the images for the joint. In our implementation the VBO is rendered as GL_POINTS primitives that will be later converted into a pair of triangles. The rationale of this approach is to avoid duplicating per-joint computations.

6.2.3 . Vertex Shader

The vertex shader transforms the vertex itself and its local orthonormal basis according to the character's pose, so that the joint follows the original skeleton animation, and computes the discrete view that best matches the joint orientation. This last step is completed with a single look-up at the cube map. Since the vertex shader is executed only once per-joint, we avoid duplicating these computations.

6.2.3 . Geometry Shader

The geometry shader is executed for each GL_POINT primitive encoding a (transformed) per-joint impostor. It simply creates a couple of triangles defining the impostor quad, using the transformed orthonormal basis encoded as vertex attributes. It also computes the (s,t,p) texture coordinates for accessing the texture array.

6.2.3 . Fragment Shader

The fragment shader performs traditional texture mapping (and per-fragment lighting, if normal maps are available) to compute the final fragment's color. Traditional approaches based on flat impostors handle visibility among intersecting characters using per-texel depth values [Aubel et al., 2000, Maim et al., 2009a]. However, since we use separate impostors for each joint, each impostor has a different depth value (see Figure 6.19), which makes this problem less important than when using a single impostor for the whole character. We only need to handle self-intersections due to overlapping parts around joints. We

take care of this by using a similar technique, with a fragment shader that adds an offset to the quad's depth values, but the offset is computed using a per-texel weight representing the (interpolated) weight of the joint over the texel. This causes fragments from the most influencing joint to have visibility priority over other overlapping joints.

FIGURE 6.19: The impostors on the left were used to render the character from the upper-right camera. The whole scene is shown from an exocentric view to better illustrate the impostors being drawn. When projected according to the upper-right camera, the impostors blend into the framed image.

6.2.4 . Results

6.2.4 . Implementation details

We have implemented the construction and rendering algorithms described above in C++ and OpenGL 3.2. Our implementation relies on the Halca animation library [Spanlang, 2009] to draw the animated characters from which we create our impostors. Halca is a hardware-accelerated library for character animation which is based on the Cal3D format [Cal3D, 2014].

The algorithms have been tested on a collection of detailed human characters from the aXYZ Design's Metropoly 2 data set (Figure 6.20). When converted to Cal3D format, the triangle meshes had 4K to 6K triangles, and used 2048×2048 texture atlases for diffuse color and normal data. All models were initially rigged to 67-bone skeletons. Reported results have been measured on an Intel Core2 Quad Q6600 PC equipped with a GeForce GTX 280.

Figure 6.20: Test data set. Each mesh contains between 4K and 6K polygons.

The conversion of the input character meshes into a multiresolution collection of per-joint impostors took on average 30 minutes on the test hardware. We created three LOD representations with 21, 7 and 1 boxes, respectively (Figure 6.4).

All textures were captured at 32×32 pixels and stored in a single texture array shared by all the instances of the same character. This resulted in $21 \times 80 \times 32^2 \times 3$ bytes = 4.9 MB for the finest LOD, 1.31 MB for the intermediate LOD, and 192 KB for the coarsest LOD, i.e. about 5.25 MB per character type.

Although our image-based representation is a clearly redundant, both within a LOD level (a single surface point is often captured from multiple views) and across levels (each LOD has its own collection of per-joint impostors), it is still more efficient, in terms of memory space, than competing image-based

approaches requiring a separate image for each view direction *and animation frame*.

6.2.4 . Illustrative Results

Our impostor representation aims at accelerating the rendering of animated characters at the expense of some image quality loss. Image artifacts in the resulting images may fall into the following categories:

Texel-to-pixel ratio. Since our impostors are image-based, the accuracy of the geometric and appearance details is obviously limited by the texel-to-pixel ratio [Dobbyn et al., 2005]. Therefore we must ensure that textures are large enough to keep the texel-to-pixel ratio above 1:1 for all the viewing distances associated to the textures. Since our 32x32 textures guarantee the above ratio, no image undersampling artifacts appear in the final images.

Lack of geometric skinning. Recall that we animate each impostor using the rigid animation of the associated bone. This contrasts with geometric skinning techniques (such as linear blending and dual-quaternion blending) typically applied when animating geometry-based characters, where some vertices are influenced by more than one bone. This results in some artifacts around joints. These artifacts might include cracks or overlapping parts. Fortunately, our optimized construction results in less artifacts around joints than naive approaches (Figure 6.21) and can be neglected at the distance range our impostors are used.

Figure 6.22 shows multiple views of one character rendered with our 21-bone impostors. Note that these artifacts are hardly noticeable for moderate viewing distances.

Figure 6.23 compares renders using 21-bone and 7-bone representations, respectively. The 1-bone LOD obviously supports no deformations and thus it is reserved for characters very far away from the camera.

Figure 6.24 shows several animation frames of the characters in the test dataset rendered with our 21-bone impostors.

(A) (B) (C) (D)

FIGURE 6.21: Importance of properly defined masks. The two characters on the left were rendered using impostors created by an undercoverage criterion, showing missing parts. The two characters on the right were created using our algorithm.

Although the images show that artifacts may appear around the joints, these are very hard to perceive in the context of a crowd simulation. Figure 6.25 shows a crowd rendered using our approach.

Figure 6.26 compares polygonal mesh rendering against our approach, and Figure 6.27 compares polygonal rendering, relief maps and per-joint impostors.

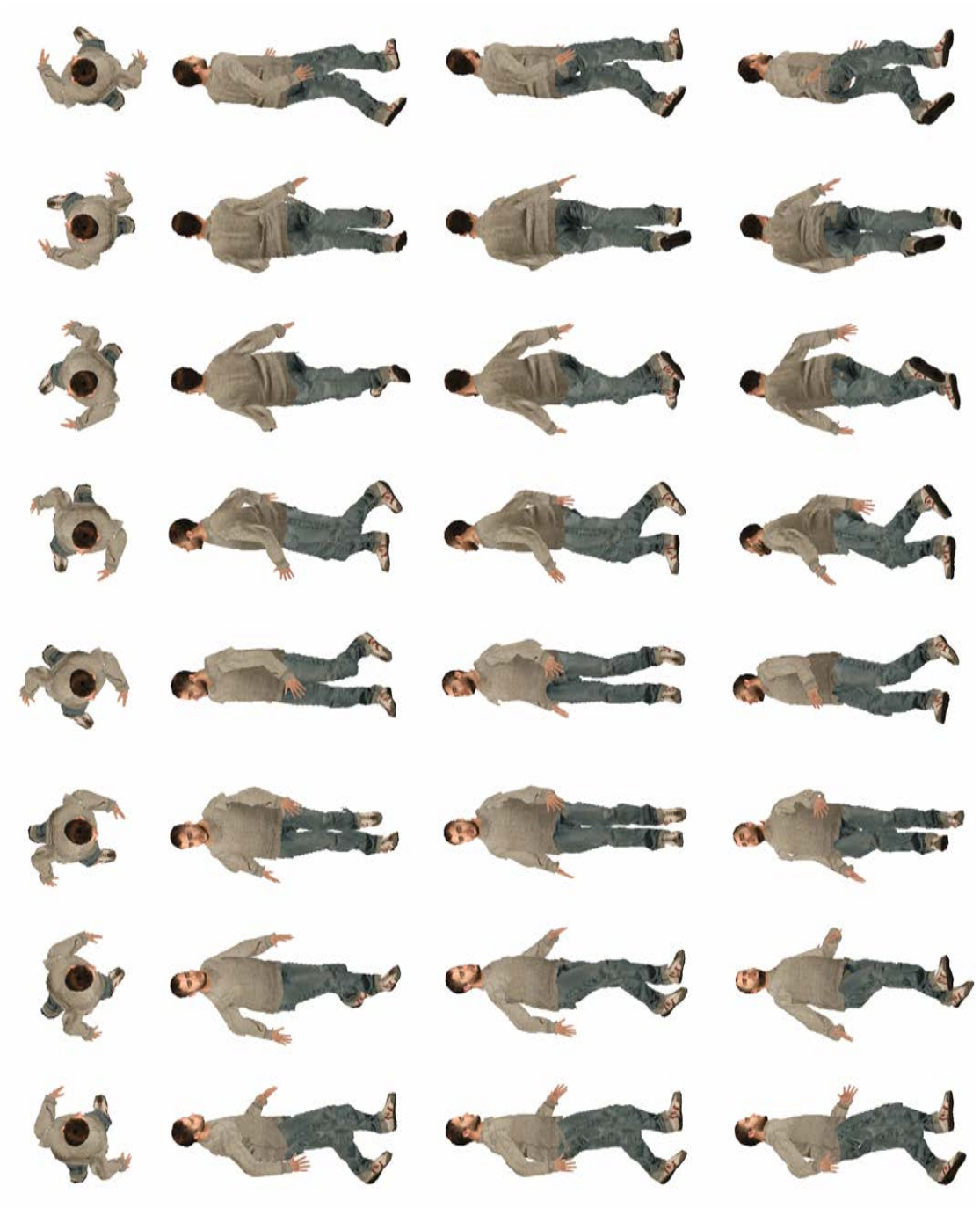


FIGURE 6.22: An animated character rendered using our 21-joint representation.

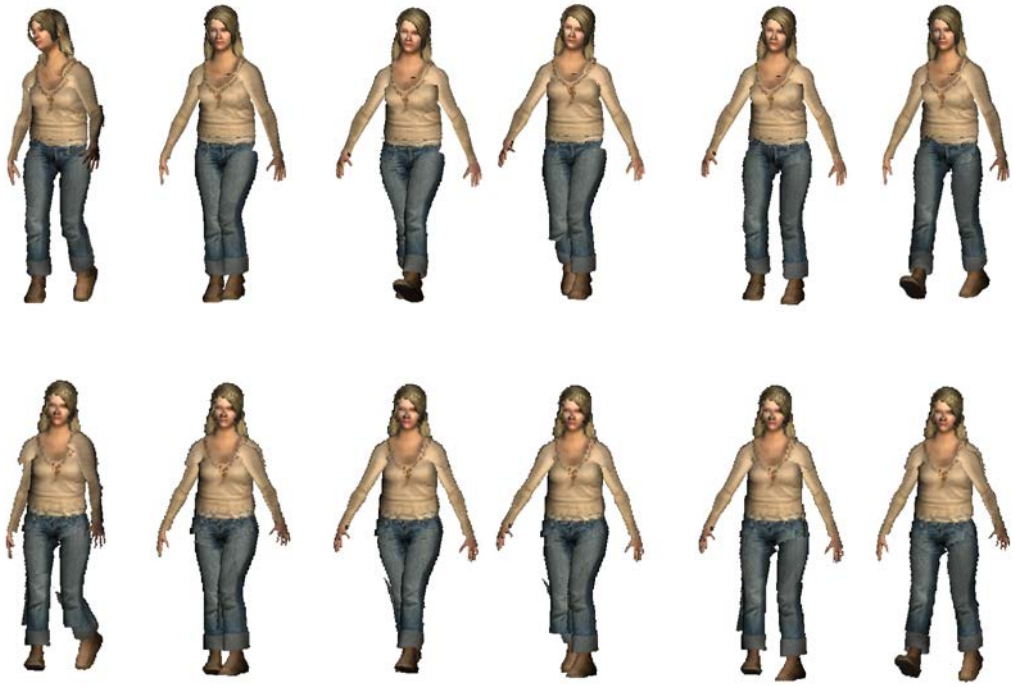


FIGURE 6.23: Rendering per-joint impostors with 21 bones (top) and 7 bones (bottom). Note that in the 7-bone representation the head bone has been collapsed with the trunk and thus both undergo the same transformation.

FIGURE 6.24: Characters rendered with our per-joint impostors.

FIGURE 6.25: Crowd with about 8,000 agents rendered with our per-joint impostors.

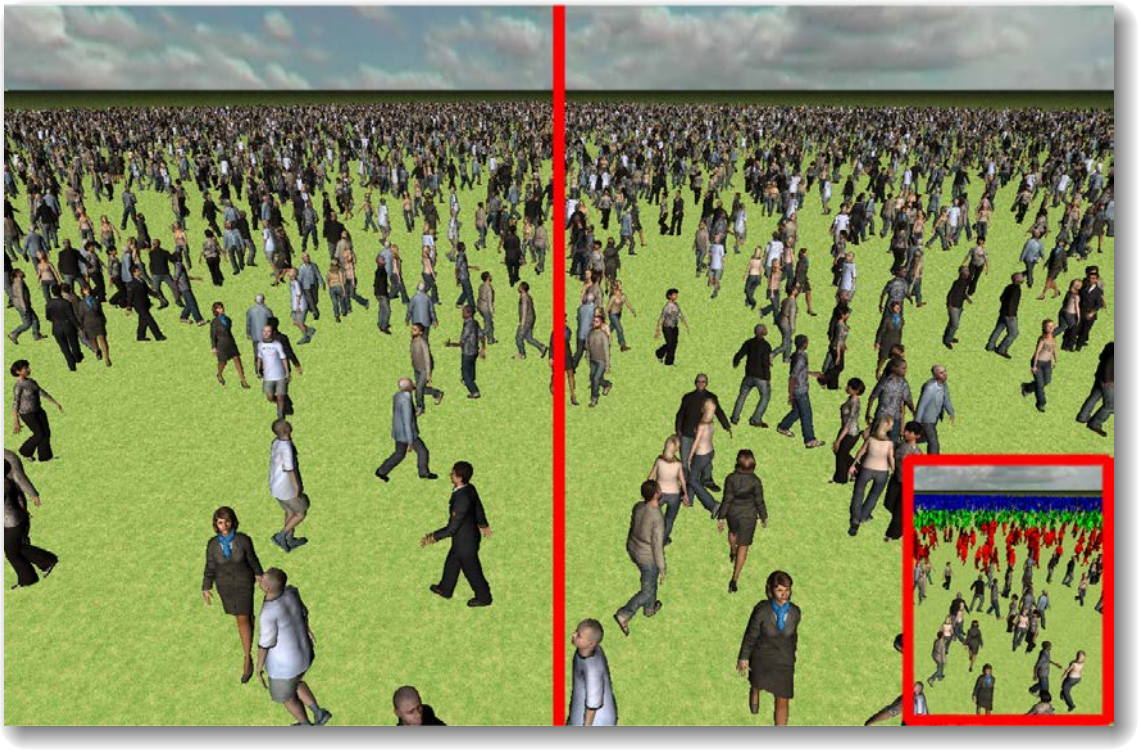


FIGURE 6.26: Image rendered with polygonal meshes (left) and our approach (right). The color in the inset image indicates the diferent LoD levels.

FIGURE 6.27: Comparison of rendering with geometry (left), relief impostors (middle) and our per-joint impostors (right).

