Scalable exploration of 3D massive models

Alberto Jaspe Villanueva

Doctor of Philosophy (Ph.D.) thesis 2018

Supervisors	Dr. Enrico Gobbetti
	Visual Computing Director, CRS4
	Dr. Julián Dorado de la Calle
	Full Professor, UDC
Tutor	Dr. Julián Dorado de la Calle

UNIVERSIDADE DA CORUÑA

PhD Programme in Information and Communications Technology

Scalable exploration of 3D massive models
Doctor of Philosophy (Ph.D.) thesis 2018
PhD Programme in Information and Communications Technology

Author:	Alberto Jaspe Villanueva
Supervisors:	Dr. Enrico Gobbetti (CRS4)
	Dr. Julián Dorado de la Calle (University of A Coruña)
Tutor:	Dr. Julián Dorado de la Calle (University of A Coruña)
Reviewers:	Oliver Staadt (University of Rostock)
	Rafael Ballester (University of Zürich)

University of A Coruña
Department of Computer Science
Faculty of Computer Science
Campus de Elviña, S//N
15071 – A Couruña, Spain

[illegible] [illegible] [illegible]

Alberto Jaspe Villanueva   Dr. Enrico Gobbetti   Dr. Julián Dorado de la Calle

A Coruña (Spain) and Cagliari (Italy), September 2018.

Dr. D. Julián Dorado de la Calle, Catedrático del Departamento de Tecnologías de la Información y las Comunicaciones, Universidade da Coruña (UDC), España.

Dr. Mr. Julián Dorado de la Calle, Full Porfessor of the Information Technologies Department, University of A Coruña (UDC), Spain.

Dr. D. Enrico Gobbetti, director del grupo de Visual Computing en el Centro de Estudios Avanzados, Investigación y Desarrollo de Cerdeña (CRS4), Italia.

Dr. Mr. Enrico Gobbetti, director of the Visual Computing Group at the Center for Advanced Studies, Research and Development in Sardinia (CRS4), Italy.

Atestan

Attest

Que la memoria titulada "**Scalable Exploration of 3D Massive Models**" presentada por **Alberto Jaspe Villanueva**, ha sido realizada bajo nuestra dirección. Considerando que el trabajo constituye tema de Tesis Doctoral, se autoriza su presentación en la Universidade da Coruña.

That the dissertation entitled "Scalable Exploration of 3D massive models" presented by Alberto Jaspe Villanueva, has been developed under our advising. Considering that the work is subject of Doctoral Thesis, we authorize its presentation at the University of A Coruña.

Y para que así conste, se expide el presente certificado en A Coruña (España) y Cagliari (Italia), en Septiembre del 2018.

And for the record, this certificate is issued in A Coruña (Spain) and Cagliari (Italy) in September 2018.

[illegible] [illegible]

Fdo. Dr. D. Enrico Gobbetti Fdo. Dr. D. Julián Dorado de la Calle

A mi familia, en vías de expansión

To my growing family

It's not the note you play that's the wrong note; it's the note you play afterwards that makes it right or wrong.

— **Miles Davis**

Acknowledgements

I would like to thank a number of people who supported me, both professionally and personally, on my journey to complete this work.

In first place, I would like to thank Enrico Gobbetti, who has been my research mentor during this time and the main supervisor of this thesis, as well as Julian Dorado de la Calle for co-supervising, tutoring and mentoring me. Their wisdom, experience and clarity of thought has always been a precious guide, and it will hopefully be for long in the future.

I also want to specially thank to all the co-authors of the publications this work is based on. Fabio Bettio, Jiří Bittner, Enrico Gobbetti, Fabio Marton, Oliver Mattausch, Emilio Merella, Omar. A. Mures, Emilio J. Padrón, Renato Pajarola, Ruggero Pintus, Juan R. Rabuñal, and Michael Wimmer. Also, during this period I had the chance to work in other publications with Marco Agus, Marcos Balsa, Marco di Benedetto, Fabio Ganovelli, Luis Hernández, Claudio Mura, Giovanni Pintore, Javier Taibo, Roberto Scopigno, Antonio Seoane, and Antonio Zorcolo. I feel so fortunate to have had the chance of learning from these great researchers from many institutions around the world.

I am also very grateful for the great effort and willingness to help shown to me by the reviewers of this thesis, Oliver Staadt and Rafael Ballester-Ripoll. Further, I would like to thank also Renato Pajarola, not only for co-leading the Marie Curie DIVA ITN together with Enrico, but also for hosting me at the VMML group at the University of Zurich (UZH) which he directs.

All my gratitude and friendship to the present and past members of (or related to) the Visual Computing Group of CRS4: Anto, Cinzia, Claudio, Emilio, Enrico, Fabio B. Fabio M., Gianluigi, Gianni, Jalley, Jose, Katia, Luca, Magus, Marcos, Matteo, Robi,

and Ru. You have always made me feel at home and taught me so much (and not just about Computer Graphics). I also want to mention my former group, VideaLab where I got very nice experiences developing Computer Graphics projects aside so talented people, as well as RNASA lab, specially Fran Cedrón who has remotely assisted me so much.

I am also so grateful for the many long scientific, tech and nerd discussions with Taibo, Ryu and Jalley. Talking to you is always a motivational boost.

Many people have supported me during these years, some old, faraway friends and other new and closer ones. You are a lot, and I feel very lucky to have you. I just want to mention some who have specially pulled me in the most stormy times: RauPau, Firinu, Maje, Franci, i compari Luca e Vale, Pierino, ViCs, Annalisa, Lili, Braisa, Javier y Gabi, Ryu, Michi, Charli, Noe, Rubén, Ada, José and the whole Orthinat clan, Mari, Cate, Vale, Eve, as well as so many others that I am probably forgetting. Thanks guys, your friendship is a treasure.

I have nothing but love and gratitude to my entire family. Not only my parents, Jose and Vicky, but my sister, brother, and the whole tribe, while far away they are always so close to me. And also to the Pau-Massidda family, especially Cinzia and Antonello, who have treated me like a son from the outset.

Finally and most important, Manu, thank you for being my favorite person and my ultimate supporter. I love you and the alien growing inside of you. And to this little guy, I never, ever would have found the energy to finish this work without your arrival. You are not born yet, but you have already changed my life for the better. Thanks and see you soon!

Graciñas, y gracias, e grazie, e gràtzias and thanks.

Cagliari, September 2018 *Alberto Jaspe Villanueva*

Resumo

Esta tese presenta unha serie técnicas escalables que avanzan o estado da arte da creación e exploración de grandes modelos tridimensionaies. No ámbito da xeración destes modelos, preséntanse métodos para mellorar a adquisición e procesado de escenas reais, grazas a unha implementación eficiente dun sistema out- of- core de xestión de nubes de puntos, e unha nova metodoloxía escalable de fusión de datos de xeometría e cor para adquisicións con oclusións. No ámbito da visualización de grandes conxuntos de datos, que é o núcleo principal desta tese, preséntanse dous novos métodos. O primeiro é unha técnica adaptabile out-of-core que aproveita o hardware de rasterización da GPU e as occlusion queries para crear lotes coherentes de traballo, que serán procesados por kernels de trazado de raios codificados en shaders, permitindo out-of-core ray-tracing con sombreado e iluminación global. O segundo é un método de compresión agresivo que aproveita a redundancia xeométrica que se adoita atopar en grandes modelos 3D para comprimir os datos de forma que caiban, nun formato totalmente renderizable, na memoria da GPU. O método está deseñado para representacións voxelizadas de escenas 3D, que son amplamente utilizadas para diversos cálculos como para acelerar as consultas de visibilidade na GPU. A compresión lógrase fusionando subárbores idénticas a través dunha transformación de similitude, e aproveitando a distribución non homoxénea de referencias a nodos compartidos para almacenar punteiros aos nodos fillo, e utilizando unha codificación de bits variable. A capacidade e o rendemento de todos os métodos avalíanse utilizando diversos casos de uso do mundo real de diversos ámbitos e sectores, incluídos o patrimonio cultural, a enxeñería e os videoxogos.

Palabras crave: Gráficos por Computador, Síntese de Imaxen Escalable, Algoritmos Out-of-core, Trazado de Raios, Voxels, Niveis de detalle.

Resumen

En esta tesis se presentan una serie técnicas escalables que avanzan el estado del arte de la creación y exploración de grandes modelos tridimensionales. En el ámbito de la generación de estos modelos, se presentan métodos para mejorar la adquisición y procesado de escenas reales, gracias a una implementación eficiente de un sistema *out-of-core* de gestión de nubes de puntos, y una nueva metodología escalable de fusión de datos de geometría y color para adquisiciones con oclusiones. Para la visualización de grandes conjuntos de datos, que constituye el núcleo principal de esta tesis, se presentan dos nuevos métodos. El primero de ellos es una técnica adaptable *out-of-core* que aprovecha el hardware de rasterización de la GPU y las *occlusion queries*, para crear lotes coherentes de trabajo, que serán procesados por *kernels* de trazado de rayos codificados en *shaders*, permitiendo *renders out-of-core* avanzados con sombreado e iluminación global. El segundo es un método de compresión agresivo, que aprovecha la redundancia geométrica que se suele encontrar en grandes modelos 3D para comprimir los datos de forma que quepan, en un formato totalmente renderizable, en la memoria de la GPU. El método está diseñado para representaciones voxelizadas de escenas 3D, que son ampliamente utilizadas para diversos cálculos como la aceleración las consultas de visibilidad en la GPU o el trazado de sombras. La compresión se logra fusionando subárboles idénticos a través de una transformación de similitud, y aprovechando la distribución no homogénea de referencias a nodos compartidos para almacenar punteros a los nodos hijo, utilizando una codificación de bits variable. La capacidad y el rendimiento de todos los métodos se evalúan utilizando diversos casos de uso del mundo real de diversos ámbitos y sectores, incluidos el patrimonio cultural, la ingeniería y los videojuegos.

Palabras clave: Gráficos por Computador, Síntesis de Imagen Escalable, Algoritmos *Out-of-core*, Trazado de Rayos, Voxels, Niveles de detalle.

Abstract

This thesis introduces scalable techniques that advance the state-of-the-art in massive model creation and exploration. Concerning model creation, we present methods for improving reality-based scene acquisition and processing, introducing an efficient implementation of scalable out-of-core point clouds and a data-fusion approach for creating detailed colored models from cluttered scene acquisitions. The core of this thesis concerns enabling technology for the exploration of general large datasets. Two novel solutions are introduced. The first is an adaptive out-of-core technique exploiting the GPU rasterization pipeline and hardware occlusion queries in order to create coherent batches of work for localized shader-based ray tracing kernels, opening the door to out-of-core ray tracing with shadowing and global illumination. The second is an aggressive compression method that exploits redundancy in large models to compress data so that it fits, in fully renderable format, in GPU memory. The method is targeted to voxelized representations of 3D scenes, which are widely used to accelerate visibility queries on the GPU. Compression is achieved by merging subtrees that are identical through a similarity transform and by exploiting the skewed distribution of references to shared nodes to store child pointers using a variable bit-rate encoding The capability and performance of all methods are evaluated on many very massive real-world scenes from several domains, including cultural heritage, engineering, and gaming.

Keywords: Computer Graphics, Scalable Rendering, Out-of-core algorithms, Ray-tracing, Voxels, Level-of-detail.

Preface

This thesis represents a partial summary of the work done between 2014 and 2018, mainly with the Visual Computing Group of the CRS4 (Center for Advanced Studies, Research and Development in Sardinia, Italy) under the direct supervision of its director Dr. Enrico Gobbetti, who I want to thank for offering me the unique opportunity to be part of his research group. During this time I also attended the Ph.D. Program in Information and Communications Technology under the kind tutoring and also advising of Dr. Julián Dorado de la Calle, director of SABIA group, and full professor at the University of A Coruña (Spain), who I thank as well. Without both their dedication and guidance this work would not have been possible.

I developed the main work of this Ph.D. thesis in the framework of the *Data Intensive Visualization and Analysis (DIVA) Project* [1], an Initial Training Network (ITN) funded by the EU (Marie Curie Actions of the European Union's Seventh Framework Programme). From 2012 to 2016 it brought together six full partner institutions, namely, the University of Zurich (UZH), the CRS4, the University of Rostock, the Chalmers University of Technology, Diginext, and Holografika, and eight associate partners from 6 different EU countries. Associate partners include Eyescale Software GmbH (EYE), Geomatics & Excellence (GEXCEL), Compagnia Generale di Riprese aeree (BLOM CGR), Centre d'Essais et de Recherche de l'ENTENTE (CEREN), Fraunhofer IGD, AIRBUS, NVIDIA, AMD. The main goal of the network was to train the next generation of researchers in the fields of 3D data presentation and understanding, with a primary focus on data intensive application environments. I was honored with a 3-years Early Stage Researcher grant to develop my research at CRS4. During those years, I had the great chance to closely collaborate with some of the partners, as well as show and discuss my work with all of them in the numerous meetings around Europe. It has been a fantastic and enlightening experience.

Following my three years withing DIVA, I completed my thesis while working at the CRS4 Visual Computing Group on several interesting projects, mostly in the area of cultural heritage computing.

Contents

List of Figures

List of Tables

1 Introduction

The availability of highly detailed 3D data is continuing to grow at a fast pace thanks to the rapid evolution of 3D sensing and 3D model creation techniques. Many application domains do not simply study offline such data, but require the interactive exploration of visually realistic 3D models. This imposes the challenges of efficiently transform massive amounts of 3D data into renderable representations, and to efficiently render those representations in a visually realistic manner at high frame rates. In this thesis, we introduce novel scalable methods for improving reality-based massive model creation and novel techniques for improving scalability of advanced rendering through GPU-friendly adaptive batching or compression of renderable representations. This chapter outlines the motivation behind this research, summarizes research achievements, and describes the organization of the thesis.

1.1 Background and motivation

HIGH-QUALITY, high-density, and large-scale 3D data is continuously generated at a growing rate from sensors, scanning systems, 3D modeling, or numerical simulations in a large variety of application fields. These technologies, and the ever increasing digitization of work methodologies in all domains, have resulted in very large and complex 3D models of various kinds. One of the most commonly used information is in the form of *3D surface models* which describe the overall shape, and possibly of color, of a real, designed, or simulated object. Several examples from very different domains are shown in Fig. 1.2. Such models are generated from different sources or processes, depending on their function. An informal characterization of the most common model kinds is the following:

- **CAD models**. Computed-Aided Design is used by engineers, architects, designers, etc. to create, modify, analyze, and optimize any kind of object or construction. This scenes often represent very large projects, and as they are intended for technical purposes, usually contain big amounts of precise details. Surface models are routinely used as a renderable representation for visual inspection. The most common renderable surface representation is an optimized triangle mesh derived from the boundary representation of the original CAD

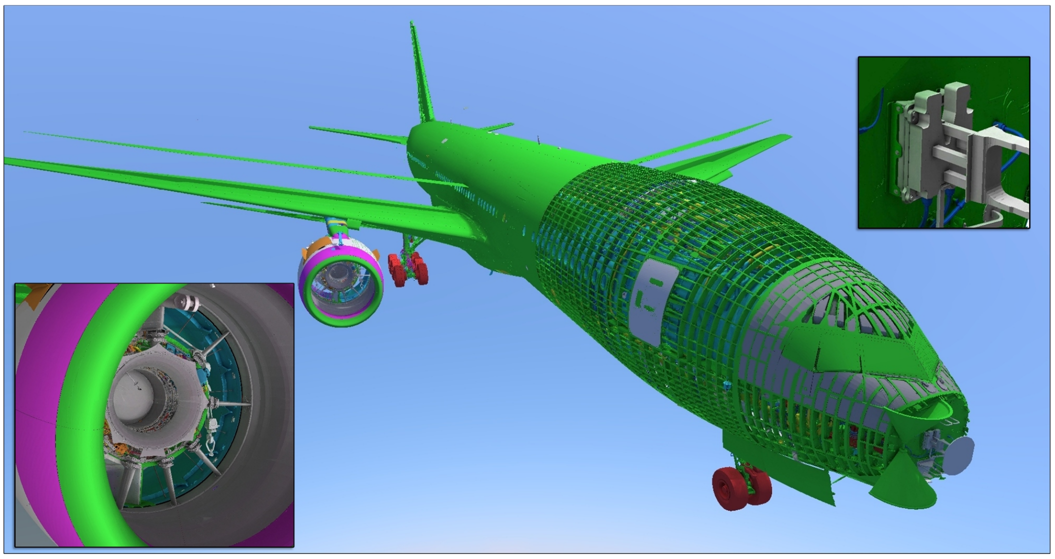
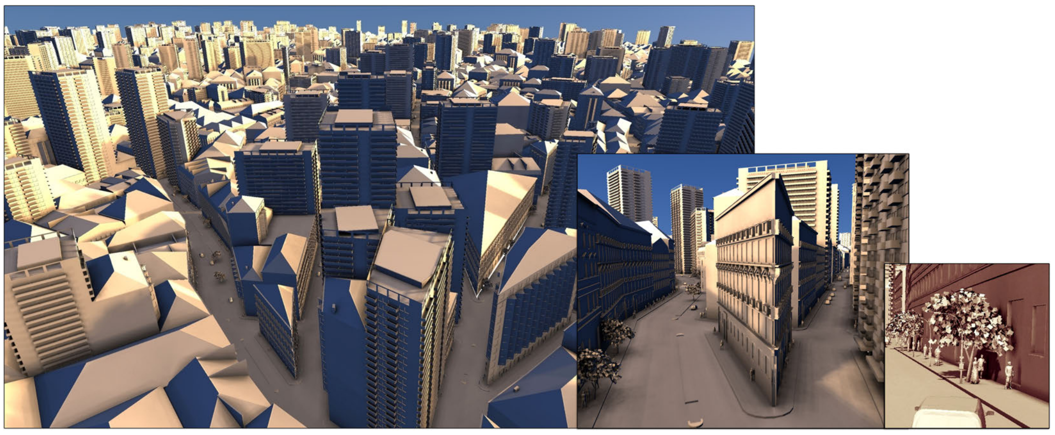



Fig. 1.1.: **Two massive 3D models at different scales, rendered at interactive frame rates with complex illumination**. Top: Boeing 777 model (over $350M$ of triangles), an example of complex real CAD scene. Bottom: City-200 model ($200M$ of triangles), an example of generated urban landscape scene. Even at different scales, it is appreciable the hard and soft shadows, color bleeding and ambient occlusion.

representation, be it solid or parametric. Typical models may be very large. For instance, the Boeing 777 model in Fig. 1.1 contains over $350M$ of triangles.

- **Acquired models**. The use of scanners such as LIDAR (*Light Detection and Ranging*, or *Laser Imaging Detection and Ranging*) or other techniques like photogrammetry has been broadly extended in the last decade. They permit to obtain fine-scale geometric, and often colorimetric information on the real world. Their costs is continuing to drastically reduce, while at the same time their resolution and precision is increasing. These technologies are used in a wide variety of sectors and scales: from engineering purposes with aerial scanning of vasts amounts of terrain or urban landscapes, to cultural heritage field for capturing art objects such as statues, paintings or little pieces. Normally the resolution of those acquired models increases by getting more and more samples, which also generates huge databases difficult to process and visually explore. The most basic representation of such models is the point cloud, with typical datasets now routinely containing from tens of millions to billions of samples. For instance, the David model in Fig. 1.2 contains half a billion points.

- **Computed models**. Many 3D models are obtained as the result of numerical processing which are used mostly in scientific sectors, such as chemistry, biology or astrophysics. They can be also used as structures for secondary computations, like global illumination, shadows or physical collisions, where algorithms similar to the ones used for visualization, are intended to support specific effects or behaviors. While computer simulation can generate models of many kinds, e.g., volumetric scalar fields or vector fields, large surface models are also common. One of the examples are large isosurfaces, which can easily reach the billion of triangles.
- **Designed mesh models**, mostly used in simulators, interior design, virtual reality environments, or by the entertainment industry like films or video-games. This kind of models are usually generated by authoring tools software by designers or artists. A wide variety of model sizes exist, from very small to very large. It is also not uncommon to see, here, models of intricated geometry, exhibiting large amounts of details, see, e.g., the examples in Fig. 1.2.

A large variety of fields, thus, produces what can be defined as a *massive model*. The dictionary [2] defines *massive* as: *(a) Consisting of or making up a large mass; bulky, heavy, and solid; (b) Large or imposing, as in quantity, scope, degree, intensity, or scale; (c) Large in comparison with the usual amount.* As highlighted in the standard reference on massive-model rendering [3], the digital 3D models addressed here are massive in all three senses. In fact, just their surface representation requires millions or billions of geometric primitives and can consume tens of gigabytes and even terabytes of storage. Moreover, the digital datasets representing the models describe high levels of detail that may not be visible to the human eye until magnified, while the overall shape is only perceivable when moving very far. Finally, handling the data on exceeds the usual capacity of conventional processing techniques.

While a variety of analyses can be made off-line on such models, many of their uses require their interactive inspection by human operators. Interactive 3D visualization of these datasets is, however, particularly challenging, given the inherent need of generating visually rich images at high frequency and with low latency in response to viewer motion.