6.2.4 . Performance

Comparing the performance of impostors with full-resolution meshes is clearly unfair, as in a real-world application each character instance would be rendered using an appropriate LoD chosen according to, among other factors, its distance to the camera. We thus compared the frame rate during crowd rendering using three different primitives: polygonal meshes, relief impostors [Beacco et al., 2011] and our per-joint impostors. Note that these three techniques support arbitrary poses. For each of these techniques, we measured rendering times both using a single LoD for all the agents (the highest resolution mesh, relief impostor or per-joint impostors) or multiple LoDs according to the agent distance. When using multiple LoDs, close-up agents were always rendered as fully-detailed polygonal meshes, whereas more distant agents were rendered as simplified meshes, relief impostors or per-joint impostor, depending on the technique. Mesh simplification was accomplished using the *Optimize* filter of Autodesk 3DS MAX 2010, which created LoD meshes with 90%, 85%, ...5% and 2.5% of the original triangles (Figure 6.28). Switch distances for the different LoDs were set according to matching RMS error, following [Beacco et al., 2011]. The distance for switching from polygonal meshes to relief impostors was set to 15 meters (relief maps do not provide performance gains for nearby agents due to the per-fragment overhead); for per-joint impostor, the distance was set to 5 meters. We used viewing conditions similar to those in our previous contribution [Beacco et al., 2011]. The bone hierarchies for relief and per-joint impostors had 21, 7 and 1 bone, respectively. For polygonal meshes we used the original bone hierarchy as their rendering cost was found to be dependent mainly on the number of triangles.

Besides the hardware and the number of simulated agents, the actual framerate depends on many factors, including population density (the higher the density, the higher the number of instances requiring fine LoD levels and thus the lower the framerate), camera field of view (the higher the fov, the higher the perspective distortion, thus allowing coarser LoD levels), screen resolution, and number of agents actually visible.

Table 6.2 shows the resulting frame rates for different animated crowd scenarios from a fixed camera position (aerial or street-level view) on a 2048×1536 viewport, using geometric instancing [Dudash, 2007a]. Per-joint impostors are

FIGURE 6.28: From left to right we show the original mesh and some examples of the simplified meshes used for comparison with our approach. Simplified meshes have 75%, 55%, 35%, 20% and 2.5% of the original triangles.

clearly the fastest representation. When using a single representation for all the agents, per-joint impostors are between 5× and 6.3× faster than relief impostors, and between 5× and 8.3× faster than polygonal meshes. When using multiple LoD representations, which better matches a real usage scenario, per-joint impostors are 2.1× - 3.1× faster than relief impostors, and 4.5× - 8.2× faster than LoD meshes, thus providing significant performance gains.

When rendering polygonal meshes, the main performance bottleneck is likely to be in the vertex processing stage due to the large amount of matrix operations needed to implement skinning. Relief impostors are faster than polygonal meshes as they achieve a drastic reduction in the number of primitives to be drawn. This results in a very small number of per-vertex computations when compared to the equivalent level using mesh geometry. Our per-joint impostors also exhibit these advantages, but with a lower per-fragment load: instead of relief mapping, which requires multiple dependent texture accesses per fragment, our shader requires a single texture access.

		Polygonal mesh		Relief maps [Beacco et al., 2011]		Per-joint impostors	
View	Agents	No LoD	LoDs	No LoD	LoDs	No LoD	LoDs
Aerial	1000	29	40	37	116	190	293
Aerial	2000	14	21	18	56	90	152
Aerial	4000	7	12	9	29	45	79
Aerial	8000	3	6	4	15	25	43
Aerial	16000	2	4	2	7	12	22
Street	1000	31	45	40	167	200	349
Street	2000	16	22	19	81	96	181
Street	4000	8	21	9	42	48	93
Street	8000	5	7	4	22	25	51
Street	16000	2	5	2	10	12	27

Table 6.2: Frame rates using polygonal meshes, relief impostors, and our per-joint impostors.

6.2.4 . Discussion

The distance threshold for switching between geometry and impostors is the most important factor to render crowds with an acceptable visual appearance. For the results presented in this section, we have empirically found that 5 meters is a reasonably good distance to make this switch for typical viewing conditions [Beacco et al., 2011]. It would be interesting though to run perceptual studies to determine the optimal distance depending on field-of-view, camera position and crowd density [McDonnell et al., 2005]. Compared to other view-dependent approaches [Tecchia et al., 2002], our approach offers more flexibility for crowd animation, and saves memory space as impostors are not captured for each animation frame. Compared to polygonal rendering, independently of the distance threshold chosen, our per-joint impostors always provide higher frame rates. Compared to relief impostors [Beacco et al., 2011], our per-joint impostors also perform better since their cost is nearly independent of the screen-projected area of the character, and thus can be used for shorter distances without increasing the rendering cost. Therefore, as long as the visual quality remains acceptable, the threshold distance could be smaller than the one used for relief impostors. Rendering our per-joint impostors instead of relief impostors offers also other advantages such as eliminating the artifacts due to missing ray-surface intersections, and supporting non-heightfield surface details (e.g. complex hairstyles). Unlike some previous work [Kavan et al., 2008b], our construction algorithm is fully automatic.

6.3 . Conclusions on Crowd Rendering

In our first contribution each character is encoded through a small collection of textured boxes storing color and depth values. At runtime, each box is animated according to the rigid transformation of its associated bone and a fragment shader is used to recover the original geometry using a dual-depth version of relief mapping [Oliveira et al., 2000], recovering surface details and reproducing view-motion parallax effectively, at the expense of some per-fragment overhead. Beyond a certain distance threshold, this compact representation is much faster to render than traditional level-of-detail triangle meshes. Our user study demonstrates that replacing polygonal geometry by our relief impostors produces negligible visual artifacts. Although this method provides a high-quality representation for distant meshes, its applicability is limited to relatively far-away characters due to the per-fragment cost of the fragment shader. So for close-up agents it is usually faster to render fully animated 3D skinned characters.

Our second contribution results in a more efficient approach. Instead of using six orthogonal relief maps for each joint, which requires multiple dependent texture accesses per fragment, we use flat impostors created by sampling each joint from multiple view directions. These view directions correspond to the faces of a subdivided icosahedron. A spherical Voronoi map is computed from it, and a cube map is built by projecting the Voronoi cells onto the cube faces, thus encoding for each texel the ID of its nearest discrete sample. At runtime, for each fragment, a single cube map texture look-up is enough to retrieve this sample ID, and another one to retrieve the color of the fragment from that sample. Since these impostors are intended to be valid for any pose, a key issue is to properly define which part of the geometry influenced by each joint must be represented as opaque pixels in the corresponding impostor. These parts are encoded through opacity masks, which are computed by considering how the geometry of each bone is affected by the transformation of neighboring joints.

Both per-joint impostor approaches allow us to render tens of thousands of

characters in real-time. Encoding per-joint geometry and appearance with relief maps provides the highest image quality at the expense of a higher per-fragment overhead, which in practice limits their applicability to distant characters. View-dependent flat impostors are more demanding in terms of texture memory and construction time, but provide the highest runtime performance even for close-up characters. With properly chosen switch distances, both representations outperform polygonal meshes with negligible visual artifacts. Regardless of the particular encoding, per-joint impostors support arbitrary animation cycles and animation blending, a missing feature in competing per-character impostors.

PUBLICATIONS

Our work in rendering has yielded the following publications ([Beacco et al., 2010a, 2011, 2012, 2013a]):

- A. Beacco, B. Spanlang, C. Andujar, and N. Pelechano. *Output-sensitive rendering of detailed animated characters for crowd simulation*. In CEIG Spanish Conference on Computer Graphic, 2010
- A. Beacco, B. Spanlang, C. Andujar, and N. Pelechano. *A flexible approach for output-sensitive rendering of animated characters*. Computer Graphics Forum, 30(8):2328 - 2340, 2011
- A. Beacco, C. Andujar, N. Pelechano, and B. Spanlang. *Efficient rendering of animated characters through optimized per-joint impostors*. Computer Animation and Virtual Worlds, 23(1): 33 - 47, 2012
- A. Beacco, C. Andujar, N. Pelechano and B. Spanlang. *Crowd Rendering with per joint impostors*. Poster in the 24th EUROGRAPHICS Symposium on Rendering (EGSR 2013), Zaragoza, Spain, 2013.

Furthermore, we have submitted a survey on real-time rendering of crowds, including most of the related work presented in 3.3. Thanks to Doctor Carlos Andujar for his collaboration in our rendering work, and to Bernhard Spanlang for his help and support when using Halca [Spanlang, 2009].

7 . Crowd Framework

As we have seen in all the other chapters, there has been a large amount of research on simulation, animation and rendering of crowds, but in most cases they seem to be treated separately as if the limitations in one area did not affect the others. At the end of the day the goal is to populate environments with as many characters as possible in real time, and it is of little use if one can for instance render thousands of characters in real time, but you cannot move more than a hundred due to a simulation bottleneck. The work presented in this chapter aims at providing a framework that lets the researcher focus on each of these topics at a time (simulation, animation, or rendering), and be able to explore and push the boundaries on one topic without being strongly limited by the other related issues. Therefore we introduce a new prototyping testbed for crowds that lets the researcher focus on one of these areas of research at a time without loosing sight of the others. We offer default representations, animation and simulation controllers for real time crowd simulation, that can easily be replaced or extended. Fully configurable level-of-detail for both rendering and simulation is also available.

7.1 . Introduction

In this chapter, we present a novel framework that embeds these three elements: Simulation, Animation and Rendering of crowds. Each of them presented in an independent-but-linked modular way. The final tool becomes then a prototyping testbed for crowds that allows the researcher to focus on one of these parts at a time without loosing sight of the other two. This tool could also be very handy for introducing crowd simulation in the classroom. Our bundle includes some basic resources such as character models, libraries and implemented controllers interfaces, which allows the researcher to have a basic crowd simulation engine to get started, and to be able to focus exclusively in a particular area or research. Our module also lets the researcher to have communication and interaction between these areas, if he desires to treat some of them in a more dependent fashion. We offer default representations, animation and simulation interfaces and controllers in a modular way, that can easily be extended with your new research work. We also include a fully configurable system of level-of-detail for animation, visualization and simulation.

Motivation and Current State

One of the main motivations for building this tool was the will to integrate in one system all the previous contributions presented in this thesis. As the reader may have noticed, these have been implemented using different tools such as the Unity engine, XVR or simply C++. After working and analyzing all these tools, as well as some others (in chapter 3, section 3.4), we decided that the best approach was to design the present testbed. Unfortunately, to reimplement all the previous contributions, although possible, is not as straight forward as it would be desirable. This is why not all the contributions presented in this thesis have been integrated in the current platform yet. Currently, we have been able to add our impostors generation and rendering to this platform (Chapter 6). All the other contributions made previously implemented using Unity and XVR will be attacked later.

7.2 . Overview

We present CAVAST: the Crowd Animation, Visualization and Simulation Testbed, a new prototyping and development framework, made for and by researchers of the graphics community specialized in crowds. The goal of this work is to provide a framework with state-of-the-art libraries and simple interfaces to ease the work of starting a project regarding simulation, animation or rendering of crowd. This framework provides a basis to start working in this field, focusing on solving a specific problem in simulation, animation or rendering without loosing sight of the other aspects, that is conserving the communication and interaction possibilities between them.

Figure 7.1 shows a rough overview of the classes and interfaces present in CAVAST. The scene render engine is going to need some basic information to render an agent. This includes at least: visual representation (even if it is just a 2D point), position and orientation.

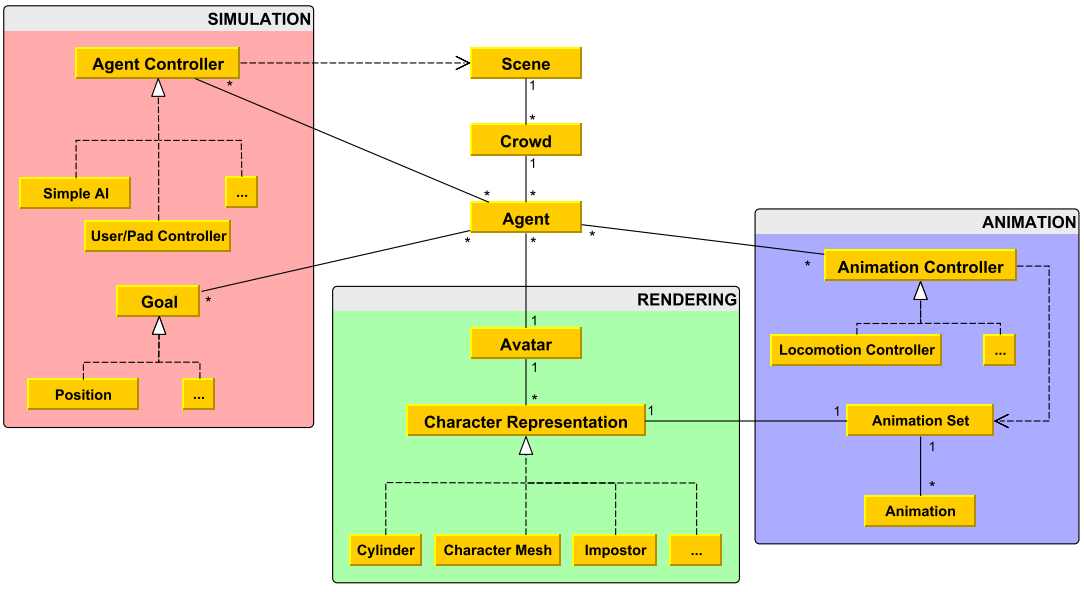


Figure 7.1: Diagram with a rough overview of the classes in CAVAST.

The Agent class is the core class of our framework and it is linked to:

- One or more Agent Controllers. These controllers deal with the kind of simulation methodology used for pathfinding and local motion. It also

needs an interface to assign and describe one or more Goals to the agents (into a queue).

- One Avatar containing the Character Representation used by the Rendering module. Notice that one avatar can be shared by many agents.
- One or more Animation Controllers, an interface class in charge of the Animation module to deal with skeletal animation.

Each one of these interfaces are described in more detail in the following subsections.

7.2.1 . Simulation

The Simulation module needs to include at least an implementation for the Agent Controller interface. This module will be responsible for moving the Agent in the virtual environment. The Agent Controller consists of either an implementation of a behavioral model based on for example steering, social forces, rules, or any other model that includes the AI of the agent. Alternatively, it could be directly controlled by the user input through a User Controller. When the Agent is controlled autonomously, it is usually required to have certain goals. The type of goals required vary from one system to another. Our framework provides a Goal interface and a basic implementation consisting of just a Position. The user can expand the kind of goals by implementing new Goals such as a position with orientation, or a position with orientation and time stamp. The Agent Controller has access to the Scene in order to query information about fixed obstacles, dynamic obstacles and other members of the Crowd. This information can be used for collision avoidance for example. Notice that the Agent Controller can also integrate physics libraries to accelerate collision detection if needed.

7.2.1 . Pathfinding

As for pathfinding, we provide a Pathfinder interface class and a basic implementation of the A* algorithm [Dechter and Pearl, 1985]. This could be easily expanded to new Pathfinder classes such as D* Lite [Koenig and Likhachev,

2002], ARA* [Dechter and Pearl, 1985], etc. This class works over a Graph interface class, which can be either a Grid or a Navigation Mesh representing the scene. A Pathfinder also might need a Heuristic to work with, in our case we have a simple Euclidean distance, but it could easily be any other function estimating a cost to reach a node of the Graph. Our current version only provides a Grid representation from a randomized generated scene (filling cells in a grid with obstacles), although we plan to include a navigation mesh creation module from any static 3D geometry loaded in the scene. So, if the Agent Controller has a Pathfinder, it will be used to find a path and generate intermediate goals (way points) to insert into the goals queue of the Agent. Figure 7.2 shows a diagram of the simulation module in more detail, but for the sake of clarity not all the classes have been included in it.