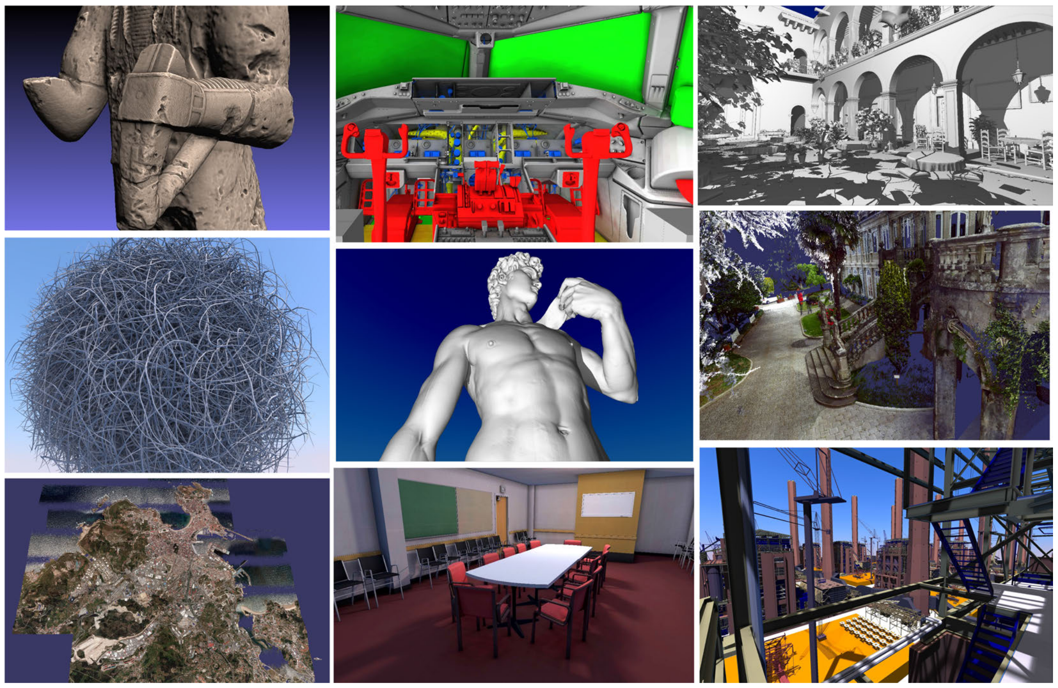


Fig. 1.2.: **Large 3D models from different sectors and types**. These are some examples of massive models processed or rendered with the approaches proposed in this system. From top-left to right-bottom: Mont'e Prama statue (cultural heritage, acquired with laser scanning and digital photography); Boeing 777 cockpit (engineering, CAD model); San Miguel (gaming, authoring software); Hairball (procedural computer generated); David statue (cultural heritage, acquired with laser scanning and colored with photo mapping); Pazo de Lourizán point cloud (architecture, from laser scanning); A Coruña city (urbanism, from aereal LIDAR); Conference room (gaming, authoring tool); 16x Power Plants (engineering, CAD model).

In fact, for a visualization application to be interactive it must, for one side, generate images at a rate high enough to provide the illusion of animation to the human perception system. This typically means to sustain at least 10Hz [3]. Moreover, the application must respond with a latency low enough to provide the impression of instantaneous feedback, which is required to support interactive controls. This means, typically, to take just a few tens of seconds to respond to an action such as a user click or a change in motion direction [3]. In addition, the images generated at high frequency and with low-latency must be of a quality high enough to deliver compelling visual information, which means, for many application, the need to compute shadowing and non-local/global shading.

Despite the continued increase in computing and graphics processing power, it is clear that one cannot just expect to use brute force techniques on more powerful hardware to achieve the goal of interactive inspection for massive data in the general case. This is not only because hardware improvements also leads to the generation of more and more complex datasets, but also because memory bandwidth and data access speed grow at significantly slower rates than processing power and become the major bottlenecks when dealing with massive datasets (see Fig. 1.3), especially in the context of complex non-local illumination simulations, which must combine, per pixel, the contribution of many parts of the scene that affect to shadows and/or interreflections, dramatically increasing the bandwidth requirements.

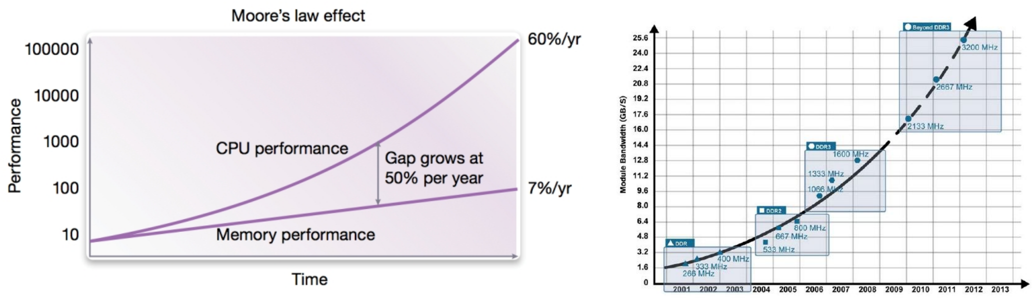


Fig. 1.3.: **Trend of processing v.s. memory performance on time**. Hardware parallelism, e.g., in the form of multi-core CPUs or multi-pipe GPUs, results in the performance improvement, which tends to follow and even outpace Gordon Moore's exponential growth prediction. The CPU performance has increased 60% per year for the last decades. On the other hand, the access time for main memory consisting of DRAM only decreased by 7-10% per year during the same period. Actually, the problem is not memory bandwidth, as it can be seen in the graph at right side, that follows Moore's law trend, but memory latency, as well as memory power consumption. Even if processors get faster and faster, they cannot fetch information fast enough. This relative gap between CPU performance and access time shows that a major computational bottleneck is usually in data access rather than computation, and we expect that this trend will continue in the near future. *Source: Synopsis & Intel*.

For these fundamental reasons, many research efforts have been focused on the problem of devising clever methods to render massive models on graphics hardware (see the classic survey by Yoon et al. [3]), and the more recent survey on ray-tracing solutions by Deng et al. [4]). In general, the main techniques employed in all solutions strive to reduce the amount of data that needs to be stored or processed at any given time. They can be characterized as follows:

- **Data filtering techniques.** Since massive models are too large to be processed and require too much computation, many methods try to quickly devise reduced working sets on which to perform the rendering computation fast enough to meet timing constraints while not reducing the quality. This goal is achieved by employing appropriate data structures and algorithms for visibility or detail culling that quickly eliminate portions of the scene that is proved not to contribute to the final image (see also survey in Sec. 4.2).
- **Adaptive out-of-core techniques.** Since massive models in their entirety just do not fit in graphics memory, and often even in main memory, massive-model

rendering methods are designed to work on out-of-core structures, loading data on demand. Given the high I/O costs, adaptive cache-coherent methods are typically employed, with the goal to reduce the number of cache misses and, thus, lower the data access time (see also survey in Sec. 3.2 and Sec. 4.2).

- **Data compression techniques.** Since the limited amount of memory imposes size bounds on the largest model (or working set) that can be managed in-core, and, at the same time, accessing large amounts of data is also very costly in terms of time, many method lower data size requirements with compression techniques. Since many complex algorithms, such as raytracing, require random access to spatial data structures as well as scene data, the compressed format is designed to support compression-domain rendering or fast and transient random decompression (see also survey in Sec. 5.2).

Many solutions have been proposed that mix and match these ingredients into complex and powerful rendering systems. However, the overall problem of massive-model rendering is far from being solved, and many aspects need further research [5, 6, 7].

In particular, many of the preceding acceleration techniques have been designed and implemented especially for GPU-accelerated rasterization methods using simple local illumination. Computing non-local effects, such as shadows and inter-reflections requires the implementation of approximated multi-pass methods, non trivial to realize in the context of a real-time out-of-core renderer because of the need to carefully schedule data access and processing passes based on complex dependencies among disjoint scene portions. This has mostly limited the quality of the images in real-time walkthroughs based on rasterization solutions [5]. By contrast, high-quality rendering systems supporting advance illumination have been proposed based on real-time raytracing [4], but fully out-of-core solutions have been realized only using CPU acceleration. The complex access pattern of ray-tracing would benefit from compression, for example to fully fit data in GPU memory for a low-latency rendering, but state-of-the-art solutions for the compression of fully renderable spatial data structures and of the associated scene data are either reducing too much the access time to support real-time performance, or not compressing data enough to support very large models [6, 8].

My work in this thesis is mainly motivated by the need of removing these limitations.

1.2 Objectives

I set as a goal of this thesis to contribute to the advancement of the state of the art in the massive-model area, exploring the potential of novel technology that push the boundary in terms of model complexity and rendering quality in an interactive setting on current GPU-accelerated commodity platforms. In particular, I set up for my work the following objectives:

- **Improve massive model creation by extending data fusion processes to scalable structures**. While the main focus of my thesis is on devising and developing techniques to interactively explore massive models, I set up as a first objective to tackle the problem of the efficient handling and creation of 3D models from massive amounts of acquired data. Working on this topic will permit to start work not only on already created models, but from the raw data used to create them. Given the fact that current reality-based surveying techniques, such as digital photography, photogrammetry and laser scanning, are making it possible to quickly acquire very dense shape and color representations of objects and environments, I set as a goal the creation of scalable methods and techniques for managing such large raw-data representation and fuse them to produce clean, renderable, detailed colored shapes, to be used in interactive rendering applications;
- **Improve massive model exploration by an out-of-core work batching approach**. Even though current GPUs support general programming models and allow for running acceleration data structures and complex traversal algorithms, efficient memory management and computation scheduling for ray tracing is significantly harder than for rasterization, leading to performance problems and/or complications when trying to integrate rasterization and ray tracing within the same application, e.g., to compute complex global illumination. In this thesis I will study how to use visibility and smart scheduling of work batches to be able to directly process and render 3D models of massive size from out-of-core memory, within a flexible rendering core that naturally supports complex illumination. I set as a goal the creation of techniques that will enable rendering of massive models of hundreds of millions of primitives with shadows and inter-reflections.
- **Improve massive model exploration by in-core compression approach**. Adaptive out-of-core methods do not have hard limits in the size of models they can handle, due to the fact that they work on batches of limited size,

but inherently introduce some latency when adaptively loading data to GPU to update the working set. In several applications, this latency, even if minimal, is a limiting factor, and therefore, there is a need also for techniques that can squeeze as much data in core as possible in a fully renderable format. In particular, voxelized representations of complex 3D scenes have been widely used recently for this purpose, as they offer a very rendering-friendly data representation. However, these representation are currently too memory-hungry to support massive-model rendering. In this thesis, I set as a goal to improve the compression performance of voxelized representation while minimizing occupancy while keeping similar traversal times to current state-of-the-art solutions, and thus, making it possible to apply GPU raytracing to massive models.

- **Validate the different approaches on real-world massive data**. In this thesis I focus on advancing the state-of-the-art in massive model creation and exploration through the design and implementation of novel data structures and algorithms. In order to really validate, in practice, all these approaches, one of the objectives will be to realize really workable prototype implementations capable to provide unparalleled performance on massive real-world data. Each of the methods will thus need to be benchmarked on a large number of massive data and compared with other existing solutions.

1.3 Achievements

The research work carried out during this thesis has led to the following achievements and peer-reviewed publications.

- The introduction of a general multiresolution design for a scalable system to create, colorize, analyze, and explore massive point clouds totally out-of-core. A GPU-accelerated implementation able to process and render a billion points dataset [9] and its application to fields like cultural heritage or engineering [10, 11]. I personally fully designed all algorithms and data structures and implemented the scalable point-cloud subsystem.

  "**Point Cloud Manager: Applications of a Middleware for Managing Huge Point Clouds**". O. A. Mures,A. Jaspe Villanueva, E.J-Padrón, J.R. Rabuñal. Chapter 13 of “Effective Big Data Management and Opportunities for Implementation” book. Pub. IGI Global (2016)

"**Virtual Reality and Point-based Rendering in Architecture and Heritage**". O. A. Mures, A. Jaspe Villanueva, E.J- Padrón, J.R. Rabuñal.Chapter 4 of "Handbook of Research on Visual Computing and Emerging Geometrical Design Tools" book. Pub. IGI Global(2016)

- An easy-to-apply acquisition protocol based on laser scanning and flash photography to generate colored point clouds [12], which introduces a novel semi-automatic method for clutter removal and photo masking to generate clean point clouds without clutter using minimal manual intervention. The multiresolution design previously introduced allows that the entire masking, editing, infilling, color-correction, and color-blending pipeline to work fully out-of-core without limits on model size and photo count. I contributed to the overal design of the approach and of its implementation. In particular, I especially focused on the camera calibration and color correction for mapping, I designed and implemented the infilling process working on scalable point clouds, and designed and performed large parts of the extensive evaluation. In terms of system, I personally implemented processing methods on top of the scalable point-cloud subsystem.

"**Mont'e Scan: effective shape and color digitalization of cluttered 3D artworks**". F. Bettio, A. Jaspe Villanueva, E. Merella, F. Marton, E. Gobbetti, R. Pintus. ACM Journal on Computing and Cultural Heritage, Vol 8, Num 1 (2015)

- A novel approach to exploit the rasterization pipeline and hardware occlusion queries in order to create coherent batches of work for localized shader-based ray tracing kernels [5]. By combining hierarchies in both ray-space and object-space, the method is able to share intermediate traversal results among multiple rays. Then, temporal coherence is exploited among similar ray sets between frames and also within the given frame. This scheduling architecture naturally allows for out-of-core ray tracing, with the possibility of rendering potentially unbounded scenes. This technique was presented in a joint Eurographics 2015 paper [5]. As for the distribution of work, I contributed to the design of the method, fully designed and implemented the ray-tracing subsytem, and devised and implemented the majority of the evaluation.

"**CHC+RT: Coherent Hierarchical Culling for Ray Tracing**". O. Mattausch, J. Bittner, A. Jaspe Villanueva, E. Gobbetti, M. Wimmer, and R. Pajarola. Computer Graphics Forum Journal Vol 32, Num 2. Presented at Eurographics'15 (2015)

- A novel compression method called SSVDAG (Symmetry-aware Sparse Voxel DAG) [6, 7], which can losslessly represent a voxelized geometry of many real-world scenes, aside with an out-of-core algorithm to construct such representation from a SVO or a SVDAG, as well as a clean modification of standard GPU raycasting algorithm to traverse and render this representation with a small overhead. This technique has proven to compress up to a $1M^3$ voxel grid to fit completely in-core and render it in realtime. I consider SSVDAGs the main contribution of this thesis, since I co-designed the method and the techniques for its efficient implementation, fully implemented all the system components, and performed the full evaluation. The resulting system has also been released as open source together with the Journal of Computer Graphics Techniques publication.

  "**SSVDAGs: Symmetry-aware Sparse Voxel DAGs**". A. Jaspe Villanueva, F. Marton, and E. Gobbetti- ACM SIGGRAPH i3D full paper (2016).

  "**Symmetry-aware Sparse Voxel DAGs (SSVDAGs) for compression-domain tracing of high-resolution geometric scenes**". A. Jaspe Villanueva, F. Marton, E. Gobbetti. Journal of Computer Graphics Techniques Vol 2 Num 6 (2017).

  "**Voxel DAGs and Multiresolution Hierarchies: From Large-Scale Scenes to Pre-computed Shadows**". U. Assarsson, M. Billeter, D. Dolonius, E. Eisemann, A. Jaspe Villanueva, L. Scandolo, E. Sintorn. Eurographics Tutorials (2018).

- The evaluation of all the previous methods on very large scale data. In particular, as described in Chapters 3.4.7, 5.6, and 4.7, all methods have been tested with models exceeding the hundreds of millions primitives.

As described in the survey of related work on data fusion (Sec. 3.2), dynamic work batching for real-time out-of-core rendering (Sec. 4.2), compressed representations (Sec.. 5.2), the work included in this thesis clearly advanced the state-of-the-art in each of the targeted domains.

1.4 Organization

This thesis is organized in order to show in a natural and coherent order all the results obtained. Many readers would prefer to skip parts of the text and go back and forth through the different chapters. In this section, the reader can find a brief overview of what can be found in each chapter.

In this chapter I covered the background and motivation for this Ph.D. dissertation, described my objectives, and summarized my results. The next chapter provides more details on the generals framework under which this thesis was developed, including a list of definitions of the main concepts, the general assumptions under which I have worked, an explicit list of the hypotheses that had to be verified by this thesis, as well as the means I intended to use for the verification.

The following three chapters are the core of this thesis, as they present my main achievements in the area of massive model creation and exploration. These three chapters have a similar structure: they first establish the goal and contributions described in the chapter, then describes the state-of-the-art works related with the proposed approach, and then explain the proposed solution itself, before evaluating the results obtained on very large massive models. Every one of them finishes with a discussion of advantages and drawbacks, and some bibliographical notes of their content, in which I refer to the original publications, explain the differences with respect to the published articles, and point to interesting follow-up works by myself or other authors.

The final chapter provides a short summary of the achievements, as well as other related works and publications carried out during the period of the Ph.D. program. Finally, there is a critical discussion of the results obtained and of how they advance the state-of-the-art, as well as some reflections on future lines of work.

2 General requirements, work hypotheses, and means of verification

Before presenting in details my research work, I summarize here the general framework under which this thesis was developed. In particular, I provide a list of definitions of the main concepts, I summarize the general assumptions under which I have worked, and provide an explicit list of the hypotheses that had to be verified by this thesis, as well as the means I intended to use for the verification.

2.1 Basic definitions

The following is a list of important definitions for the main concepts that appear in this thesis:

- **Massive model**. Three dimensional scene or set of objects which require are extremely large in comparison to those usually found in similar application and pose scalability problems. This can mean that their representation is massive, e.g., they require millions or billions of geometric primitives and can consume tens of gigabytes and even terabytes of storage, or that the digital datasets representing the models describe high levels of detail that may not be visible to the human eye until magnified, while the overall shape is only perceivable when moving very far, or that handling the data for a particular graphics application exceeds the usual capacity of conventional processing or rendering technique.
- **Surface model**. In contrast to the solid nature of reality, surface models describe only the superficial matter of the objects, by represent only the external boundary of the objects using geometric surface entities and defining how they interact with light. This kind of models are the most used in a variety of domains (see Sec. 1.1), as they are normally used to explore the visual aspect of the scenes, without a need of their solid properties.

- **Point cloud representation**. This is the most elementary of the surface representations, it consists of set of point samples over the surface. Its simplicity as dataset lies on its lack of topology. It is mostly used for acquired models, e.g., with techniques like LIDAR or photogrammetry. Beyond its spatial coordinates, every point can store properties like reflectance or color. It can be directly rendered by many techniques (see Sec. 3.2), but usually the point cloud representation is used as a starting step to produce more complex representations (e.g., triangle meshes).
- **Triangle mesh representation**. This is probably the most popular representation, as many rendering algorithms and graphic hardware have specialized in its format. It is a special case of a polygonal mesh, and defines an explicit surface representation topologically composed of a set of vertices and of triangles connecting them. The geometric component is specified by associating a 3D position at each vertex. Vertices can also have properties, which represent geometrical aspects of the surface, such as the normal, or of its material, such as albedo, reflection index, roughness, etc. This representation are widely used by CAD software, interactive applications, design, films, video-games, etc. (see Sec. 1.1).
- **Voxelized representation**. A voxel represents the minimal cubical subdivision on a regular grid in a three-dimensional space. A surface model can be represented by voxels by rasterizing the surface in the grid, i.e., by marking a voxel full if it intersects the surface (eventually also storing the surface attributes at that location), or empty if no intersection is found. The advantage of this representation is that it is very efficient to trace, i.e., provides quick methods for computing whether a given ray segment intersects a non-empty voxel. Thus, it can be used to accelerate rendering computations, not only for direct render but as support structures for secondary computations, such as visibility. A voxelized representation at high resolution is potentially very large, and thus requires an implementation based on compression methods. Normally these grids have a high degree of sparsity and redundancy, and thus are highly compressible.
- **Local illumination**. Also known as direct illumination, is a lighting model of 3D surfaces that only takes into account the radiation coming directly from the light source to the faces of the object, without any interaction with the rest of the scene. This constraint makes it very simple to apply, but its lacks visual realism and details (shadows, interreflections, occlusions). It is very fast as it only needs a streaming memory access pattern. It is directly implemented in the graphic processing units using rasterization algorithm.

- **Shadows**. Darkening of areas of the scenes caused by objects between the light source and the illuminated area. The objects are caused blockers. Computing shadows increases rendering complexity as it requires computing visibility not only from the camera point of view but also from point of view of each contributing light.
- **Ambient occlusion**. It is a particular kind of shadow, in which the darkening of each point in a scene depends on how exposed it is to ambient lighting. Ambient occlusion can thus be seen as an accessibility value that is calculated for each surface point. The result is a diffuse, non-directional shading effect that casts no clear shadows but provides a low-frequency darkening.
- **Global illumination**. A lighting model for 3D surfaces that takes into account not only the light that comes directly from a light source (direct illumination), but also subsequent cases in which light rays from the same source are reflected by other surfaces in the scene, whether reflective or not (indirect illumination). Theoretically, reflections, refractions, and shadows are all examples of global illumination. Often, however, only the simulation of diffuse inter-reflection or caustics is called global illumination. In this theses, we consider the expanded definition.
- **Rasterization**. Used widely in interactive graphics, is implemented in all Graphics Processing Units (GPUs) of modern video boards. It is an object-order rendering algorithm, thus, primitives are sequentially projected to image plane, where they are converted to pixels and shaded. For resolving visibility, It is used in combination with a z-buffer, which stores for each pixel the distance to the observer. A rasterization pipeline can potentially process an arbitrary number of primitives in stream-like manner. This can be useful for large scenes that do not fit in memory. While this algorithm has linear complexity with the number of objects to process, it can be converted to logarithmic by using spatial index structures. Moreover, as the gap between performance and bandwidth throughout the memory hierarchy is growing, appropriate techniques must be employed to manage large working sets, and ensure coherent access patterns. Basic rasterization, however, supports only local illumination, and implementation of complex non-local effects is non trivial, typically requiring multi-pass methods.
- **Ray-tracing**. In contrast to rasterization, ray-tracing is an image-order rendering approach. It tries to model physical light transport as straight paths across the space. In their classical form, it starts shooting primary rays from the observer trough the pixels of the image plane, towards the 3D scene. Then,

secondary rays can be casted for compute many light effects, such as shadows, ambient occlusion, diffuse interreflection, etc. The methods is embarrassingly parallel and very elegant, but also very costly in terms of computation and memory optimization.