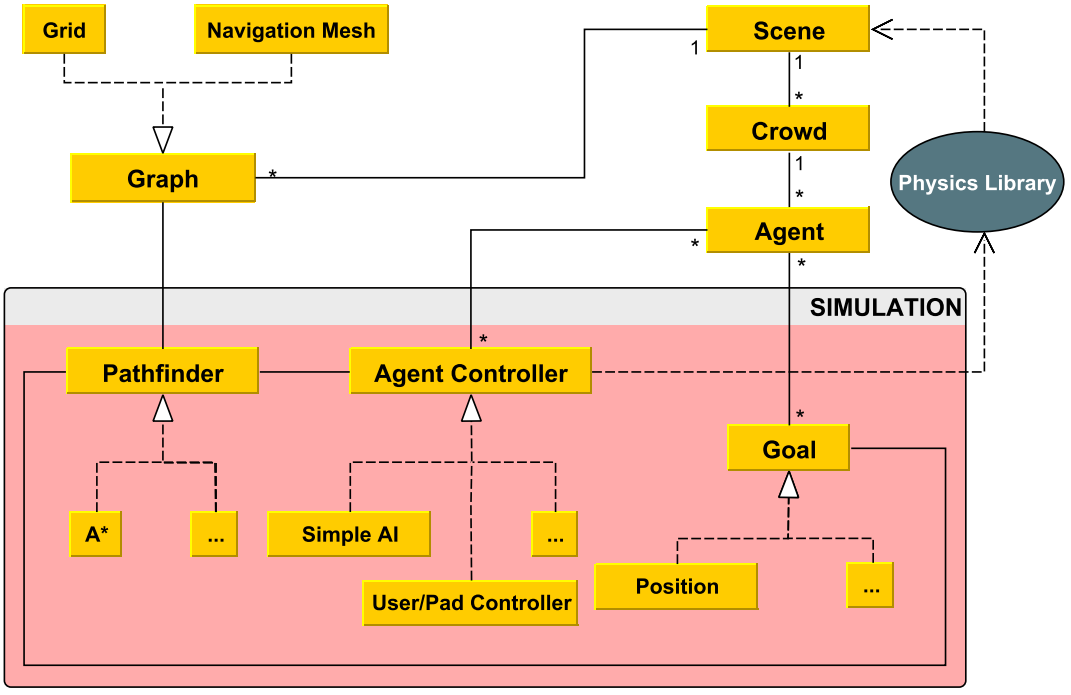


FIGURE 7.2: The simulation module

7.2.1 . AGENT CONTROLLER

The main method an Agent Controller has to implement is `exeSimulation(ref Agent a, float elapsedTime)`. This method will be in charge of actually moving the agent, and will modify attributes of the Agent instance a, such as position, orientation, and velocity, as the result of executing forward the simulation during an elapsed time equal to `elapsedTime`. The other method that an

Agent Controller needs is `getPathFinder()`, returning the PathFinder of the controller if there is any. A Pathfinder requires to implement a function `findpath(ref Graph graph, ref Node start, ref Node end, ref Heuristic h)`, returning a sequence of Nodes from `graph` representing a path from `start` to `end`. A Graph will be composed of Nodes, which can either be Cells in the case of a Grid or Polygons in the case of a Navigation Mesh. A Goal must have a function `isReachedByAgent(ref Agent a)` returning true when it is reached by an Agent instance a.

7.2.1 . Crowd Simulation

Transparent to the user, the Crowd will be in charge of iterating over all its agents. Currently, when an agent does not have a goal it will get one randomly assigned, although an Agent has a function to set up its current Goal. If the Agent Controller has a PathFinder, it uses it to find waypoints and insert them into the queue as intermediate goals. Once a goal has been reached, the next goal of the queue is the new goal to be used by the `exeSimulation` method. Then the Agent Controller executes its simulation for the elapsed time.

Figure 7.3 shows an example crowd of 200 cylinders moving in a random generated grid, using A* for path planning and Reynolds [Reynolds, 1999] for steering, using CAVAST.

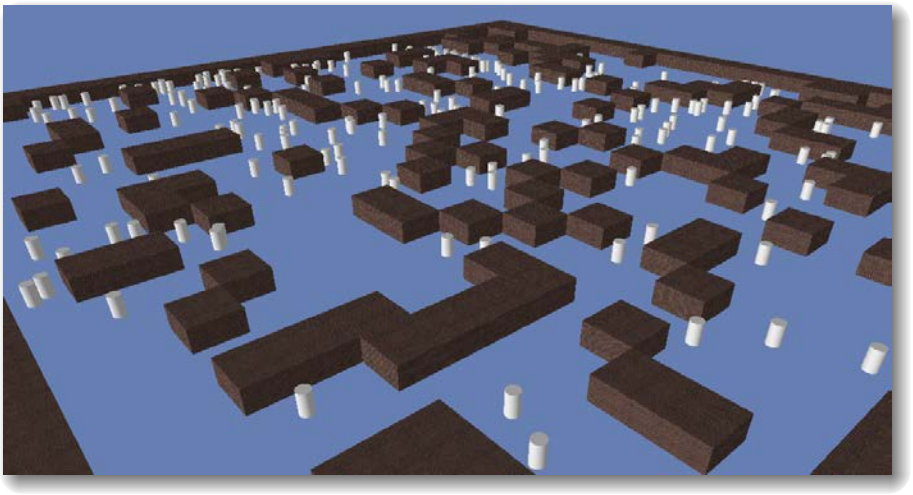


Figure 7.3: An example simulation of 200 agents using CAVAST.

7.2.2 . Rendering

An Agent has to be associated with an Avatar. We call an Avatar a collection of one or more Character Representations, along with the main dimensions we want for it, that is the size we want for our representations for each 3D axis. Having different representations allows us to use them for different levels of detail (LODs), and the main size of the Avatar allows us to be consistent between different representations. If for example we want to replace our character by some 3D model of a cylinder for far away agents, we will be able to scale the cylinder to the same dimensions as the original 3D mesh by scaling its bounding box. An implementation of the Character Representation interface must provide a Shader Program (with at least a Vertex and a Fragment shader) and the methods `render()`, to render it individually, and `instancedRender(int n, ref VertexBufferObject instancesDataBuffer)` in order to use instancing [Dudash, 2007b], efficiently rendering n instances using the corresponding data in `instancesDataBuffer` for each one of them. We also request for methods to get the bounding box, the bounding sphere and the bounding cylinder radius of the representation in order to help for collision detection and selection algorithms.

7.2.2 . Scene Render

CAVAST uses its own scene library to manage the crowd scene. Its main modules are a Scene Graph represented by a Scene Tree, and a Transform class which is the nodes class of our Scene Tree. A Transform has a name and contains one absolute transformation matrix and one relative to its parent. A Transform can contain a Render Object, although it is not required and therefore a Transform can be empty (to perform relative transformations). If so, it also needs a Shader Program name to bind it before rendering. A Render Object is an interface class for the scene library to know how to render things in the Scene Graph, and one Avatar is a subclass of a Render Object. All of this is transparent to the user who wants to implement its own Character Representations. CAVAST and the Avatar class will be in charge of creating the proper transforms and add them to the scene.

When a Crowd *i* is added to the Scene, a Transform named "Crowd i" is added to the scene root. When adding agents with the same Avatar, their corresponding Transforms are grouped in a group Transform with the name of the Avatar. Inside that transform, agents are also grouped by Character Representations in a group transform for each one. When using level of detail, agents can change between Character Representations, and thus need to change between groups. This is dynamically and automatically carried out by the Crowd class. The main reason for doing this is to be able to perform instancing, and to accelerate the rendering of all the instances sharing the same Character Representation. Having all the Transforms for all the instances of one representation in the same group makes it fast to fill the instances data vertex buffer object with their individual data. Figure 7.4 shows an example view of a possible scene hierarchy.

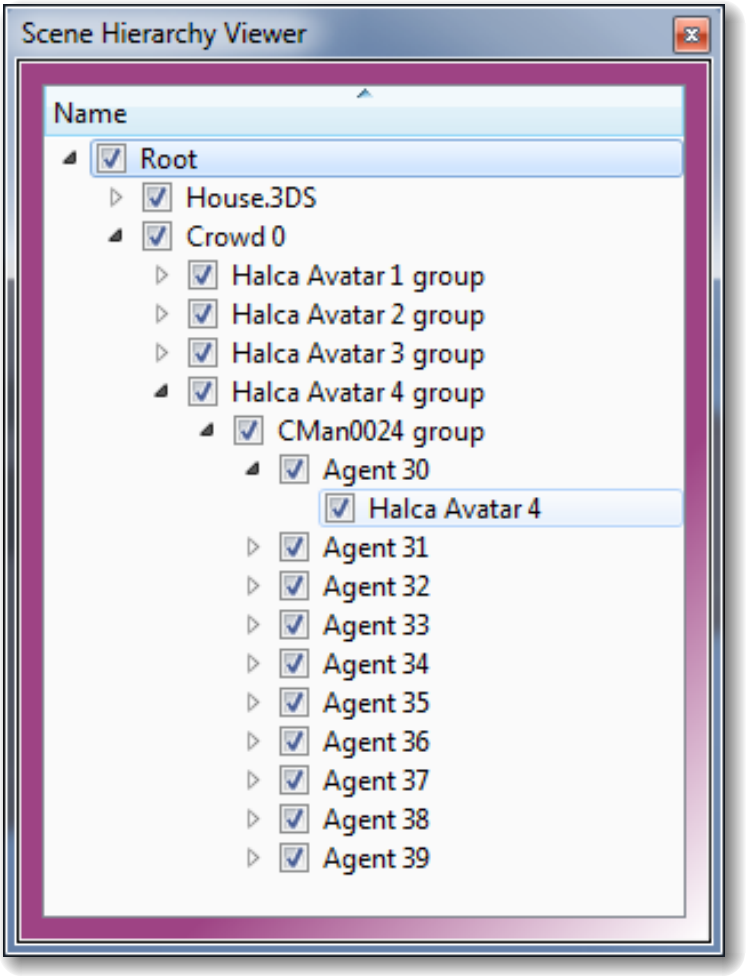


Figure 7.4: An example view of a scene hierarchy.

7.2.2 . Character Representations

Our current system provides two Character Representations: a basic one based on just a cylinder 3D mesh (which could be easily extended by any 3D static mesh) and an animated Character Mesh in the Cal3D format [Cal3D, 2014] through HALCA [Spanlang, 2009]. The user could introduce any other Character Representations such as those based on impostors. We have also recently incorporated an automatic impostor generation module, working with Cal3D characters, and their corresponding implementations of the Character Representation interface.

The constructor function should load the necessary geometry and resources (textures). A Shader Program should be loaded using our Shader Manager, and its name should be retrieved by the `getShaderName()` function (one of the Character Representation interface functions). The `render()` function only needs to send all the necessary information to the shader (via uniforms, attributes, or whatever you want to use) and to render the geometry. In addition, the `instancedRender(int n, ref VertexBufferObject instancesDataBuffer)` function needs to bind the vertex buffer object and enable any vertex attribute pointers necessary to render the different instances.

Notice that the shader binding is done by the scene library when rendering the corresponding transform of the agent. This allows the user to dynamically change the shader through the interface (whenever the new shader is able to handle the same data).

Figure 7.5 shows an example scene where 500 Avatars of the same type are represented using two Character Representations, a static 3D geometry for closer agents, and cubes for farther away agents. And figure 7.6 shows 1000 agents represented using 4 different Avatars with only one Character Mesh for each one.

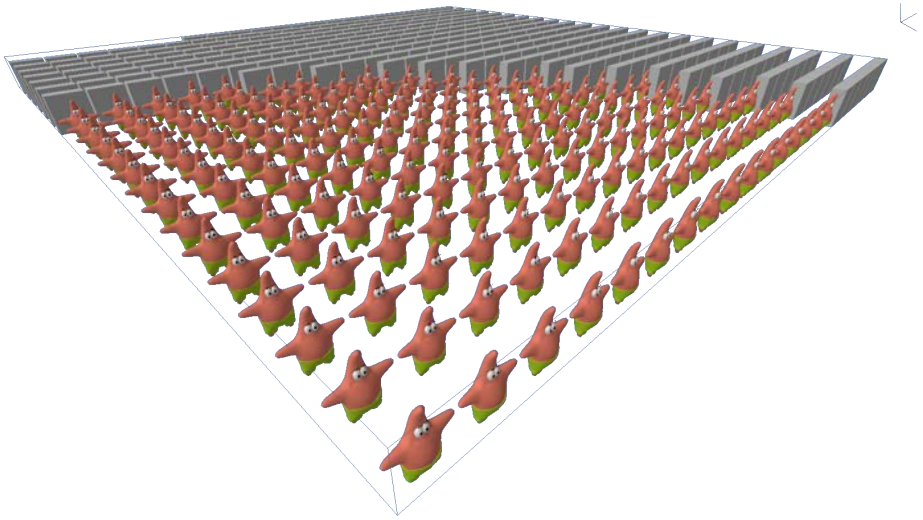


Figure 7.5: Level of detail: 500 Avatars with a 3D static mesh and with a cube for agents at 30 meters or more from the camera.

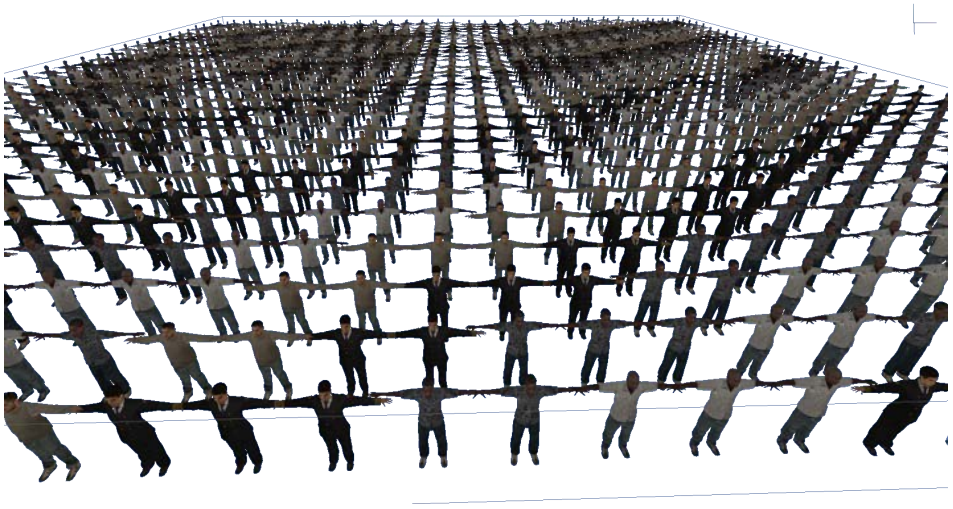


Figure 7.6: 1000 characters of 4 different types of Avatars using only one Character Mesh of around 5000 triangles for each one (without level of detail, nor animation).