- **Interactive exploration**. An reactive software which allows users to see a given model from different points of views by continuously controlling a virtual camera using an interaction device. The main kinds of interactive exploration are *interactive walkhtroughs*, in which the users simulate moving inside an environment, and *object inspection*, in which the user moves an object to see different portions of it from different angles and at different scales. All interactive exploration applications will need to (loosely) meet frequency and latency constraints with massive models. This will translate to support a refresh rate of at least 5-10Hz and a latency in responding to user actions of just a few frames (see Sec. 1.1).

2.2 Research questions

I set as a goal of this thesis to contribute to the advancement of the state of the art in the massive-model area, exploring the potential of novel technology that push the boundary in terms of model complexity and rendering quality in an interactive setting on current GPU-accelerated commodity platforms. In particular, as mentioned in Sec. 1.2, my main research objectives are to improve massive model creation by extending data fusion processes to scalable structures, and to improve massive model exploration by an out-of-core work batching approach, as well as by an in-core compression approach. To reach these objectives, I will need to answer the following specific research questions:

1. *How to create a scalable data-fusion method to create consolidated 3D point clouds and 3D meshes from large collections of photographs and range scans in the difficult case of cluttered acquisition?* The problem here will be to be able to develop scalable algorithms and data structures capable to discriminate clutter from object data and to consolidate them in a fused colored surface model (point cloud or triangle mesh). The particular test case analyzed will need to be that of stone statues with metal supports. The methods should be designed to work on a potentially unbounded amount of input photographs and range scans.

2. *How to exploit the hardware-supported rasterization hardware to support ray-tracing of massive-models with non-local illumination?* The problem here will be to develop scalable algorithms and data structures to make it possible to naturally use an adaptive approach for out-of-core rendering in a GPU-accelerated ray-tracing framework. The resulting framework should be designed to enable the easy inclusion of non-local illumination, and in particular shadows, ambient occlusion, and diffuse inter-reflections for interactive exploration of massive models.
3. *How to compress voxelized representations, so as to make them usable as a main geometric representation and not just as a secondary structure for low-frequency shadows?* The problem here will be to create a voxelized representation that exploits sparseness and redundancy of scenes to compress very high resolution voxelizations to a very compact GPU representation that can be traversed by a GPU-accelerated ray-tracer at approximately the same speed of state-of-the-art uncompressed representation. The creation process will need to be able to work on a potentially unbounded amount of input voxels.

As a general requirement, all the proposed solutions will need to work on the platforms that are typically used by end-users for construction and exploration in the targeted domains (see Sec. 1.1). These are typically consumer-level PCs with multicore CPUs with standard amounts of RAM (e.g. 16GB and above) and high-end gaming graphics boards (e.g., NVIDIA GeForce).

2.3 Hypotheses supporting the prospected solutions

My answers to the research questions, which will be detailed in the following chapters, are based on the following hypotheses:

1. *How to create a scalable data-fusion method to create consolidated 3D point clouds and 3D meshes from large collections of photographs and range scans in the difficult case of cluttered acquisition?*
   a) An acquisition pipeline based on flash photography and laser scanning permits to easily acquire color and geometry. Since for cultural heritage applications clutter and objects have different reflectance characteristics, it will be possible to use a very small subset of the input range maps and color

maps to produce a training dataset that will be sufficient for classification algorithms to automatically mask unwanted colors and geometry.

b) Out-of-core point clouds managed in spatial structures such as kd-trees can be able not only to render massive point datasets but also support the different stages of a construction pipeline and make the results of the processes permanent in the disk structure. In particular, it will be possible to color consolidate all laser acquisitions into a single point cloud using a single streaming pass over range maps, and to color the resulting point cloud with an albedo by also streaming over the registered point clouds. Since the location of the flash with respect to the camera is known, it will be possible to exploit it to derive surface albedo from the measured apparent color for each of the surface points.

2. *How to exploit the hardware-supported rasterization hardware to support ray-tracing of massive-models with non-local illumination?*

   a) A novel generalization of hierarchical occlusion culling in the style of the *CHC++* [13] will make it possible to exploit the rasterization pipeline and hardware occlusion queries in order to create coherent batches of work for localized shader-based ray tracing kernels. The main hypothesis is that by extending *CHC++* to jointly traverse hierarchies in both ray-space and object-space, we will be able to generalize occlusion culling for arbitrary packets of rays, opening the door to writing ray-tracing algorithms on top of a rasterization framework.

   b) An occlusion-based GPU raytracer working on coherent batches can naturally support full out-of-core raytracing by scheduling together data I/O and computation, and using a cache of most recently used objects to reduce data transfer.

3. *How to compress voxelized representations, so as to make them usable as a main geometric representation and not just as a secondary structure for low-frequency shadows?*

   a) Sparse Voxel DAGs [14] dramatically reduces the size of a sparse voxel octree without impacting on voxel traversals by merging identical subtrees. Our expectation is that it will be possible to further improve compression by also considering similarity transforms in the matching process.

   b) Among the many possible similarity transformations, *reflective symmetries*, i.e. mirror transformations along the main grid planes, are very interesting for our application, since the 8 possible reflections can be encoded using only 3 bits (reflection X,Y,Z), the transformation ordering is not important, as transformations along one axis are independent from the others, and efficient access to reflected subtrees, which requires application of the direct transformation or of its inverse can be achieved by simple coordinate (or index) reflection. Our expectation is that it will be easy to find a large enough number of matches to compensate the overhead of symmetry encoding, and that the traversal algorithm will be quick.

c) We expect that the distribution of references to shared nodes will be very skewed, with a small number of nodes appearing a large number of times, and a larger number of nodes appearing less times. We thus assume that it will be possible to further improve compression by exploiting the skewed distribution of references to shared nodes to store child pointers using a variable bit-rate encoding.

d) Fully out-of-core transformation from a sparse voxel octree to a sparse voxel DAG will be possible by a bottom-up external-memory algorithm that reduces an SVO to a minimal SSVDAG by alternating matching and merging at each level.

e) We expect that the cost of decoding and application of transformations will be negligible, and that traversal costs will be similar to those of sparse voxel octrees and sparse voxel DAGs.

2.4 Means of verification

In order to really validate, in practice, the proposed solutions, I will need to develop working implementations capable to provide unparalleled performance on massive real-world data. Each of the methods will thus need to be benchmarked on a large number of massive data and compared with other existing solutions. The planned benchmark results are the following:

1. We expect the point-cloud data structure to manage at least a billion-point dataset (construction and rendering).
2. We expect to semi-automatically identify clutter by an automated classification method based on a small manually marked working set, with a classification error of less than 1%.
3. We expect to be able to render triangulated representations of models of the size of the Boing 777 at interactive rates with non-local shading using a commodity PC (NVIDIA GeForce class acceleration with 4GB memory).

4. We expect to significantly exceed (at least 30%) the compression rate of state-of-the-art SVDAG solution for voxelized representation of scenes.
5. We expect to fit voxelized representations of models of the size of the Boing 777 at sub-millimetric precision fully in GPU memory (NVIDIA GeForce class acceleration).
6. We expect to render voxelized representations in our compressed format at a speed similar (less than 100% overhead) to sparse voxel octree solutions.

3

Improving reality-based massive model creation: scalable out-of-core point-clouds and effective data-fusion methods

Current reality-based surveying techniques, such as digital photography, photogrammetry and laser scanning, are making it possible to quickly acquire, at a low cost, and relatively simply, very dense shape and color representations of objects and environments. Point clouds are a very natural representation of such sampled models. However, current datasets acquired at a high-resolution tend to be very massive. This large size makes handling these datasets very complex, and requires scalable solutions at all processing stages. In this chapter, after presenting a general design and implementation of a system for creating, coloring, analyzing, and exploring very large point clouds, I will focus on the solution of an important problem, especially in the cultural heritage area: the effective shape and color digitization of cluttered 3D artworks. As a result, I will show how complex reality-based models can be effectively created. The forthcoming chapters will, instead, focus on solutions to the exploration problem.

The increasing performance and proliferation of digital photography and 3D scanning devices is making it possible to acquire, at reasonable costs, very dense and accurate sampling of both geometric and optical surface properties of real objects. A wide variety of cultural heritage applications stand to benefit particularly from this technological evolution. In fact, this progress in the technology is leading to the possibility to construct accurate colored digital replicas not only for single digital objects but at a large scale. Accurate reconstructions built from objective measures have many applications, ranging from virtual restoration to visual communication.

Point clouds are one of the most used data types to represent such models in fields like engineering, environmental sciences, or cultural heritage. They are naturally scalable as, the more samples the dataset has, the finer is the representation of the real object or scene. However, current point cloud datasets may become untractable

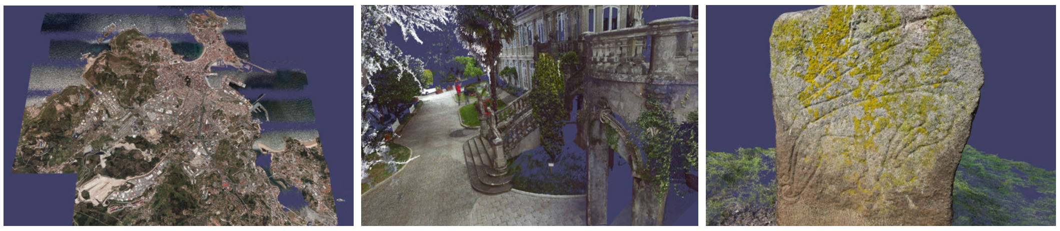


Fig. 3.1.: **Three large point clouds from different acquisition techniques.** Examples of large point clouds at different scales, rendered interactively with the described system in the first part of this chapter. Left: aerial LIDAR data of A Coruña city. Center: Pazo de Lourizán, with terrestrial LIDAR and digital photography. Right: Ancient inscriptions on a dolmen, photogrammetry and digital photography.

on nowadays hardware, given that they can easily exceed the billions of samples. Managing such large datasets requires scalable techniques. In this chapter, I will consider the common case in which a very large point cloud must be optimized so as to quickly allow for multiresolution exploration, analysis, and coloring. Common examples of applications of these structures are:

- data fusion of point clouds with photographic data, e.g., for the creation of photorealistic models from acquisitions done with lasers for the shape and cameras for the color;
- extraction of geometric features such as planes, cylinders, etc. for engineering purposes;
- real-time exploration of massive point cloud models on a variety of computers, adapting the complexity of rendering to the capability of the platforms.