7.2.3 . Animation

A Character Representation may be animated, that is, have a method of adapting the character representation to different poses. In order to be able to use the best animations at each time, it is necessary to extract certain semantic information from the original animation clips. An Animation Controller will then be in charge of using that information to decide which animation to play or synthesize.

7.2.3 . Preprocessing Animations

Independently of the Character Representation chosen, if we want to animate it then we need to have an Animation Set. An Animation Set is an interface class offered by our framework, which is composed of different Animations (also called animation clips). The Agent can have an Animation Controller, in charge of selecting or synthesizing the best animation, in order to properly follow its current motion. The Animation Controller is thus dependent on the current Character Representation and on its Animation Set.

When using skeletal animation through an animation library, the animation controller will probably make an extended use of it. For example, we could have an Avatar linked to an Agent, whose main Character Representation is a character, rendered and animated with an accelerated animation library, such as HALCA [Spanlang, 2009]. That Agent could have a Locomotion Controller implementing an Animation Controller, which will be using that library to preprocess the Animation Set, by analyzing and extracting information from each Animation. Then at execution time, the Locomotion Controller could read the velocity values of the Agent, decide which is the best Animation to play at that moment, and use HALCA to do it.

7.2.3 . Animation Controller

The Animation Controller needs to implement a method `animate(ref Agent a, ref CharacterRepresentation cr)`. Although the same Animation Controller could be used by different Agents and/or different Character Representations, it has sense to assume that the different Character Representations of

the same Avatar could share the extracted information from the Animation Set of the main Character Representation. An avatar should walk or run at the same speed when it is represented by a 3D mesh as when it is by an animated impostor. So even if the Agent Controller is linked to the Agents, we can think that there should be at least one Animation Controller instance for each Avatar. And therefore a method `processAnimations(ref AnimationSet)` should be implemented too (although it can be empty if it is not required by your controller implementation). With this interface the user should be able to implement his own animation controllers, to have different locomotion styles, idle behaviors, etc.

7.2.3 . Instancing and Palette Skinning

At every frame agents are sorted in the Scene Tree, being grouped by Avatar and by Character Representation. This way we can fill a Vertex Buffer Object with the transform matrices of all the Agent instances, send it to the GPU, and perform just one render call with instancing. If we want to animate all the instances individually, that is, with each agent having its unique animation pose, their animation information must be sent too. This could become a bottleneck and therefore a problem, when the amount of agents is too high and that information is too big.

For example, skeletal animation requires to send matrices or quaternions for each joint of every agent. When talking about thousands of agents with character representations of around 50 bones or more, the amount of information to update and send to the GPU can be very large, and thus the bandwidth between CPU and GPU can become a major bottleneck. A solution is to have all the animations loaded in the GPU and perform there the matrix palette skinning [Dudash, 2007a], having a different pose for every instance.

We therefore suggest that an Animation Set class of a Character Representation, in addition to have all the analyzed animations, implements a function `createBufferTexture()` to create the buffer texture where all the animations will be encoded. This way the buffer will be bound and the vertex shader will be able to use it to perform instancing and palette skinning at the same time. The advantage of this will be to have individual agents playing different animations.

The counterpart of simply doing this would be that the vertex shader would be in charge of computing the blended pose (given different animation instances and weights), by blending within two key frames for each animation, and between the resulting poses of the different weighted animations. The number of animations blended at the same time, as well as the geometry complexity of the character would have a high impact over the performance.

To avoid this we use an additional vertex shader with transform feedback. This allows to compute the blended poses of all the agents in the GPU, but with a per joint cost rather than a per vertex cost. The computational cost depends therefore on the number of joints to be deformed rather than on the number of vertices of the character mesh.

7.2.4 . Integration

Agents within a Crowd are represented by Avatars and their Transforms, being part of the Scene Graph. At runtime the Scene is rendered by our render engine using the Transform information of every Render Object. In order to render each Avatar, it takes the position and orientation of the corresponding agent and builds the transformation matrix, which is then used to render the selected Character Representation. Depending on the distance to the camera, the Avatar selects one Character Representation or another and its Shader Program (implementing the Level of Detail selection). The user sets the LODs through an interface. The simulation is carried out on another thread, executing it for every instance of every agent. Since an Agent Controller has access to the Scene and to the Crowd, it can also have access to the possible obstacles of the scene as well as to the other agents. The Animation is performed right before rendering the Avatar.

7.3 . Features

Although this chapter presents an ongoing work to build a tool for crowd simulation, animation and rendering, we offer in the current version a fully working framework with the main functionalities and controllers already integrated. This tool is built using OpenGL 4.3 for rendering and we offer different GUIs using Qt5 (see figure 7.7). It is therefore straight forward for the user to create a crowd, add agents with different Avatars, and assign controllers. A Shader Manager and a Shader Editor are available allowing quick shader editing. Managers and graphic editors to configure the Levels-Of-Detail for Simulation, Animation and Visualization are also available. The user can interactively edit an Avatar by changing the different Character Representation switching distances and its main dimensions, or you can add more than one Agent Controller and more than one Animation Controller with different distance thresholds to one or more Agents for each one (please see the accompanying video for a demo).

The current tool has implemented a Character Representation called Model 3DS Representation without animations which just uses a 3ds mesh file as a character, and one HALCA Character which loads characters animated with HALCA [Spanlang, 2009]. In future versions we want to add a representation for characters in the FBX format by using the FBX SDK [Autodesk, 2014b]. There are also two simple unoptimized Agent Controllers that perform a wandering behavior, one with collision detection and avoidance, and the other without it. To detect collisions each agent simply iterates over the other agents, predicts future positions (according to the current velocity) and checks intersection between the two agent radius. We also provide a simple version of some of Reynolds steering behaviors [Reynolds, 1999]. In addition to that we offer a simple random grid generator and an A* Pathfinder that can work with the Reynolds controller. As a proof of concept we have also integrated the RVO2 library [van den Berg et al., 2014], allowing us to have a controller using it to simulate the agents. The integration was easy, requiring less than 2 hours.

Also integrated in the current systems, there is a basic Animation Controller that works with our HALCA Characters and analyzes their animations. It extracts the root speeds of each animation and therefore is able to select the best

FIGURE 7.7: GUIs for managing the crowd agents, avatars and controllers in CAVAST.

one and adjust its speed to match the speed of the agent in the crowd simulation. This way we reduce the foot-sliding effect.

Figure 7.8 shows a crowd of agents represented by HALCA Characters, moved by our Reynolds Controller and animated with our basic Animation Controller.

Frustum culling using bounding spheres, and occlusion culling with bounding boxes are implemented. As it has been previously mentioned, the system is also prepared to support instancing [Dudash, 2007b]. Stereoscopic visualization is also implemented, so the port to a virtual reality environment is also possible.

7.4 . Results and Discussion

Even in its current preliminary stage, we believe that CAVAST can be a powerful tool for researchers and students. To show the potential of CAVAST, we provide some performance measurements such as frame rates for different scenes and different conditions, but it is important to notice that CAVAST is designed to be flexible and adapt to the needs and conditions of the work carried out by the user. Therefore performance measurements will strongly vary depending on the different controllers used or implemented, but having said this, we believe that the current framework provides higher performance benefits when it comes to crowd simulation, than other tools mentioned in 3.4. For example, using our test equipment (PC Intel Core i7-2600K CPU 3.40 GHz, 16 GB Ram, and a GeForce GTX 560 Ti), we can render a thousand characters represented with a non-animated 3d model of 3000 polygons each at 90 fps. If we add an unoptimized Agent controller with collision detection, frame rate drops to 60 fps. But playing with the different LODs of the Agent controllers and of the Character Representations, we can again reach frame rates over 100 fps. Using animated characters with HALCA of 5000 polygons each, but without an Agent Controller and all playing the same animation and the same pose (thus sharing the animation data), you can have a thousand of them at 30 fps. But using an

Figure 7.8: A real time visualization of a crowd of agents represented by HALCA Characters. using the Reynolds Agent Controller and the basic Animation Controller from CAVAST.

Animation Controller and giving individuality to the animation of each agent, can drop the frame rate to 23 fps. Adding an Agent Controller and thus forcing to blend more than one animation for each agent can drop again the frame rate to 14 fps.

These examples are just to illustrate that the performance of CAVAST is strongly related to the Character Representations you use, the Controllers you implement and the LOD configurations you choose. One interesting feature we would like to add is a profiling tool that could give you automatically information about your controllers and representations. This should allow the user to identify potential bottlenecks easily. It could also compare the performance between all controllers, and automatically seek for the optimum LOD set-up in order to keep a real-time frame rate. Figure 7.9 shows a screenshot of CAVAST with a 350 agents scenario configured to have 60 fps.

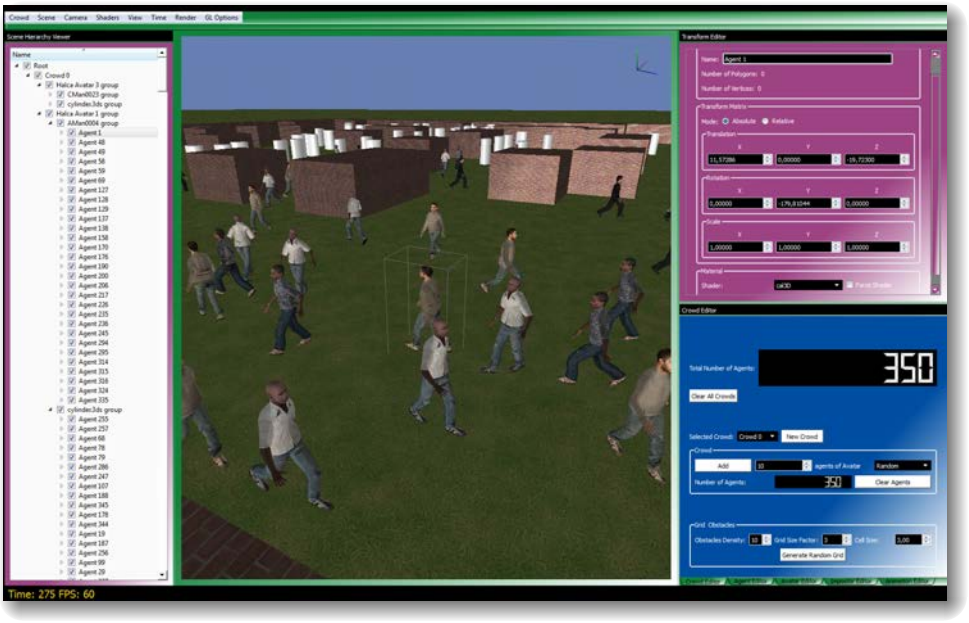


FIGURE 7.9: Screenshot of CAVAST.

7.5 . Conclusions on Crowd Framework

We have presented a new prototyping and development tool for crowds research integrating animation, visualization and simulation: CAVAST. By implementing some controller interfaces and/or using some default ones, the CAVAST framework allows the user to start a new research project on crowds with all these parts running in real-time, featuring configurable Level-of-Detail and multi-thread. Although it may seem CAVAST do not outperforms other existing systems, our performance is strongly dependent on what you do with it. Our main contribution with CAVAST is therefore a flexible framework and a powerful tool with out-of-the-box crowd sandbox features. We believe that CAVAST could be useful also as an education tool for crowd simulation courses, allowing the students to quickly and easily visualize the results of their different algorithms.

Publications

The contents of this chapter has yield the following publication [Beacco and Pelechano, 2014]:

- A. Beacco and N. Pelechano. *CAVAST: The Crowd Animation, Visualization, and Simulation Testbed.*. EUROGRAPHICS Spanish Conference of Computer Graphics (EGse CEIG 2014), Zaragoza, Spain. 2-4 July 2014.

8 . Future Work and Conclusions

This chapter includes all the future work we want to attack regarding all our contributions. We also try to wrap up all our work and give a summary of conclusions on real-time crowds.

8.1 . Future Work

Our first addition to our simulation work would be to extended our multi-domain system to use our footstep planner. Planning at a footstep level would be a finer resolution domain that could be used in cases with high density of obstacles or other agents, resulting in more natural and collaborative tasks between agents. An interesting part of this project would be to think of a way of implementing the tunneling between the space-time domain and the footstep domain. Another idea for our footstep planner would be to have a special class of actions, constituting a reactive domain that would only be used in case of an imminent threat. Having more characters and different sets of actions that can be used depending on the situation would also accelerate the search and give better results to our simulations in constantly changing dynamic virtual environments.

Our work on removing the foot sliding effect with the APM, required a foot on the floor at all times to calculate root displacement. This is a big limitation since it does not allow for running animation where both feet can be in the air during some frames. This problem should be addressed in the future. The library employed for this first work was hand created, and thus our original animations suffered from rigidity which affects the overall look of the crowd. Since the quality of the final crowd animation depends strongly on the set of animation clips available, having time aligned motion capture animation clips will achieve more natural looking results. We plan to implement the APM into our testbed tool, CAVAST, in order to use it with motion capture clips which we have now been able to retrieve.

For our footsteps-driven animation system, we would like to extend our barycentric coordinates interpolator to 3D space with the third coordinate being the root velocity. This will free our system from the polar band interpolator which not only takes longer to compute but also selects too many animations which results in slower blending. One thing to explore could be to interleave the execution of the Footstep-based Locomotion Controller from different characters in different frames, ensuring we do not execute it for all the agents in the crowd every frame.

As for our rendering work, we would like to conduct perceptual studies supporting the LOD selection instead of the metric based on RMS we used for our relief impostors. This will evaluate not only the quality of static images but also the impact of visualizing animated characters through our impostors. We are currently about to perform more user studies to compare and better evaluate our flat per-joint impostors, against relief impostors and classic impostors [Tecchia and Chrysanthou, 2000].

Regarding our crowd framework, CAVAST, implementing and integrating all the contributions of this thesis would be a huge step. We could provide callbacks for potential simulation events (such as a fire alarm) to the Agent Controller interface, and add the corresponding trigger button to the GUI, letting the user to implement the behavior in its own controllers. We plan to separate the path-finding simulation module and move it to another kind of controller interface, that can easily be linked to a navigation mesh generator module. A nice thing to add would be a generalized random generator of scenes (not just working with a grid or axis aligned objects), as well as some challenging example and benchmark scenarios. Finally, we would like to have evaluation and profiling tools, such as an automatic output of statistics or automatic perception tests. For example an automatic render of the same scene or simulation with different controllers. This might imply to add the possibility of recording and playing back simulations. Currently the code of CAVAST is not multi-platform because we are using a Windows version of the HALCA library, although it should be possible to port it to other platforms when using other animation libraries. Another limitation of the current version is that the user needs to rebuild the entire application to add new controllers, so we plan for a plugin API or a scripting interface using LUA or C#. Also, characters must be added into the code before being imported, but it should be easy to add code in order to automatically import resources in a specific folder. We would like to release a free open source version of the code and make it available online soon.