All these use cases require techniques capable to statically optimize a point cloud to optimally transform it to a multiresolution structure maintained out-of-core from which to extract at run-time levels of details for the various required operations. In this chapter, I will present an implementation of an architecture based on a refinement of the Layered Point Cloud approach. I will then focus on a particularly challenging application of data fusion of points and images in cultural heritage: the effective shape and color digitization of cluttered 3D artworks.

3.1 Contribution

The main contributions of this research are:

- a general design for a scalable system for creating, coloring, analyzing, and exploring massive point clouds totally out-of-core;
- an easy-to-apply acquisition protocol based on laser scanning and flash photograph to generate colored point clouds;
- a simple and practical semi-automatic method for clutter removal and photo masking to generate clean point clouds without clutter using minimal manual intervention;
- a scalable implementation of the entire masking, editing, infilling, color-correction, and color-blending pipeline, that works fully out-of-core without limits on model size and photo number;
- the evaluation of the method and tools in a large-scale real-world application.

I personally fully designed and implemented the scalable point-cloud subsystem, also described in [9, 15, 16]. As for the novel technique for scalable shape and color digitalization of cluttered artwork, published in JOCCH [12], I contributed to the design and implementation of color processing and mapping on point clouds, to the infilling process, and to the extensive evaluation.

3.2 Related work

The use of points as rendering primitives has been introduced very early [17, 18], but over the last decade they have reached the significance of fully established geometry and graphics primitives [19, 20, 21]. Many techniques have since been proposed for improving upon the display quality, levels-of-detail rendering, as well as for efficient out-of-core managing of large point models. The approaches shown in this chapter combines state-of-the-art results in a number of technological areas. In the following text we only discuss the approaches most closely related to our novel contributions. For more details, we refer the reader to the survey literature [21, 22, 23].

3.2.1 Out-of-core point cloud management

For many years, QSplat [24] has been the reference system in massive point rendering. It consists in an out-of-core hierarchy of bounding spheres, traversed at run-time to generate points. Nowadays, its main drawback is that the algorithm is CPU bounded, as in the original technique computations are made per point, and CPU/GPU communication requires a direct rendering interface, thus the graphic board is never exploited at its maximum performance.

More recently, Grottel et al.[25] presented an approach for rendering of Molecular Dynamics datasets represented by point glyphs, which also includes occlusion culling and deferred splatting and shading. The method uses a regular grid rather than a hierarchical data decomposition, and has thus limited adaptivity. Sequential Point Trees [26] introduced a sequential adaptive high performance GPU oriented structure for points limited to models that can fit on the graphics board. XSplat [27] and Instant points [28] extend this approach for out-of-core rendering. XSplat is limited in LOD adaptivity due to its sequential block building constraints, while Instant points mostly focuses on rapid moderate quality rendering of raw point clouds. Both systems suffer from a non-trivial implementation complexity. Layered point clouds [29] and Wand et al.'s out-of core renderer [30] are prominent examples of high performance GPU rendering systems based on hierarchical model decompositions into large sized blocks maintained out-of-core. The layered point clouds are based on adaptive BSP subdivision, and subsamples the point distribution at each level. In order to refine an level-of-detail, it adds points from the next level at runtime. This composition model and the pure subsampling approach limits the applicability to uniformly sampled models and produces moderate quality simplification at coarse levels-of-details. In Bettio et al.'s approach [31] these limitations are removed by making all BSP nodes self-contained and using an iterative edge collapse simplification to produce node representations. We propose here a faster high quality simplification method based on adaptive clustering. Wand et al.'s approach [30] is based on an out-of-core octree of grids, and deals primarily with grid based hierarchy generation and editing of the point cloud. The limitation is in the quality of lower resolutions created by the grid no matter how fine it is. All these previous block-based methods produce variable sized point clouds allocated to each node. None of them support fully continuous blending between nodes, potentially leading to popping artifacts.

All the mentioned pipelines for massive model rendering create coarser level-of-detail nodes through a simplification process. Some systems, e.g., [29, 30], are inherently forced to use fast but low-quality methods based on pure subsampling or grid-based clustering. Others, e.g., [27, 31], can use higher quality simplification methods, as those proposed by Pauly et al. [32]. Lastly, Goswami et al.[33] propose a fast high quality technique which combines clustering, greedy selection, and delayed point combination, with unstructured sets of points.

3.2.2 Color acquisition and blending

Most cultural heritage applications require the association of material properties to geometric reconstructions of the sampled artifact. While many methods exist for sampling Bidirectional Radiance Distribution Functions (BRDF) [34, 35] in sophisticated environments with controlled lighting, the typical cultural heritage applications impose fast on-site acquisition and the use of low-cost and easy to use procedures and technologies. Color photography is the most common approach. Since removing lighting artifacts requires knowledge of the lighting environment, one approach is to employ specific techniques that use probes [36, 37]. However, these techniques are hard to use in practice, in typical museum settings with local lights. Dellepiane et al. [38, 39] proposed, instead, to use light from camera flashes. They propose to use the Flash Lighting Space Sampling (FLiSS) – a correction space where a correction matrix is associated to each point in the camera field of view. Nevertheless, this method requires a laborious calibration step. Given that medium- to high-end single-lens reflex (SLR) cameras support fairly uniform flash illumination and RAW data acquisition modes that produce images where each pixel value is proportional to incoming radiance [40], we take the simpler approach of using a constant color balance correction for the entire set of photographs and apply a per-pixel intensity correction based on geometric principles. This approach effectively reduces calibration work. In this work, in addition to distance-based correction, we employ a more complete correction that also takes into account surface orientation. The method is similar to the one originally used by Levoy et al. [41], without the need of special fiber optic illuminators and of per-pixel color response calibration. Under our flash illumination, taken from relatively far from the statues (2.5m), the flash can be approximated as a point source and energy deposition on the statue is negligible compared to typical ambient lighting. In addition, while previous color blending pipelines worked on large triangulated surfaces [41, 42] or single-resolution point-clouds [43], we blend images directly on multiresolution structures leading to increased scalability. The pipeline presented in our original work [44] is also combined here with inpainting and infilling methods for constructing seamless models. Our implementation is based on combining screened Poisson reconstruction [45] with an anisotropic color diffusion process [46] implemented in a multigrid framework.

3.2.3 Color and geometry masking

Editing and cleaning the acquired 3D model is often the most time-consuming reconstruction task [47]. While some techniques exist for semi-automatic 3D clutter removal in 3D scans, they are typically limited to well-defined situations (e.g., walls vs. furniture for interior scanning [48] or walls vs. organic models for exterior scanning [49]). Manual editing is also typically employed in approaches that work on images and range maps. For instance, Farouk et al. [50] embedded a simple image editor into the scanning GUI. We also follow the approach of working on 2D representations, but concentrate our efforts to reduce human interventions. Interactive 2D segmentation is a well-known research topic with several state-of-the-art solutions that typically involve classification and/or editing of color image datasets (see well-established surveys [51, 52]). In general, the aim of these techniques is to efficiently cope with the foreground/background extraction problem with the least possible user input. The simplest tool available is the Magic Wand in Adobe Photoshop 7 [53]. The user selects a point and the software automatically computes a connected set of pixels that belong to the same region. Unfortunately, an acceptable segmentation is rarely achieved automatically since choosing the correct color or intensity tolerance value is a difficult or even impossible task. Many classic methods, such as intelligent scissors [54], active contours [55] and Bayes matting [56], require a considerable degree of user input in order to achieve satisfactory results. More accurate approaches have been presented that solve the semi-automatic image segmentation problem by using Graph Cuts [57]; here the user marks a small set of background and/or foreground pixels as seeds and the algorithm propagates that information to the remaining image regions. Among the large number of extensions to the Graph Cuts methodology [58, 59], the *GrabCut* technique [60] combines a very straightforward manual operation, with a color modeling and an extra layer of (local) minimization to the Graph Cuts technique; this requires a small effort from the user but proves to be very robust in different segmentation scenarios. In this work we propose an adaptation of the *GrabCut* approach to the problems of editing point cloud geometries and pre-processing images for texture blending.

3.3 Out-of-core massive point cloud management

Normally, the process of acquiring 3D scans generates sets of unsorted three-dimensional samples with a number of attributes associated per point, such as color, refraction index, normal, etc. The type of processes that usually run over this data have an spatial nature, such as the measurement geometrical properties, computation of point attributes, etc. Many of them also are adapted for a coarse-to-fine solutions, giving an initial approximation which is refined as the computation time goes on. Actually, interactive exploration, i.e. render algorithms, can be seen as one of this processes where the result is a visual representation of the data given a point of view.

By the other side, it is desirable to take the most advantage possible of the hardware that commodity computers (even including mobile ones) implement today. In particular, graphic hardware has been shown to be very efficient dealing with streams of spatial data, not only for rendering but even for general purpose computations. However, its main memory is normally orders of magnitude smaller than these datasets. The objective of the proposed system is to convert these theoretically unbounded, unstructured point clouds into a multiresolution and memory friendly structure that allows:

- scale its performance depending on the hardware;
- interactive output-sensitive exploration;
- solving spatial queries in fast access times;
- running non-interactive processes over the point cloud;
- level-of-details and recursive refinement of those processes;
- use not only CPU but GPU hardware to maximize the computing throughput;
- read/write access to the associated point data;

In the next pages I will briefly describe a design which is able to fulfill these requisites. This system is not intended for editing the point cloud in geometrical terms (i.e. changing the position of the points, or making small add/remove operations) but take advantage of its static nature to allow very fast, spatial and multiresolution read/write access to the associated data.

3.3.1 System architecture

The figure 3.2 shows a high-level diagram of the proposed architecture. As it was established before, the main parts are the spatial structure, which arranges the points in a efficient, multiresolution, layered way, and the memory hierarchy, in charge of transfer the data through the system. In my proposed implementation, point information is divided into same-sized chunks. The number of points contained in

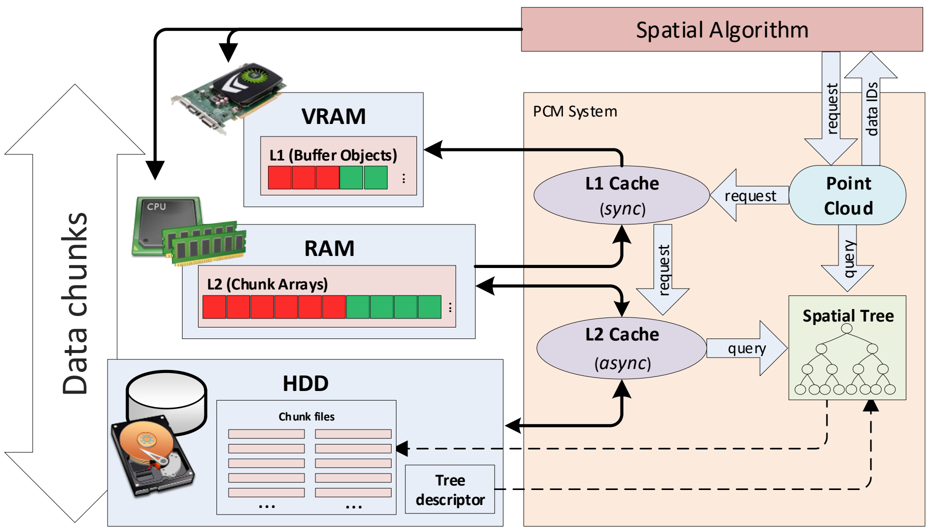


Fig. 3.2.: **Architecture of the proposed system to manage large point clouds.** Point cloud is divided into data chunks following an additive kd-tree. These data chunks are transferred over a memory hierarchy, directed by the two levels of cache.

every chunk is fixed by the hardware capabilities and the operative system parameters. Their size depends also on the amount of information (normal, color, etc.) associated to every point. The memory hierarchy proposed, allows fast access to the points with spatial requests, managing transparently the transfers between permanent storage (HDD), main CPU memory (RAM) and main GPU memory (VRAM). This stack of memories has pyramidal shape, as normally is size-decreasing. Between these memory stages, two levels of cache manage the data transfers, using geometrical criteria based on the tree structure to preload subdivisions of the scene at different levels of resolution.

An algorithm that needs to operate with the point cloud, can access both at CPU level or a GPU level. It requests to work on a region of the space at some particular level of detail. It can also specify a time and memory budgets for its requests, so the system can reserve resources for interactive processes or other computations. The system also allows many instances with different point clouds sharing the same virtual space.

The spatial tree structure. The system uses the structure proposed by Gobbetti and Marton in their technique *Layered Point Clouds* [29, 61]. They construct a non-redundant, additive kd-tree structure, where points do not lay only in the leaves, but are sparse into the different node levels by an statistical, space-homogeneous distribution. This concept is represented in the figure 3.3.

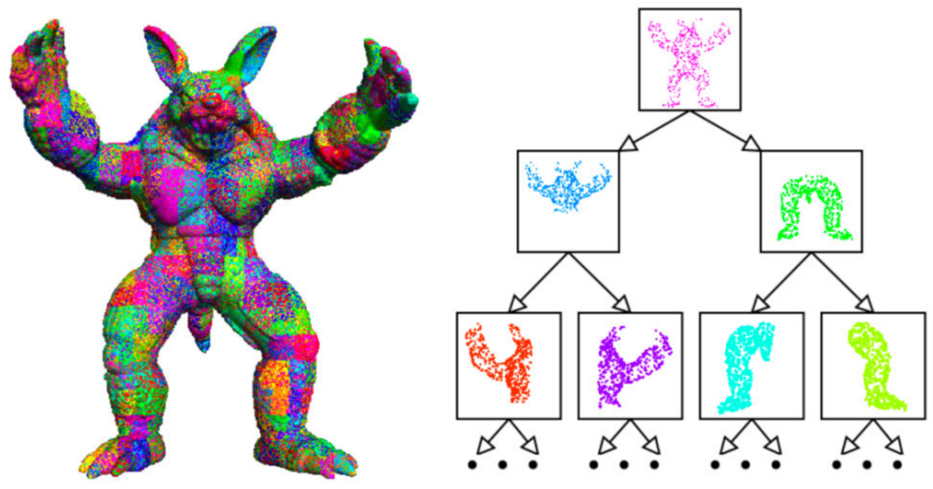


Fig. 3.3.: **Layered point cloud structure example.** This example from [29] illustrates the kd-tree structure used in the proposed system, where points are homogeneously distributed over the nodes statistically covering the most part of the region they represent.

Every node is mapped to a persistent data chunk in disk, which encode not only the position of the points, but also their associated information, that can variable for every point cloud. The number of points per chunk, M, is the only parameter to decide at building time. The construction of the kd-tree is done by an out-of-core process that recursively partitions the space occupied by the points, that should be uniformly distributed over the model. For every iteration we choose a partition axis, and in only one pass over points, we both compute a pseudo-median and statistically choose the candidate M points. At the end of the iteration, the chunk with the candidates is written, as well as the node information (splitting plane, splitting axis, chunk ID, etc.) and rest of the point cloud is divided in two by the splitting cut. The process then is applied independently to the two resultant point clouds if its cardinal its $> M$, otherwise a leaf node is written.

As the number of points can be theoretically unbounded, we need scalable techniques for some of these steps. For on side, all the algorithms are applied in arrays that work out-of-core, so that its length is not bounded by the system main memory. By the other side, two of the processes, the computation of the median as well as the choose of well-distributed samples, have to use statistical techniques with linear complexity. In both cases, analytical solutions would have complexities such as $O(n * log)$ or even $O(n^2)$, which are no affordable for the massive datasets the system pretends to deal with. In the case of the median, a low-level algorithm takes advantage of the IEEE float encoding, using a first pass with an histogram of the coordinates most-significant bits to choose a high probability centered range, and a second pass generating a new histogram pruning all the samples non coincident in their first bits with the selected in the previous pass, and selecting the center one. The tests has proven to have a very low error rates, even with massive amounts of coordinates [9]. Instead, the choice of the points to belong to every node is done by a stratified russian-roulette style algorithm, taking care of having a uniform distribution of the samples.

This structure fulfills the requisites of the system about being able to progressively access to different levels of detail, by simply loading deeper nodes. Also, its organization is very suitable for a cache when used with spatial coherency, which is the kind of algorithms the system is intended for.

The L2 asynchronous cache. This cache is in charge of the data transfers between persistent memory (HDD) and RAM (CPU), and it is used by a only-CPU algorithm or by the L1 cache. They commit it request for points contained in regions of the space up to certain level of detail. The cache uses the tree structure to continually load nodes that fulfill those space constraints, making them available in main memory. For caching, it uses a simple Less Recently Used (LRU) replacement policy as it has been shown to have a good performance in this kind of systems. It works in a completely asynchronous manner in its own process, so the request are attended parallel while the CPU and/or GPU are processing the available chunks. The list of chunks to load always sorted by the level of the nodes in the tree that they belong, from the root to the leaves. This way a course representation is available as soon as possible, and finer levels are loaded as time goes on. L2 cache is able also to write the data flagged chunks by the L1 cache and keep consistency inside the memory hierarchy.

The L1 synchronous cache. This cache manages the data transfers between RAM (CPU) and VRAM (GPU). It breaks the point chunks into *buffer objects* and upload them synchronously to the GPU memory, making them available both for rendering (with *OpenGL*) or per parallel computing (with *OpenCL*). Like L1 cache, it uses a Less Recently Used (LRU) replacement policy and a hash table, which maps between chunks and buffer objects IDs. In this case there are two types of requests:

- **Restricted by time**, where the cache will load all the possible nodes in the given budget of time, and return the control to the calling process. These request are used when computing in iterative or multiresolution mode, so the solution can be progressively refined, such in rendering or when calculating an spatial measure.

- **Restricted by level**, where it will load all the required nodes up to some level of resolution (or up to fill the maximum allowed budget of memory for the cache), no matter how long it takes. This is the option for precise computations in selected regions of space, or when using algorithm not adapted for a coarse-to-fine solution.

The L1 cache has also the ability to store changes made in the GPU (e.g. by an OpenGL kernel) to be persistent. When a node is evicted from the cache, its changes will be written to the L2 before its deletion, as well as flagged for persistent writing in disk.

Both caches can also use a prediction technique when they are in idle state after having loaded all their requests. It tries to anticipate which could be the next region of interest analyzing the spatial patterns of the last requests, so it can fill the whole memory budget, or even replace chunks of points that have not been demanded lately.

3.3.2 Interactive exploration

The proposed architecture allows real-time visualization of the point clouds, as a particular read-only spatial process. The render algorithms keeps a front cut of the tree and creates a list of nodes every frame, given a center of detail. Different techniques prune this list, such as:

- **level of detail culling**, computing the size of the projected nodes into the screen, and dismissing those smaller than pixel, as their contribute would be insignificant;
- **view culling**, to dismiss nodes outside the camera frustrum;
- **backface culling**, to dismiss nodes which do not face to the camera;

Then, the remaining list is sorted by two different criteria. First, by levels of detail, so at least there is always a course representation of the model, and then by distance to the camera. At this point, the query is sent to the system with a few milliseconds of time budget, that can be also adapted on-the-fly by the desired frame rate. After this lapse of time for loading, the render is implemented in OpenGL, and take advantage of the programmable stages of the pipeline for using the splatting technique [24].

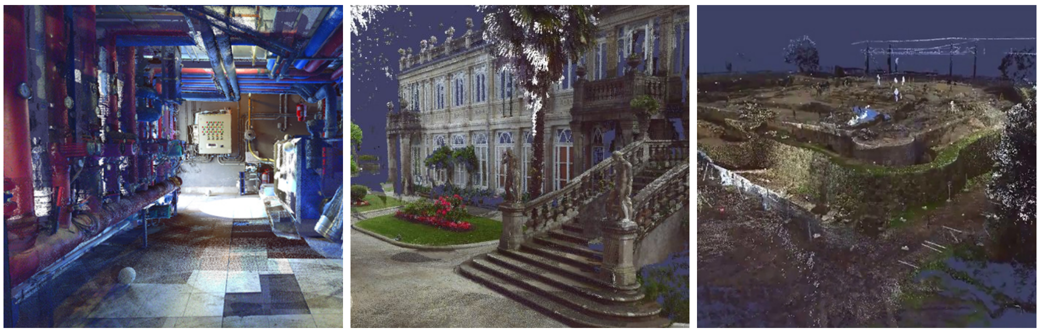


Fig. 3.4.: **Three examples of massive point clouds datasets explorated interactively with PCM.** All these dataset has been processed and rendered at real-time frame rates in different configurations of commodity hardware, equipped with dedicated GPUs. Left: Calderas datasets, $\sim 200M$ points. Center: Lourizan dataset, $\sim 90M$ points. Right: Castelo dataset, $\sim 1B$ points.

3.4 Supporting shape and color digitization cluttered 3D artworks

Cultural heritage is one of the application areas in which digitization is most commonly applied. In this field, scalable point cloud architectures such as the one described above are employed for a variety of needs. One of the most challenging is the creation of realistic and detailed colored models from digitized data.