8.2 . Conclusions

In chapter 2 we introduced different axis of complexity for the problem of simulating, animating and rendering crowds in real-time. As we summarized in section 2.12, the elements in this classification are:

- Agent Complexity
- Control Granularity
- Environment Complexity
- Animation Quality
- Visual Quality
- Variety
- Scale
- Performance
- Global coherence and consistency

We stated that the ideal outcome would be to improve each of these elements, but that pushing the boundaries of one of them might compromise the others, specially performance. Throughout this thesis and its contributions we have seen some of the thresholds appearing when doing so.

Our work on simulation has shown how we can have different agent complexities, controlled with different granularities in complex environments with undeterministic dynamic events in an efficient way. Although we are still far from reaching full mesh agent complexity and the highest level of control granularity over the whole body of the agents in real-time, we have proposed an elegant approximation to maintain efficiency.

Our work on animation proposes new techniques to represent with enough quality the motion of the agents within a crowd. Due to memory and computation restrictions we have limited our approaches to using small databases of

motion capture clips. We think that these allow us to synthesize natural enough animations in order to achieve a crowd moving smoothly in the environment. The most important factor here was to maintain coherence and consistency with the reflected simulation.

Regarding variety we find that there are still many open problems, such as having individuality at all levels (appearance, animation and even behavior). To achieve variety of appearances, tessellation shaders could become a powerful tool in order to add geometric variation to close-up characters, using stochastic techniques. Hardware tessellation might also accelerate the skinning process by rigging and animating a mid-resolution mesh that can be refined dynamically into a high-resolution mesh.

Talking about our work on rendering, we must think that beyond some maximum viewing distance, a 3D character projects into a single pixel. At such distances it does not make sense to spend effort deciding which representation to use, since a single colored point would be valid. Moreover, at such distances we may not be visualizing microscopic simulations anymore, but crowds may be moving at a macroscopic level. In such a case, crowd rendering should be approached in a completely different way, representing massive groups of people flowing, and not individual agents anymore. There are other questions about what humans are able to perceive within their field of view, such as what is the real maximum amount of agents that can be perceived simultaneously by a single viewer, what is the minimum visual quality they need to have, or what are the visual queues users are able to distinguish. This also applies to the animation and simulation quality we are able to perceive. As we have seen, some perceptual studies already attack the perception problem but focusing on agent variability. Additional perception studies and psychophysical experiments are definitely required to answer the questions above and to discover the boundaries of perception in the context of real-time crowds.

Finally, performance and scale are the two main limiting and bounding axis in this research. As we desire real-time crowds we have a limited computation time, as well as a limited number of memory resources. On one hand, by grouping far agents into macroscopic simulations and representing them with a different entity than individual 3D characters, we could use these resources in a different way and scale even more our crowds. But we would not be anymore in the same explicit area of research we have been. In some way, defining

at which point agents can be considered as part of a macroscopic crowd could indicate the maximum target value in our scale factor. On the other hand, the computational limitation could be reduced by using some of the modern GPU capabilities. Most of the crowd simulation and animation processes could be transferred and computed into the GPU, thus being able to parallelize the computation of all the agents. One advantage of this will be to avoid the amount of information to be transferred between the CPU and the GPU, as it will all be in the GPU, but of course, this will still be restricted by the graphics memory.

. Bibliography

Allbeck J. CAROSA: A Tool for Authoring NPCs. In *Proceedings of the Third International Conference on Motion in Games*, MIG'10, pp. 182–193. Springer-Verlag, Berlin, Heidelberg (2010).
URL Link Cited on p. 105

Andujar C., Boo J., Brunet P., Fairen M., Navazo I., Vazquez P. and Vinacua A. Omni-directional Relief Impostors. *Computer Graphics Forum*, volume 26(3):pp. 553–560 (2007).
URL Link Cited on p. 186

Arikan O. and Forsyth D. Interactive motion generation from examples. In *SIGGRAPH*, pp. 483–490. ACM (2002).
URL Link Cited on p. 55

Assarsson U. and Möller T. Optimized View Frustum Culling Algorithms for Bounding Boxes. *J. Graph. Tools*, volume 5(1):pp. 9–22 (2000).
URL Link Cited on p. 87

Aubel A. and Thalmann D. Realistic Deformation of Human Body Shapes. In *Proc. Computer Animation and Simulation 2000*, pp. 125–135 (2000).
URL Link Cited on p. 73

Aubel A., Boulic R. and Thalmann D. Animated Impostors for Real-Time Display of Numerous Virtual Humans. In *VW '98: Proceedings of the First International Conference on Virtual Worlds*, pp. 14–28. Springer-Verlag, London, UK (1998).
URL Link Cited on p. 80, 99

Aubel A., Boulic R. and Thalmann D. Real-time Display of Virtual Humans: Levels of Details and Impostors. *IEEE Transactions on Circuits and Systems for Video Technology*, volume 10:pp. 207–217 (2000).
URL Link Cited on p. 86, 220

Autodesk. 3D Studio Max.
http://usa.autodesk.com/3ds-max/ (2014a).
URL Link Cited on p. 60, 66, 102

Autodesk. FBX SDK.
http://www.autodesk.com/products/fbx/overview (2014b).
URL Link Cited on p. 103, 250

Autodesk. Maya. http://usa.autodesk.com/maya/ (2014c).
URL Link Cited on p. 60, 102

Baboud L. and Décoret X. Rendering geometry with relief textures. In *GI '06: Proceedings of Graphics Interface 2006*, pp. 195–201. Canadian Information Processing Society, Toronto, Ont., Canada, Canada (2006).
URL Link Cited on p. 186

Bærentzen J. Hardware-Accelerated Point Generation and Rendering of Point-Based Impostors. *J. Graphics Tools*, volume 10(2):pp. 1–12 (2005).
URL Link Cited on p. 77

Baran I. and Popović J. Automatic rigging and animation of 3D characters. *ACM Trans. Graph.*, volume 26(3):p. 72 (2007).
URL Link Cited on p. 74

Beacco A. and Pelechano N. CAVAST: The Crowd Animation, Visualization, and Simulation Testbed. In *CEIG Spanish Conference on Computer Graphic*, pp. 1–10 (2014).
URL Link Cited on p. 254

Beacco A., Spanlang B., Andújar C. and Pelechano N. Output-Sensitive Rendering of Detailed Animated Characters for Crowd Simulation. In *CEIG Spanish Conference on Computer Graphic* (2010a).
URL Link Cited on p. 235

Beacco A., Spanlang B. and Pelechano N. Efficient Elimination of Foot Sliding for Crowds. In *Posters Proceedings, The ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, pp. 19–20 (2010b).
URL Link Cited on p. 182

Beacco A., Spanlang B., Andujar C. and Pelechano N. A flexible approach for output-sensitive rendering of animated characters. *Computer Graphics Forum*, volume 30 (2011).
URL Link Cited on p. 85, 99, 230, 232, 235

Beacco A., Andújar C., Pelechano N. and Spanlang B. Efficient rendering of animated characters through optimized per-joint impostors. *Journal of Computer Animation and Virtual Worlds*, volume 23(2):pp. 33–47 (2012).
URL Link Cited on p. 85, 86, 99, 235

Beacco A., Andújar C., Pelechano N. and Spanlang B. Crowd Rendering with Per joint Impostors. In *Eurographics Symposium on Rendering* (2013a).
URL Link Cited on p. 235

Beacco A., Pelechano N. and Kapadia M. Dynamic Footsteps Planning for Multiple Characters. In *CEIG Spanish Conference on Computer Graphic*, pp. 147–160 (2013b).
URL Link Cited on p. 50, 148

Beeson C. and Bjorke K. Skin in the "Dawn" Demo. In *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*, pp. 45–62. Pearson Higher Education (2004).
URL Link Cited on p. 92

Bharaj G., Thormählen T., Seidel H.P. and Theobalt C. Automatically Rigging Multi-component Characters. *Computer Graphics Forum*, volume 31(2pt4):pp. 755–764 (2012).
URL Link Cited on p. 74

Bishop L., Eberly D., Whitted T., Finch M. and Shantz M. Designing a PC Game Engine. *IEEE Comput. Graph. Appl.*, volume 18(1):pp. 46–53 (1998).
URL Link Cited on p. 87

Blender-Foundation. Blender.
http://www.blender.org/ (2014).
URL Link Cited on p. 102

Botea A., Müller M. and Schaeffer J. Near optimal hierarchical path-finding. *Journal of Game Development*, volume 1:pp. 7–28 (2004).
URL Link Cited on p. 55

Boulic R., Magnenat-Thalmann N. and Thalmann D. A Global Human Walking Model with Real-time Kinematic Personification. *Vis. Comput.*, volume 6(6):pp. 344–358 (1990).
URL Link Cited on p. 58

Braun A., Musse S.R., Oliveira L. and Bodmann B. Modeling Individual Behaviors in Crowd Simulation. In *Proceedings of the 16th International Conference on Computer Animation and Social Agents (CASA 2003)*, CASA '03, pp. 143–. IEEE Computer Society, Washington, DC, USA (2003).
URL Link Cited on p. 45

Brogan D.C. and Hodgins J.K. Group Behaviors for Systems with Significant Dynamics. *Autonomous Robots*, volume 4:pp. 137–153 (1997).
URL Link Cited on p. 45

Bruderlin A. and Calvert T.W. Goal-directed, Dynamic Animation of Human Walking. *SIGGRAPH Comput. Graph.*, volume 23(3):pp. 233–242 (1989).
URL Link Cited on p. 58

Bulitko V., Sturtevant N., Lu J. and Yau T. Graph abstraction in real-time heuristic search. *J. Artif. Int. Res.*, volume 30(1):pp. 51–100 (2007).
URL Link Cited on p. 55

Butterfield J. Animadead: A Skeletal Animation Library.
http://animadead.sourceforge.net/ (2014).
URL Link Cited on p. 103

Cal3D. 3D Character Animation Library.
http://home.gna.org/cal3d/ (2014).
URL Link Cited on p. 103, 188, 222, 245

Carnegie-Mellon-University. Panda 3D.
https://www.panda3d.org/ (2014).
URL Link Cited on p. 103, 104

Carucci F. Inside Geometry Instancing. In M. Pharr (Editor), *GPU Gems 2*, pp. 47–67. Addison-Wesley (2005).
URL Link Cited on p. 93

Chai J. and Hodgins J. Constraint-Based Motion Optimization Using A Statistical Dynamic Model. *ACM SIGGRAPH* (2007).
URL Link Cited on p. 67, 68

Charalambous P. and Chrysanthou Y. The PAG Crowd: A Graph Based Approach for Efficient Data-Driven Crowd Simulation. *Computer Graphics Forum*, pp. n/a–n/a (2014).
URL Link Cited on p. 52

Chen C., Lin I., Tsai M. and Lu P. Lattice-Based Skinning and Deformation for Real-Time Skeleton-Driven Animation. In *Proceedings of the 2011 12th International Conference on Computer-Aided Design and Computer Graphics*, CAD-GRAPHICS '11, pp. 306–312. IEEE Computer Society, Washington, DC, USA (2011).
URL Link Cited on p. 76

Chen Z., Feng R. and Wang H. Modeling friction and air effects between cloth and deformable bodies. *ACM Trans. Graph.*, volume 32(4):pp. 88:1–88:8 (2013).
URL Link Cited on p. 96

Chenney S. Flow tiles. In *ACM Symposium on Computer Animation*, pp. 233–242 (2004).
URL Link Cited on p. 47, 48

Choi M., Lee J. and Shin S. Planning biped locomotion using motion capture data and probabilistic roadmaps. *ACM Transactions on Graphics*, volume 22(2):pp. 182–203 (2003).
URL Link Cited on p. 55, 67

Choi M., Kim M., Hyun K. and Lee J. Deformable Motion: Squeezing into Cluttered Environments. *Comput. Graph. Forum*, volume 30(2):pp. 445–453 (2011).
URL Link Cited on p. 55

Chung S. and Hahn J. Animation of Human Walking in Virtual Environments. In *Proceedings of the Computer Animation*, CA '99, pp. 4–. IEEE Computer Society, Washington, DC, USA (1999).
URL Link Cited on p. 66

Clark J.H. Hierarchical Geometric Models for Visible Surface Algorithms. *Commun. ACM*, volume 19(10):pp. 547–554 (1976).
URL Link Cited on p. 87, 88

Cohen-Or D., Chrysanthou Y.L., Silva C.T. and Durand F. A Survey of Visibility for Walkthrough Applications. *IEEE Transactions on Visualization and Computer Graphics*, volume 9(3):pp. 412–431 (2003).
URL Link Cited on p. 86

Coic J., Loscos C. and Meyer A. Three LOD for the Realistic and Real-Time Rendering of Crowds with Dynamic Lighting. Research Report RN/06/20, Université Claude Bernard, LIRIS, France (2007).
URL Link Cited on p. 83, 84, 99

Cordier F. and Magnenat-Thalmann N. A data-driven approach for real-time clothes simulation. *Computer Graphics Forum*, volume 24:pp. 173–183 (2005).
URL Link Cited on p. 75

Coros S., Beaudoin P., Yin K. and van de Pann M. Synthesis of Constrained Walking Skills. *ACM Trans. Graph.*, volume 27(5):pp. 113:1–113:9 (2008).
URL Link Cited on p. 67

Crow F. Shadow algorithms for computer graphics. *SIGGRAPH Comput. Graph.*, volume 11(2):pp. 242–248 (1977).
URL Link Cited on p. 95

Da Silva M., Abe Y. and Popovic J. Simulation of Human Motion Data using Short-Horizon Model-Predictive Control. *Comput. Graph. Forum*, volume 27(2):pp. 371–380 (2008).
URL Link Cited on p. 63

Dechter R. and Pearl J. Generalized best-first search strategies and the optimality of A*. *J. ACM*, volume 32(3):pp. 505–536 (1985).
URL Link Cited on p. 240, 241

DeCoro C. and Rusinkiewicz S. Pose-independent Simplification of Articulated Meshes. In *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*, I3D '05, pp. 17–24. ACM, New York, NY, USA (2005).
URL Link Cited on p. 89

Dobbyn S., Hamill J., O'Conor K. and O'Sullivan C. Geopostors: a real-time geometry / impostor crowd rendering system. In *I3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games*, pp. 95–102. ACM, New York, NY, USA (2005).
URL Link Cited on p. 82, 83, 99, 197, 223

Dudash B. Animated Crowd Rendering. In *GPU Gems 3*, pp. 39–52 (2007a).
URL Link Cited on p. 94, 230, 248

Dudash B. Skinned Instancing. In *NVIDIA SDK 10* (2007b).
URL Link Cited on p. 94, 243, 251

Egges A. and van Basten B. One Step at a Time: Animating Virtual Characters Based on Foot Placement. *The Visual Computer*, volume 26(6-8):pp. 497–503 (2010).
URL Link Cited on p. 48, 68, 70

Eisemann E., Assarsson U., Schwarz M., Valient M. and Wimmer M. Efficient Real-time Shadows. In *ACM SIGGRAPH 2013 Courses*, SIGGRAPH '13, pp. 18:1–18:54. ACM, New York, NY, USA (2013).
URL Link Cited on p. 95

Epic. Unreal Engine.
http://www.unrealengine.com/ (2014).
URL Link Cited on p. 103, 104

Faloutsos P., van de Panne M. and Terzopoulos D. Composable Controllers for Physics-based Character Animation. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, pp. 251–260. ACM, New York, NY, USA (2001).
URL Link Cited on p. 58

Felis M. and Mombaur K. Using Optimal Control Methods to Generate Human Walking Motions. *Motion in Games*, pp. 197–207 (2012).
URL Link Cited on p. 48

Feng A., Huang Y., Kallmann M. and Shapiro A. An Analysis of Motion Blending Techniques. *Motion in Games*, pp. 1–12 (2012).
URL Link Cited on p. 60

Fiorini P. and Shillert Z. Motion Planning in Dynamic Environments using Velocity Obstacles. *International Journal of Robotics Research*, volume 17:pp. 760–772 (1998).
URL Link Cited on p. 45

Fraichard T. Trajectory planning in dynamic workspace: a "state-time space" approach. *Advanced Robotics 13*, volume 6(8):pp. 75–94 (1999).
URL Link Cited on p. 55

Gamebase-USA. Gamebryo.
http://www.gamebryo.com/index.php (2014).
URL Link Cited on p. 103, 104

Girard M. and Maciejewski A. Computational modeling for the computer animation of legged figures. In *SIGGRAPH*, pp. 263–270. ACM (1985).
URL Link Cited on p. 66

Goalem. Golaem Crowd.
http://www.golaem.com/content/products/golaem-crowd/overview (2014).
URL Link Cited on p. 102

Gochev K., Cohen B., Butzke J., Safonova A. and Likhachev M. Path planning with adaptive dimensionality. In *Fourth Annual Symposium on Combinatorial Search* (2011).
URL Link Cited on p. 55, 121

González García F., Paradinas T., Coll N. and Patow G. *Cages:: A multilevel, multi-cage-based system for mesh deformation. *ACM Trans. Graph.*, volume 32(3):pp. 24:1–24:13 (2013).
URL Link Cited on p. 76

Grassia F. *Believable Automatically Synthesized Motion by Knowledge-enhanced Motion Transformation*. Research paper. School of Computer Science,

Carnegie Mellon University (2000).
URL Link Cited on p. 68

Grochow K., Martin S.L., Hertzmann A. and Popović Z. Style-based inverse kinematics. *ACM Trans. Graph.*, volume 23:pp. 522–531 (2004).
URL Link Cited on p. 63

Gu Q. and Deng Z. Context-Aware Motion Diversification for Crowd Simulation. *Computer Graphics and Applications, IEEE*, volume PP(99):p. 1 (2010).
URL Link Cited on p. 66

Hart P., Nilsson N. and Raphael B. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". *SIGART Bull.*, (37):pp. 28–29 (1972).
URL Link Cited on p. 53

Heck R. and Gleicher M. Parametric Motion Graphs. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, I3D '07, pp. 129–136. ACM, New York, NY, USA (2007).
URL Link Cited on p. 60, 68

Hejl J. Hardware Skinning with Quaternions. In A. Kirmse (Editor), *Game Programming Gems 4*, pp. 487–495. Charles River Media (2004).
URL Link Cited on p. 75

Helbing D., Farkas I. and Vicsek T. Simulating Dynamical Features of Escape Panic. *Nature*, volume 407:pp. 487–490 (2000).
URL Link Cited on p. 44, 45, 51

Henderson L. The statistics of crowd fluids. *Nature*, volume 229:pp. 381–383 (1971).
URL Link Cited on p. 43

Hernández B. and Rudomin I. A Rendering Pipeline for Real-Time Crowds. In W. Engel (Editor), *GPU Pro 2*, pp. 369–383. A K Peters (2011).
URL Link Cited on p. 88, 91

HMS. Center for Human Modeling and Simulation.
http://cg.cis.upenn.edu/hms/ (2014).
URL Link Cited on p. 111

Hoff K. I., Culver T., Keyser J., Lin M. and Manocha D. Interactive motion planning using hardware-accelerated computation of generalized Voronoi diagrams. In *ICRA*, volume 3, pp. 2931–2937 vol.3 (2000).
URL Link Cited on p. 55

Holte R., Perez M., Zimmer R. and MacDonald A. Hierarchical A *: searching abstraction hierarchies efficiently. In *National conference on Artificial intelligence*, AAAI, pp. 530–535. AAAI Press (1996).
URL Link Cited on p. 55

Holte R., Grajkowski J. and Tanner B. Hierarchical Heuristic Search Revisited. In *Abstraction, Reformulation and Approximation*, volume 3607 of *LNCS*, pp. 121–133. Springer Berlin Heidelberg (2005).
URL Link Cited on p. 55

Hoogendoorn S. Pedestrian Travel Behavior Modeling. In *IN 10TH INTERNATIONAL CONFERENCE ON TRAVEL BEHAVIOR RESEARCH, LUCERNE*, pp. 507–535 (2003).
URL Link Cited on p. 43

Hoppe H. Progressive Meshes. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, pp. 99–108. ACM, New York, NY, USA (1996).
URL Link Cited on p. 89

Hsu D., Kindel R., Latombe J. and Rock S. Randomized Kinodynamic Motion Planning with Moving Obstacles. *The International Journal of Robotics Research*, volume 21(3):pp. 233–255 (2002).
URL Link Cited on p. 55

Huang F.C., Chen B.Y. and Chuang Y.Y. Progressive Deforming Meshes Based on Deformation Oriented Decimation and Dynamic Connectivity Updating. In *Proceedings of the 2006 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '06, pp. 53–62. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland (2006).
URL Link Cited on p. 89

Ikemoto L., Arikan O. and Forsyth D. Knowing when to put your foot down. In *In I3D*, pp. 49–53. Press (2006).
URL Link Cited on p. 61, 62

Jacobson A., Deng Z., Kavan L. and Lewis J. Skinning: Real-time Shape Deformation. In *ACM SIGGRAPH 2014 Courses* (2014).
URL Link Cited on p. 73

Jarabo A., Van Eyck T., Sundstedt V., Bala K., Gutierrez D. and O'Sullivan C. Crowd Light: Evaluating the Perceived Fidelity of Illuminated Dynamic Scenes. *Computer Graphics Forum (Proc. EUROGRAPHICS 2012)*, volume 31(2) (2012).
URL Link Cited on p. 95

Jimenez J., Echevarria J.I., Oat C. and Gutierrez D. *GPU Pro 2*, chapter Practical and Realistic Facial Wrinkles Animation, pp. 15–27. AK Peters Ltd. (2011).
URL Link Cited on p. 73

Johansen R. Automated Semi-Procedural Animation. *Master Thesis* (2009).
URL Link Cited on p. 65, 69, 71, 127, 166, 167, 172

Kallmann M. Shortest paths with arbitrary clearance from navigation meshes. In *ACM SIGGRAPH/Eurographics SCA*, pp. 159–168 (2010).
URL Link Cited on p. 57

Kapadia M., Wang M., Reinman G. and Faloutsos P. Improved Benchmarking for Steering Algorithms. In J. Allbeck and P. Faloutsos (Editors), *Motion in Games*, volume 7060 of *Lecture Notes in Computer Science*, pp. 266–277. Springer Berlin Heidelberg (2011a).
URL Link Cited on p. 138

Kapadia M., Wang M., Singh S., Reinman G. and Faloutsos P. Scenario space: characterizing coverage, quality, and failure of steering algorithms. In *ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA, pp. 53–62 (2011b).
URL Link Cited on p. 123

Kapadia M., Beacco A., Garcia F., Reddy V., Pelechano N. and Badler N. Multi-Domain Real-time Planning in Dynamic Environments. In *Proceedings of the 2013 ACM SIGGRAPH/EUROGRAPHICS Symposium on Computer Animation*, SCA (2013).
URL Link Cited on p. 111, 119, 148

Kavan L. and Žára J. Spherical Blend Skinning: A Real-time Deformation of Articulated Models. In *Proceedings of the 2005 Symposium on Interactive 3D*

Graphics and Games, I3D '05, pp. 9–16. ACM, New York, NY, USA (2005).
URL Link Cited on p. 75

Kavan L., Collins S., Žára J. and O'Sullivan C. Skinning with Dual Quaternions. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, I3D '07, pp. 39–46. ACM, New York, NY, USA (2007).
URL Link Cited on p. 75

Kavan L., Collins S., Žára J. and O'Sullivan C. Geometric Skinning with Approximate Dual Quaternion Blending. *ACM Trans. Graph.*, volume 27(4):pp. 105:1–105:23 (2008a).
URL Link Cited on p. 75

Kavan L., Dobbyn S., Collins S., Žára J. and O'Sullivan C. Polypostors: 2D polygonal impostors for 3D crowds. In *I3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games*, pp. 149–155. ACM, New York, NY, USA (2008b).
URL Link Cited on p. 85, 86, 99, 232

Kim K., Grundmann M., Shamir A., Matthews I., Hodgins J. and Essa I. Motion fields to predict play evolution in dynamic sport scenes. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pp. 840–847 (2010).
URL Link Cited on p. 51

Kim M., Hwang Y., Hyun K. and Lee J. Tiling Motion Patches. pp. 117–126 (2012).
URL Link Cited on p. 59

Kircher S. and Garland M. Progressive Multiresolution Meshes for Deforming Surfaces. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '05, pp. 191–200. ACM, New York, NY, USA (2005).
URL Link Cited on p. 89

Klosowski J. and Silva C. Efficient Conservative Visibility Culling Using the Prioritized-Layered Projection Algorithm. *IEEE Trans. Vis. Comput. Graph.*, volume 7(4):pp. 365–379 (2001).
URL Link Cited on p. 87

Ko H. and Badler N. Animating Human Locomotion with Inverse Dynamics. *IEEE Computer Graphics & applications*, volume 16(2):pp. 50–59 (1996).
URL Link Cited on p. 66

Kobbelt L. and Botsch M. A survey of point-based techniques in computer graphics. *Comput. Graph.*, volume 28(6):pp. 801–814 (2004).
URL Link Cited on p. 77

Koenig S. and Likhachev M. D* Lite. In *National Conf. on AI*, pp. 476–483. AAAI (2002).
URL Link Cited on p. 53, 54, 146, 240

Koenig S., Likhachev M. and Furcy D. Lifelong Planning A*. *Artificial Intelligence*, volume 155(1-2):pp. 93–146 (2004).
URL Link Cited on p. 54

Kovar L. and Gleicher M. Automated extraction and parameterization of motions in large data sets. *ACM SIGGRAPH 2004 Papers on - SIGGRAPH '04*, volume 3(Siggraph):p. 559 (2004).
URL Link Cited on p. 68, 69

Kovar L., Gleicher M. and Pighin F. Motion Graphs. *ACM Transactions on Graphics*, volume 21, 3:pp. 473–482 (2002a).
URL Link Cited on p. 63

Kovar L., Gleicher M. and Schreiner J. Footskate Cleanup For Motion Capture Editing. In *In ACM SIGGRAPH Symposium on Computer Animation*, pp. 97–104 (2002b).
URL Link Cited on p. 61, 63

Kovar L., Gleicher M. and Pighin F. Motion graphs. In *ACM SIGGRAPH 2008 classes*, p. 51. ACM (2008).
URL Link Cited on p. 49

Kring A., Champandard A.J. and Samarin N. DHPA* and SHPA*: Efficient Hierarchical Pathfinding in Dynamic and Static Game Worlds. In *Sixth Artificial Intelligence and Interactive Digital Entertainment Conference* (2010).
URL Link Cited on p. 55

Lacaze A. Hierarchical planning algorithms. In *AeroSense 2002*, pp. 320–331. International Society for Optics and Photonics (2002).
URL Link Cited on p. 55

Landreneau E. and Schaefer S. Simplification of Articulated Meshes. *Computer Graphics Forum*, volume 28(2):pp. 347–353 (2009).
URL Link Cited on p. 90

Larkin M. and O'Sullivan C. Perception of Simplification Artifacts for Animated Characters. In *Proceedings of the ACM SIGGRAPH Symposium on Applied Perception in Graphics and Visualization*, APGV '11, pp. 93–100. ACM, New York, NY, USA (2011).
URL Link Cited on p. 78, 89

Lau M. and Kuffner J. Behavior planning for character animation. In *ACM SIGGRAPH/Eurographics SCA*, pp. 271–280 (2005).
URL Link Cited on p. 55

Lau M. and Kuffner J. Precomputed search trees: planning for interactive goal-driven animation. In *ACM Symposium on Computer Animation*, pp. 299–308 (2006).
URL Link Cited on p. 50, 63

Lee K., Choi M., Hong Q. and Lee J. Group Behavior from Video: A Data-driven Approach to Crowd Simulation. In *Proceedings of the 2007 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '07, pp. 109–118. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland (2007).
URL Link Cited on p. 52

Lerner A., Chrysanthou Y. and Lischinski D. Crowds by Example. *Computer Graphics Forum*, volume 26(3):pp. 655–664 (2007).
URL Link Cited on p. 52

Levine S., Lee Y., Koltun V. and Popović Z. Space-time planning with parameterized locomotion controllers. *ACM Trans. Graph.*, volume 30:pp. 23:1–23:11 (2011).
URL Link Cited on p. 55, 57

Levoy M. and Whitted T. *The Use of Points as a Display Primitive*. UNC report. University of North Carolina, Department of Computer Science (1985).
URL Link Cited on p. 77

Likhachev M., Gordon G. and Thrun S. ARA*: Anytime A* with provable bounds on sub-optimality. *Advances in Neural Information Processing Systems (NIPS)*, volume 16 (2003).
URL Link Cited on p. 54, 55

Likhachev M., Ferguson D., Gordon G., Stentz A. and Thrun S. Anytime Dynamic A*: An Anytime, Replanning Algorithm. In *ICAPS*, pp. 262–271 (2005).
URL Link Cited on p. 54, 146

Limper M., Jung Y., Behr J. and Alexa M. The POP Buffer: Rapid Progressive Clustering by Geometry Quantization. *Comput. Graph. Forum*, volume 32(7):pp. 197–206 (2013).
URL Link Cited on p. 89

Lister W., Laycock R. and Day A. A Key-Pose Caching System for Rendering an Animated Crowd in Real-Time. *Computer Graphics Forum*, volume 29(8):pp. 2304–2312 (2010).
URL Link Cited on p. 94, 95, 96, 99

Lopez T., Lamarche F. and Li T. Space-time planning in changing environments : using dynamic objects for accessibility. *CAVW*, volume 23(2):pp. 87–99 (2012).
URL Link Cited on p. 56

Lorach T. GPU Blend Shapes. *NVidia Whitepaper* (2007).
URL Link Cited on p. 76

Loscos C., Marchal D. and Meyer A. Intuitive Crowd Behaviour in Dense Urban Environments using Local Laws. In *Proceedings of the Theory and Practice of Computer Graphics 2003*, TPCG '03, pp. 122–. IEEE Computer Society, Washington, DC, USA (2003).
URL Link Cited on p. 48

Luebke D., Watson B., Cohen J., Reddy M. and Varshney A. *Level of Detail for 3D Graphics*. Elsevier Science Inc., New York, NY, USA (2002).
URL Link Cited on p. 88

Løvås G.G. Modeling and simulation of pedestrian traffic flow. *Transportation Research Part B: Methodological*, volume 28(6):pp. 429–443 (1994).
URL Link Cited on p. 43

Magnenat-Thalmann N., Laperrire R., Thalmann D. and Montréal U.D. Joint-Dependent Local Deformations for Hand Animation and Object Grasping. In *In Proceedings on Graphics interface'88*, pp. 26–33 (1988).
URL Link Cited on p. 74

Maim J., Yersin B. and Thalmann D. Unique Character Instances for Crowds. *Computer Graphics and Applications, IEEE*, volume 29(6):pp. 82 –90 (2009a).
URL Link Cited on p. 86, 96, 220

Maim J., Yersin B., Thalmann D. and Pettre J. YaQ: An Architecture for Real-Time Navigation and Rendering of Varied Crowds. *Ieee Computer Graphics And Applications*, volume 29:pp. 44–53 (2009b).
URL Link Cited on p. 66

Massive. Massive Software.
http://www.http://www.massivesoftware.com (2014).
URL Link Cited on p. 102

Mattausch O., Bittner J. and Wimmer M. CHC++: Coherent Hierarchical Culling Revisited. *Comput. Graph. Forum*, volume 27(2):pp. 221–230 (2008).
URL Link Cited on p. 91

McCarthy J.M. *Introduction to Theoretical Kinematics*. MIT Press, Cambridge, MA, USA (1990).
URL Link Cited on p. 75

McDonnell R., Dobbyn S. and O'Sullivan C. LOD human representations: A comparative study. In *Proc. of the First International Workshop on Crowd Simulation*, pp. 101–115 (2005).
URL Link Cited on p. 232

McDonnell R., Dobbyn S., Collins S. and O'Sullivan C. Perceptual evaluation of LOD clothing for virtual humans. In *Proceedings of the 2006 ACM SIGGRAPH/Eurographics symposium on Computer animation*, SCA '06, pp. 117–126. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland (2006).
URL Link Cited on p. 96

McDonnell R., Larkin M., Dobbyn S., Collins S. and O'Sullivan C. Clone attack! Perception of crowd variety. *ACM Trans. Graph.*, volume 27(3):pp. 1–8 (2008).
URL Link Cited on p. 96

McDonnell R., Larkin M., Hernández B., Rudomin I. and O'Sullivan C. Eye-catching crowds: saliency based selective variation. *ACM Trans. Graph.*, volume 28(3):pp. 55:1–55:10 (2009).
URL Link Cited on p. 96, 97, 213

McLaughlin T., Cutler L. and Coleman D. Character Rigging, Deformations, and Simulations in Film and Game Production. In *ACM SIGGRAPH 2011 Courses*, SIGGRAPH '11, pp. 5:1–5:18. ACM, New York, NY, USA (2011).
URL Link Cited on p. 73

Ménardais S., Kulpa R., Multon F. and Arnaldi B. Synchronization for dynamic blending of motions. In *Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*, SCA '04, pp. 325–335. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland (2004).
URL Link Cited on p. 63

Milazzo J., Rouphail N. and Allen D.P. Effect of Pedestrians on Capacity of Signalized Intersections. *Transportation Research Record*, volume 1646:pp. 37–46 (1998).
URL Link Cited on p. 43

Millan E. and Rudomin I. A comparison between impostors and point-based models for interactive rendering of animated models. In *Proceedings of the International Conference on Computer Animation and Social Agents (CASA)*. University Press (2006a).
URL Link Cited on p. 79

Millan E. and Rudomin I. Impostors and pseudo-instancing for GPU crowd rendering. In *GRAPHITE '06: Proceedings of the 4th international conference on Computer graphics and interactive techniques in Australasia and Southeast Asia*, pp. 49–55. ACM, New York, NY, USA (2006b).
URL Link Cited on p. 93, 99

Min J. and Chai J. Motion Graphs++. *ACM Transactions On Graphics*, volume 31(6):p. 1 (2012).
URL Link Cited on p. 50

Mitchell J. and Sander P. SIGGRAPH 2004 - Real-Time Shading Course Applications of Explicit Early-Z Culling (2004).
URL Link Cited on p. 88

Mixamo. Mixamo Auto-Rigger.
https://www.mixamo.com/auto-rigger (2014).
URL Link Cited on p. 74

Mohr A. and Gleicher M. Deformation Sensitive Decimation. Technical Report, University of Wisconsin Graphics Group (2003).
URL Link Cited on p. 89

Mononen M. Recast Navigation Toolkit Webpage.
code.google.com/p/recastnavigation/ (2009).
URL Link Cited on p. 53, 113, 136, 142, 146

Mori M., MacDorman K. and Kageki N. The Uncanny Valley [From the Field]. *IEEE Robot. Automat. Mag.*, volume 19(2):pp. 98–100 (2012).
URL Link Cited on p. 34

Moussaïd M., Helbing D., Garnier S., Johansson A., Combe M. and Theraulaz G. Experimental study of the behavioural mechanisms underlying self-organization in human crowds. *Proceedings of the Royal Society B: Biological Sciences* (2009).
URL Link Cited on p. 51

Moussaïd M., Perozo N., Garnier S., Helbing D. and Theraulaz G. The Walking Behaviour of Pedestrian Social Groups and Its Impact on Crowd Dynamics. *PLoS ONE*, volume 5(4):p. e10047 (2010).
URL Link Cited on p. 51

Mukai T. and Kuriyama S. Geostatistical motion interpolation. *ACM Trans. Graph.*, volume 24:pp. 1062–1070 (2005).
URL Link Cited on p. 63, 68

Musse S., Jung C., Jacques J. and Braun A. Using Computer Vision to Simulate the Motion of Virtual Agents: Research Articles. *Comput. Animat. Virtual Worlds*, volume 18(2):pp. 83–93 (2007).
URL Link Cited on p. 52

Narain R., Golas A., Curtis S. and Lin M.C. Aggregate Dynamics for Dense Crowd Simulation. *ACM Transactions on Graphics*, volume 28 (2009).
URL Link Cited on p. 42

Ogre. Ogre: Object-Oriented Graphics Rendering Engine.
http::/www.ogre3d.org// (2014).
URL Link Cited on p. 103, 104

Oh K., Ki H. and Lee C. Pyramidal displacement mapping: a GPU based artifacts-free ray tracing through an image pyramid. In *VRST*, pp. 75–82 (2006).
URL Link Cited on p. 193, 197

Oliva R. and Pelechano N. NEOGEN: Near optimal generator of navigation meshes for 3D multi-layered environments. *Computer & Graphics*, volume 37(5):pp. 403–412 (2013).
URL Link Cited on p. 53

Oliveira M., Bishop G. and McAllister D. Relief texture mapping. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pp. 359–368. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA (2000).
URL Link Cited on p. 184, 233

Oshita M. and Ogiwara Y. Sketch-Based Interface for Crowd Animation. In *Proceedings of the 10th International Symposium on Smart Graphics*, SG '09, pp. 253–262. Springer-Verlag, Berlin, Heidelberg (2009).
URL Link Cited on p. 51

O'Sullivan C. and Ennis C. Metropolis: Multisensory Simulation of a Populated City. In *Games and Virtual Worlds for Serious Applications (VS-GAMES), 2011 Third International Conference on*, pp. 1–7 (2011).
URL Link Cited on p. 105

Pamplona V., Oliveira M. and Nedel L. *Game Programming Gems VII*, chapter Animating Relief Impostors Using Radial Basis Functions Textures, pp. 401–412. Charles River Media, Inc., Hingham, Massachusetts (2008).
URL Link Cited on p. 186, 187

Pantuwong N. and Sugimoto M. A novel template-based automatic rigging algorithm for articulated-character animation. *Computer Animation and Virtual*

Worlds, volume 23(2):pp. 125–141 (2012).
URL Link Cited on p. 74

Paris S., Pettré J. and Donikian S. Pedestrian Reactive Navigation for Crowd Simulation: a Predictive Approach. *Computer Graphics Forum*, volume 26(3):pp. 665–674 (2007).
URL Link Cited on p. 51

Park S., Shin H. and Shin S. On-line Locomotion Generation Based on Motion Blending. In *Proceedings of the 2002 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '02, pp. 105–111. ACM, New York, NY, USA (2002).
URL Link Cited on p. 68

Payan F., Hahmann S. and Bonneau G.P. Deforming surface simplification based on dynamic geometry sampling. In *Shape Modeling and Applications, 2007. SMI '07. IEEE International Conference on*, pp. 71–80 (2007).
URL Link Cited on p. 89

Pearl J. *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1984).
URL Link Cited on p. 54

Pelechano N., Allbeck J. and Badler N. Controlling Individual Agents in High-Density Crowd Simulation. *ACM SIGGRAPH / Eurographics Symposium on Computer Animation (SCA'07), San Diego (USA)* (2007).
URL Link Cited on p. 45, 46, 150

Pelechano N., Allbeck J. and Badler N. *Virtual Crowds: Methods, Simulation, and Control*. Morgan & Claypool (2008).
URL Link Cited on p. 2, 57

Pelechano N., Spanlang B. and Beacco A. A framework for rendering, simulation and animation of crowds. In *CEIG Spanish Conference on Computer Graphic* (2009).
URL Link Cited on p. 182

Pelechano N., Spanlang B. and Beacco A. Avatar Locomotion in Crowd Simulation. In *International Conference on Computer Animation and Social Agents (CASA)*. Chengdu, China (2011).
URL Link Cited on p. 182

Pelechano N., Kapadia M., Allbeck J., Chrysanthou Y., Guy S. and Badler N. Simulating heterogeneous crowds with interactive behaviors. In *EUROGRAPHICS 2014 Tutorials* (2014).
URL Link Cited on p. 107

Pellegrini S., Ess A., Tanaskovic M. and Van Gool L. Wrong turn - No dead end: A stochastic pedestrian motion model. In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2010 IEEE Computer Society Conference on*, pp. 15–22 (2010).
URL Link Cited on p. 51

Pettré J. and Laumond J.P. A Motion Capture-based Control-space Approach for Walking Mannequins: Research Articles. *Comput. Animat. Virtual Worlds*, volume 17(2):pp. 109–126 (2006).
URL Link Cited on p. 63, 64

Pettré J., Laumond J.P. and Siméon T. A 2-stages Locomotion Planner for Digital Actors. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '03, pp. 258–264. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland (2003).
URL Link Cited on p. 63

Pettré J., De Heras Ciechomski P., Maïm J., Yersin B., Laumond J. and Thalmann D. Real-time navigating crowds: scalable simulation and rendering: Research Articles. *Comput. Animat. Virtual Worlds*, volume 17(3-4):pp. 445–455 (2006).
URL Link Cited on p. 83, 88

Pettré J., Kallmann M. and Lin M. Motion Planning and Autonomy for Virtual Humans. In *ACM SIGGRAPH classes*, pp. 1–31 (2008).
URL Link Cited on p. 55

Pettré J., Ondřej J., Olivier A.H., Cretual A. and Donikian S. Experiment-based Modeling, Simulation and Validation of Interactions Between Virtual Walkers. In *Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '09, pp. 189–198. ACM, New York, NY, USA (2009).
URL Link Cited on p. 51

Policarpo F. and Oliveira M. Relief mapping of non-height-field surface details. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, I3D

'06, pp. 55–62 (2006).
URL Link Cited on p. 197

Policarpo F. and Oliveira M. Relaxed cone stepping for relief mapping. In *GPU Gems 3: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, pp. 409–428. Addison-Wesley Professional (2007).
URL Link Cited on p. 186

Policarpo F., Oliveira M. and Comba J. Real-time relief mapping on arbitrary polygonal surfaces. *ACM Trans. Graph.*, volume 24(3):pp. 935–935 (2005).
URL Link Cited on p. 186, 193, 197

Popović Z. and Witkin A. Physically Based Motion Transformation. *SIGGRAPH '99* (1999).
URL Link Cited on p. 58

Pratt D., Pratt S., Barham P., Barker R., Waldrop M., Ehlert J. and Chrislip C. Humans in Large-scale, Networked Virtual Environments. *Presence*, volume 6(5):pp. 547–564 (1997).
URL Link Cited on p. 88, 99

RAD-Game-Tools. Granny 3D.
http://www.radgametools.com/granny.html (2014).
URL Link Cited on p. 103

Ramirez J., Lligadas X. and Susin A. Automatic Adjustment of Rigs to Extracted Skeletons. In F. Perales and R. Fisher (Editors), *Articulated Motion and Deformable Objects*, volume 5098 of *Lecture Notes in Computer Science*, pp. 409–418. Springer Berlin Heidelberg (2008).
URL Link Cited on p. 74

Ren C., Zhao L. and Safonova A. Human Motion Synthesis with Optimization-Based Graphs. *Proceedings of Eurographics, CGF*, volume 29(2) (2010).
URL Link Cited on p. 50

Reynolds C. Flocks, Herds, and Schools: A Distributed Behavioral Model. In *Computer Graphics*, pp. 25–34 (1987).
URL Link Cited on p. 43, 44

Reynolds C. Steering Behaviors For Autonomous Characters. In *Proceedings of Game Developers Conference 1999*, GDC '99, pp. 763–782. Miller Freeman

Game Group, San Francisco, California (1999).
URL Link Cited on p. 43, 44, 242, 250

Reynolds C. Big Fast Crowds on PS3. In *Proceedings of the 2006 ACM SIGGRAPH Symposium on Videogames*, Sandbox '06, pp. 113–121. ACM, New York, NY, USA (2006).
URL Link Cited on p. 87

Reynolds C. OpenSteer: Steering Behaviors for Autonomous Characters. http://opensteer.sourceforge.net/ (2014).
URL Link Cited on p. 102

Rose C., Sloan P.P.J. and Cohen M. Artist-Directed Inverse-Kinematics Using Radial Basis Function Interpolation. *Comput. Graph. Forum*, volume 20(3):pp. 239–250 (2001).
URL Link Cited on p. 68

Rossignac J. and Borrel P. Multi-resolution 3D approximations for rendering complex scenes. In B. Falcidieno and T.L. Kunii (Editors), *Modeling in Computer Graphics*, IFIP Series on Computer Graphics, pp. 455–465. Springer (1993).
URL Link Cited on p. 89

Safonova A. and Hodgins J. Construction and optimal search of interpolated motion graphs. In *ACM SIGGRAPH* (2007).
URL Link Cited on p. 55, 67

Schmalstieg D. and Fuhrmann A. Coarse View-Dependent Levels of Detail for Hierarchical and Deformable Models. Technical Report, Vienna University of Technology (1999).
URL Link Cited on p. 89

Schroders M. and Gulik R. Quadtree relief mapping. In *Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pp. 61–66. ACM (2006).
URL Link Cited on p. 186, 197

Sewall J., Wilkie D., Merrell P. and Lin M.C. Continuum Traffic Simulation. *Computer Graphics Forum*, volume 29:pp. 439–448 (2010).
URL Link Cited on p. 42

Shao W. and Terzopoulos D. Autonomous Pedestrians. *ACM SIGGRAPH*, pp. 19–28 (2005).
URL Link Cited on p. 44

Shapiro A. SmartBody, University of Southern California Institute for Creative Technologies.
http://smartbody.ict.usc.edu/ (2014).
URL Link Cited on p. 105

Shapiro A., Kallman M. and Faloutsos P. Interactive Motion Correction and Object Manipulation. In *ACM SIGGRAPH I3D* (2007).
URL Link Cited on p. 55

Shoulson A., Marshak N., Kapadia M. and Badler N. ADAPT : The Agent Development and Prototyping Testbed. *ACM SIGGRAPH I3D* (2013).
URL Link Cited on p. 105, 143, 171

Singh S., Kapadia M., Faloutsos P. and Reinman G. An Open Framework for Developing, Evaluating, and Sharing Steering Algorithms. In *Proceedings of the 2nd International Workshop on Motion in Games*, MIG '09, pp. 158–169. Springer-Verlag, Berlin, Heidelberg (2009).
URL Link Cited on p. 103

Singh S., Kapadia M., Reinman G. and Faloutsos P. Footstep navigation for dynamic crowds. *Computer Animation and Virtual Worlds*, volume 22(2-3):pp. 151–158 (2011).
URL Link Cited on p. 48, 49, 55, 66, 176

Southern R. and Gain J. Creation and Control of Real-time Continuous Level of Detail on Programmable Graphics Hardware. *Computer Graphics Forum*, volume 22:pp. 35–48 (2003).
URL Link Cited on p. 92

Spanlang B. HALCA Hardware Acclerated Library for Character Animation. Technical Report, EVENT Lab. Universitat de barcelona (2009).
URL Link Cited on p. 103, 152, 182, 188, 204, 222, 235, 245, 247, 250

Sturtevant N. and Geisberger R. A comparison of high-level approaches for speeding up pathfinding. *Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, pp. 76–82 (2010).
URL Link Cited on p. 55

Sueda S., Kaufman A. and Pai D.K. Musculotendon Simulation for Hand Animation. *ACM Trans. Graph. (Proc. SIGGRAPH)*, volume 27(3) (2008).
URL Link Cited on p. 73

Sung M., Kovar L. and Gleicher M. Fast and accurate goal-directed motion synthesis for crowds. In *ACM SIGGRAPH/Eurographics SCA*, pp. 291–300 (2005).
URL Link Cited on p. 57

Tatarchuk N. Dynamic parallax occlusion mapping with approximate soft shadows. In *I3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pp. 63–69. ACM, New York, NY, USA (2006).
URL Link Cited on p. 186

Tatarchuk N., Barczak J. and Purnomo B. GPU Tessellation for Detailed, Animated Crowds. Technical Report, AMD, Inc. (2008).
URL Link Cited on p. 90, 91

Tecchia F. and Chrysanthou Y. Real-Time Rendering of Densely Populated Urban Environments. In *Proceedings of the Eurographics Workshop on Rendering Techniques 2000*, pp. 83–88. Springer-Verlag, London, UK (2000).
URL Link Cited on p. 81, 99, 257

Tecchia F., Loscos C., Conroy R. and Chysanthou Y. Agent behavior simulator (ABS): A Platform for Urban Behavior Development. *ACM/EG Games Technology Conference* (2001).
URL Link Cited on p. 48

Tecchia F., Loscos C. and Chrysanthou Y. Image-Based Crowd Rendering. *IEEE Comput. Graph. Appl.*, volume 22(2):pp. 36–43 (2002).
URL Link Cited on p. 81, 82, 95, 96, 232

Thalmann D., Musse S.R. and Kallmann M. Virtual Humans' Behaviour: Individuals, Groups, and Crowds. In *Digital Media: The Future* (1999).
URL Link Cited on p. 44

Toledo L., De Gyves O. and Rudomín I. Hierarchical level of detail for varied animated crowds. *The Visual Computer*, volume 30(6-8):pp. 949–961 (2014).
URL Link Cited on p. 78, 79, 87, 92, 99

Torkos N. and van de Panne M. Footprint-based Quadruped Motion Synthesis. In *In Graphics Interface'98, 151-160. ISBN*, pp. 0–9695338 (1998).
URL Link Cited on p. 66

Torrens P. Behavioral Intelligence for Geospatial Agents in Urban Environments. In *Proceedings of the 2007 IEEE/WIC/ACM International Conference on Intelligent Agent Technology*, IAT '07, pp. 63–66. IEEE Computer Society, Washington, DC, USA (2007).
URL Link Cited on p. 48

Treuille A., Cooper S. and Popović Z. Continuum Crowds:. *ACM Transactions on Graphics SIGGRAPH*, pp. 1160–1168 (2006).
URL Link Cited on p. 47

Treuille A., Lee Y. and Popović Z. Near-Optimal Character Animation with Continuous Control. *ACM Transactions on Graphics*, volume 26(3):p. 7 (2007).
URL Link Cited on p. 4, 65

Ulicny B., Ciechomski P.d.H. and Thalmann D. Crowdbrush: interactive authoring of real-time crowd scenes. In *Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*, SCA '04, pp. 243–252. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland (2004).
URL Link Cited on p. 77, 88, 99

Unity. Unity: Game Engine.
http::/unity3d.com/ (2014).
URL Link Cited on p. 103, 104, 143, 176

University C.M. CMU Graphics Lab Motion Capture Database.
http://mocap.cs.cmu.edu/ (2013).
URL Link Cited on p. 61, 132

Vaillant R., Barthe L., Guennebaud G., Cani M., Rohmer D., Wyvill B., Gourmel O. and Paulin M. Implicit skinning: real-time skin deformation with contact modeling. *ACM Trans. Graph.*, volume 32(4):pp. 125:1–125:12 (2013).
URL Link Cited on p. 76

van Basten B. and Egges A. Evaluating distance metrics for animation blending. In *Proceedings of the 4th International Conference on Foundations of Digital*

Games, FDG '09, pp. 199–206. ACM, New York, NY, USA (2009).
URL Link Cited on p. 71, 134

van Basten B., Peeters P. and Egges A. The Step Space : Example-Based Footprint-Driven Motion Synthesis. *Computer Animation and Virtual Worlds*, volume 21(May):pp. 433–441 (2010).
URL Link Cited on p. 68, 70

van Basten B., Stüvel S. and Egges A. A Hybrid Interpolation Scheme for Footprint-Driven Walking Synthesis. *Graphics Interface*, pp. 9–16 (2011).
URL Link Cited on p. 50, 68, 69, 70

van de Panne M. From Footprints to Animation. *Computer Graphics Forum*, volume 16(4):pp. 211–223 (1997).
URL Link Cited on p. 66

van den Berg J., Ferguson D. and Kuffner J. Anytime path planning and replanning in dynamic environments. In *ICRA*, pp. 2366 –2371 (2006).
URL Link Cited on p. 55

van den Berg J., Patil S., Sewall J., Manocha D. and Lin M. Interactive navigation of multiple agents in crowded environments. In *ACM SIGGRAPH I3D*, pp. 139–147 (2008).
URL Link Cited on p. 45, 46

van den Berg J., Guy S., Snape J., Lin M. and Manocha D. RVO2 Library: Reciprocal Collision Avoidance for Real-Time Multi-Agent Simulation.
http://http://gamma.cs.unc.edu/RVO2/ (2014).
URL Link Cited on p. 102, 250

van Toll W., Cook A. and Geraerts R. Real-time density-based crowd simulation. *CAVW*, volume 23(1):pp. 59–69 (2012).
URL Link Cited on p. 113, 146

Wand M. and Straßer W. Multi-Resolution Rendering of Complex Animated Scenes. *Computer Graphics Forum*, volume 21(3):pp. 483–491 (2002).
URL Link Cited on p. 78, 99

Williams L. Casting curved shadows on curved surfaces. *SIGGRAPH Comput. Graph.*, volume 12(3):pp. 270–274 (1978).
URL Link Cited on p. 95

Willmott A. Rapid Simplification of Multi-attribute Meshes. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG '11, pp. 151–158. ACM, New York, NY, USA (2011).
URL Link Cited on p. 90

Wimmer M. and Bittner J. Hardware Occlusion Queries Made Useful. In M. Pharr and R. Fernando (Editors), *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley (2005).
URL Link Cited on p. 87

Winkler T., Drieseberg J., Alexa M. and Hormann K. Multi-Scale Geometry Interpolation. *Computer Graphics Forum*, volume 29(2):pp. 309–318 (2010). Proceedings of Eurographics.
URL Link Cited on p. 76

Witkin A. and Popovic Z. Motion warping. In *ACM SIGGRAPH*, pp. 105–108 (1995).
URL Link Cited on p. 50, 63

Wolfram S. Statistical mechanics of cellular automata. *Reviews of Modern Physics*, volume 55(3):pp. 601–644 (1983).
URL Link Cited on p. 48

Wu C. and Zordan V. Goal-directed stepping with momentum control. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '10, pp. 113–118. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland (2010).
URL Link Cited on p. 63, 67

Yee Y. and Newman A. A perceptual metric for production testing. In *SIGGRAPH'04: ACM SIGGRAPH 2004 Sketches*, p. 121. ACM, New York, NY, USA (2004).
URL Link Cited on p. 202

Yin K., Loken K. and Van de Panne M. SIMBICON: Simple Biped Locomotion Control. *ACM Trans. Graph.*, volume 26(3):p. Article 105 (2007).
URL Link Cited on p. 63

Yuksel K., Yucebilgin A., Balcisoy S. and Ercil A. Real-time feature-based image morphing for memory-efficient impostor rendering and animation on GPU.

The Visual Computer, volume 29(2):pp. 131–140 (2013).
URL Link Cited on p. 82

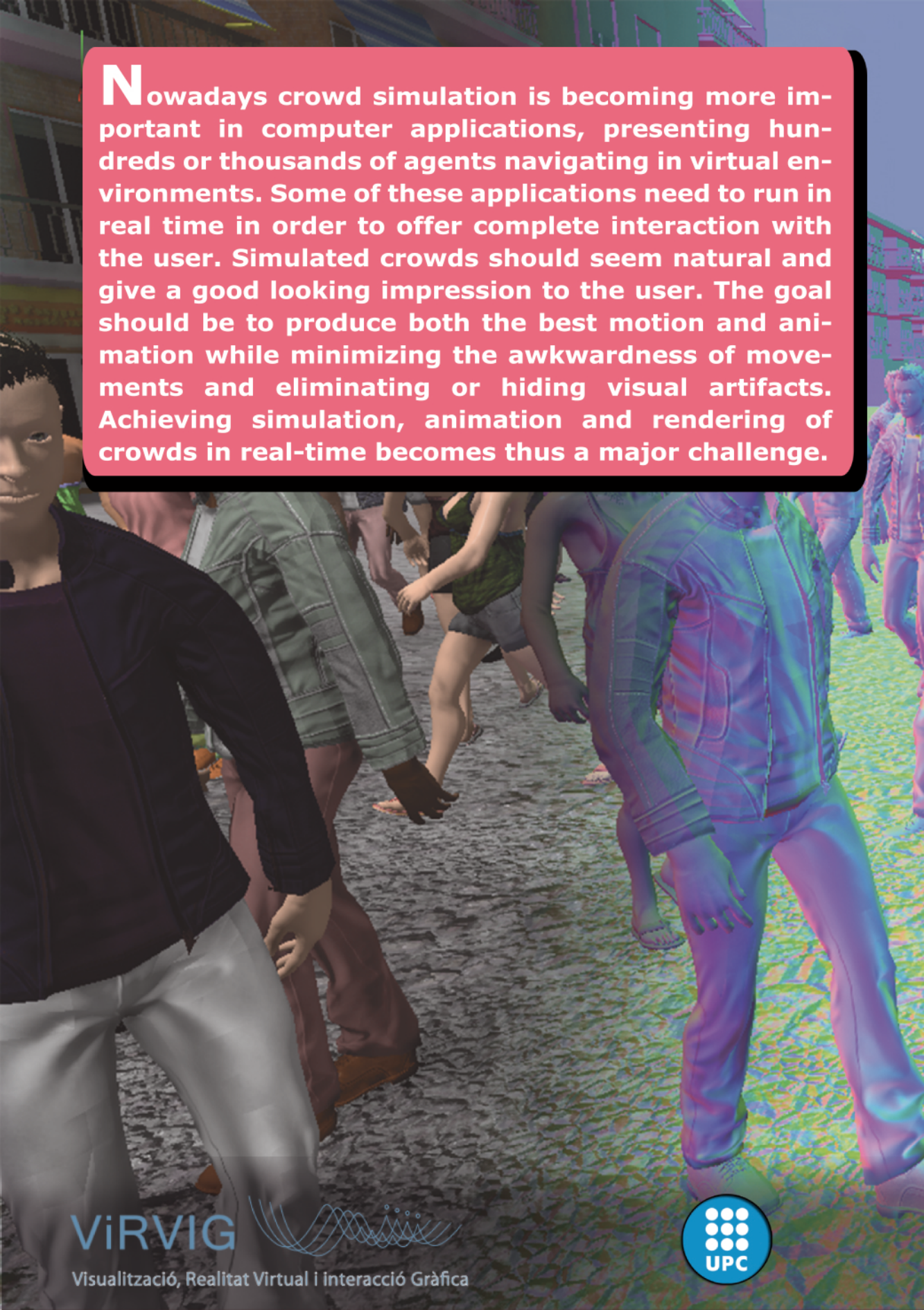
Zach C., Mantler S. and Karner K. Time-critical rendering of discrete and continuous levels of detail. In J. Shi, L.F. Hodges, H. Sun and Q. Peng (Editors), *The ACM symposium on Virtual reality software and technology (VRST)*, pp. 1–8. ACM (2002).
URL Link Cited on p. 91

Zhang S. and Wu E. Deforming Surface Simplification Based on Feature Preservation. In L. Ma, M. Rauterberg and R. Nakatsu (Editors), *Entertainment Computing - ICEC 2007*, volume 4740 of *Lecture Notes in Computer Science*, pp. 139–149. Springer Berlin Heidelberg (2007).
URL Link Cited on p. 89

Zhao L. and Safanova A. Achieving Good Connectivity in Motion Graphs. *ACM SIGGRAPH / Eurographics Proc. Symp. on Computer Animation (SCA). ACM Press* (2007).
URL Link Cited on p. 63, 64

Zhao L. and Safonova A. Achieving good connectivity in motion graphs. *Graphical Models*, volume 71(4):pp. 139–152 (2009).
URL Link Cited on p. 50

Zwicker M., Pfister H., van Baar J. and Gross M. Surface splatting. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '01, pp. 371–378. ACM, New York, NY, USA (2001).
URL Link Cited on p. 77



Nowadays crowd simulation is becoming more important in computer applications, presenting hundreds or thousands of agents navigating in virtual environments. Some of these applications need to run in real time in order to offer complete interaction with the user. Simulated crowds should seem natural and give a good looking impression to the user. The goal should be to produce both the best motion and animation while minimizing the awkwardness of movements and eliminating or hiding visual artifacts. Achieving simulation, animation and rendering of crowds in real-time becomes thus a major challenge.

ViRVIG
Visualització, Realitat Virtual i interacció Gràfica
UPC