One of the digitization approaches most widely used is a combination of laser scanning with digital photography. Using computational techniques, digital object surfaces are reconstructed from the laser-scan-generated range maps, while the apparent color value sampled in digital photos is transferred by registering the photos with respect to the 3D model and mapping it to the 3D surface using the recovered inverse projections. Since early demonstrations of the complete modeling pipeline (e.g., [62, 41]), most of its components have reached sufficient maturity for adoption in a variety of application domains. This approach is particularly well suited to cultural heritage digitization, since scanning and photographic acquisition campaigns can be performed quickly and easily, without the need to move objects to specialized acquisition labs. The most costly and time consuming part of 3D reconstruction is thus moved to post-processing, which can be performed off-site. Thus, in recent years research has focused on improving and automating the post-processing steps – for instance, leading to (semi-)automated scalable solutions for range-map alignment [63], surface reconstruction from point clouds [64, 65, 66, 67], photo registration [68, 69], and color mapping [42, 70, 43]. Even though passive image-based methods have recently emerged as a viable (and low-cost) 3D reconstruction technology [71], the standard pipeline based on laser scanning or other active sensors still remain a widely used general-purpose approach, mainly because of the higher reliability for a wider variety settings (e.g., featureless surfaces) [71, 72].

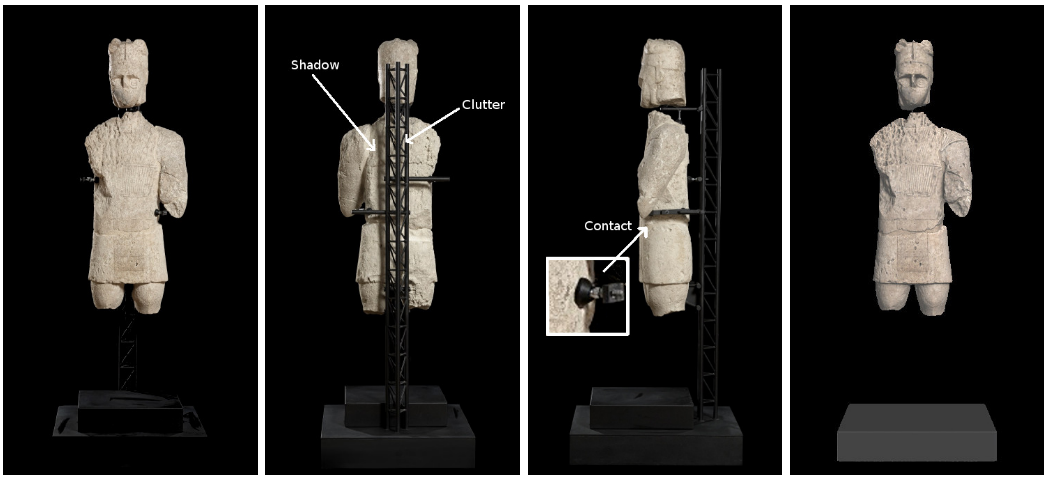


Fig. 3.5.: **Reassembled Nuragic statue with supports and its virtual reconstruction.** The black support structure holds the fragments in the correct position, with minimal contact surface, avoiding pins and holes in the original material. A 360-degree view is possible, but color and shape capture is difficult because of clutter, occlusions, and shadows. The rightmost image depicts our 3D reconstruction. Photo courtesy of ArcheoCAOR.

In this section, we tackle the difficult problem of effectively adapting the 3D scanning pipeline to the acquisition of color and shape of 3D artworks on-site, in a cluttered environment. This case, arises, for instance, when scanning restored and reassembled ancient statues in which (heavy) stone fragments are maintained in place by a custom exostructure (see Fig. 3.5 for an example).

Digitizing statues without removing the supports allows one to perform scanning directly on location and without moving the fragments, therefore enabling a completely contactless approach. On the other hand, the presence of the supporting structure typically generates shadow-related color artifacts, holes due to occlusion effects and extra geometry that must be removed. With the standard 3D scanning pipeline these issues lead to laborious and mostly manual post-processing steps – including cleaning the geometry and careful pixel masking (for more details see Sec. 3.2 with an overview of the related work).

Motivated by these issues, in this work we present a practical approach for improving the digitization of the shape and color of 3D artworks in a cluttered environment. While our methods are generally applicable, the work was spurred by our involvement in the *Digital Mont'e Prama* project (see Sec. 3.4.1), which included the fine-scale acquisition of 37 large statues and therefore required robust and scalable methods.

3.4.1 Context and method overview

The design of our method, which is of general use, has taken into account requirements gathered from domain experts in the context of a large scale project. In this subsection we provide a general overview of our approach, justifying the design decisions in relation to the requirements.

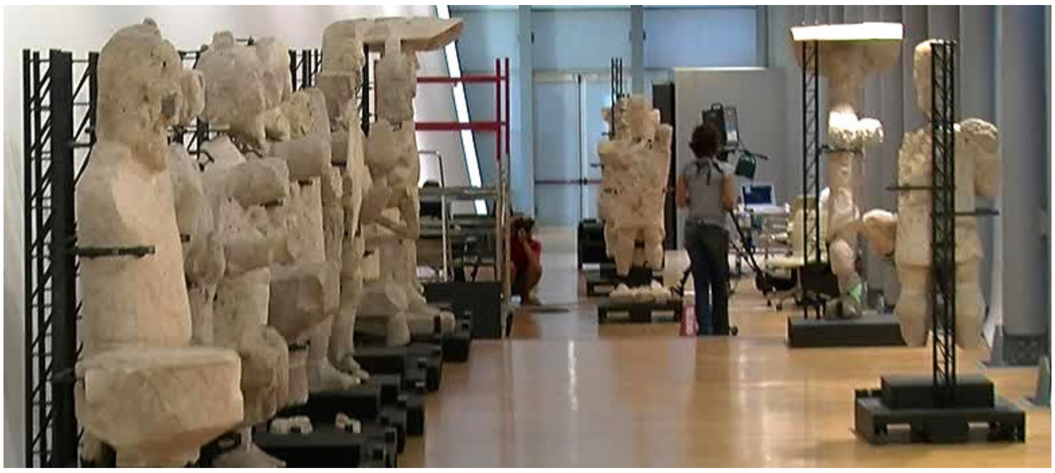


Fig. 3.6.: **Mont'e Prama Statues on display at the CRCBC exhibition hall.** Scanning was performed on-site.

Figure 3.7 outlines the approach used, which consists of a short on-site phase and a subsequent, mostly automatic, off-site phase. The only on-site operations are the acquisition of geometry and color, which are performed in a contact-less manner by only sliding and rotating the statue while it is mounted on its support. The geometry acquisition operation is performed with a triangulation laser scanner, which produces range and reflectance maps that are incrementally coarsely aligned during scanning in order to monitor 3D surface coverage. On the other hand, color is acquired in a dark environment by taking a collection of photographs with an uncalibrated camera while using the camera flash as the only source of light. A Macbeth color checker, visible in at least one of the photographs, is used for post-process color calibration. Analogously to the geometry acquisition step, coverage is (optionally) checked on-site by coarsely aligning the photographs using a Structure-from-Motion (SfM) pipeline.

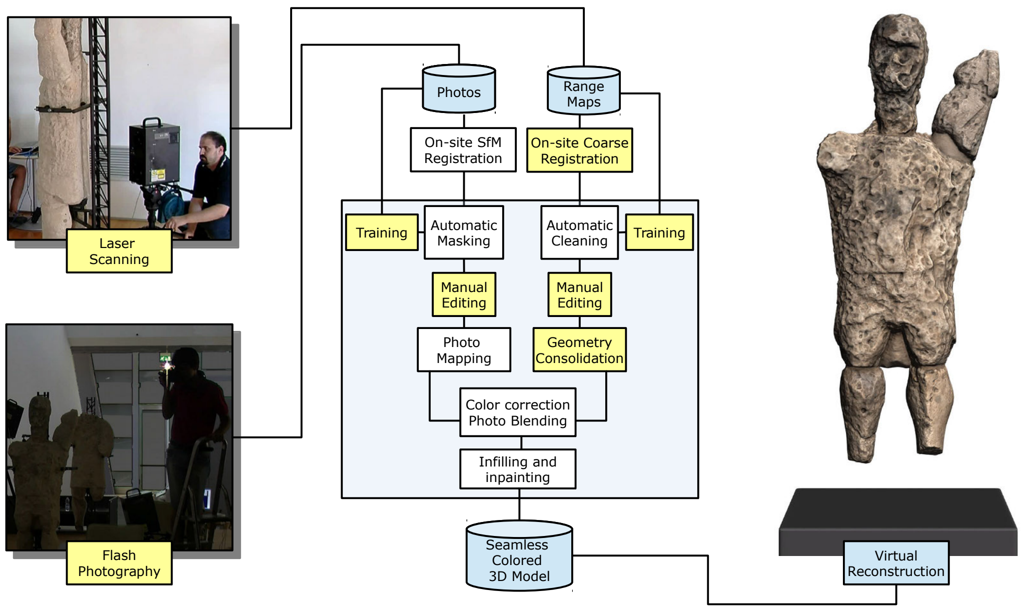


Fig. 3.7.: **Pipeline.** We improve digitization of 3D artworks in a cluttered environment using 3D laser scanning and flash photography. Semi-automated methods are employed to generate masks to segment the 2D range maps and the color photographs, removing unwanted 3D and color data prior to 3D integration. Sharp shadows generated by flash acquisition are handled by the masking process and color deviations introduced by the flash light are corrected at color blending time by taking into account object geometry. A final seamless model is created by combining Poisson reconstruction with anisotropic color diffusion. User-guided phases are highlighted in yellow.

The remainder of the work can be performed off-site using semi-automatic geometry and color pipe-lines that communicate only at the final merging step. In order to remove geometric clutter the user manually segments a very small subset of the input range maps and produces a training dataset that is sufficient for the algorithm to automatically mask unwanted geometry. This step exploits the reflectance channel of the laser scanner. As commonly done for cultural heritage pipelines, the automatic masking can be in principle revised by visually inspecting and optionally manually improving the segmentation, using the same tools employed for creating the example masks. Note that this step, in contrast to previous work [50], is entirely optional (see Sec. 3.4.7 for an evaluation of the manual labor required). In order to create a clean 3D model, the masks are applied to all range maps, which are then finely registered with a global registration method and optionally edited manually for the finishing touch. The geometry reconstruction is then performed using Poisson reconstruction [64], which takes care of infilling small holes that appears in the unscanned areas (i.e., where supports touch the surface of the scanned object) using a smoothness prior on the indicator function. The effect is similar to volumetric diffusion [73].

The color pipeline follows a similar work pattern. It begins from the photographs in raw format. After the training performed by the user on a small subset of images the algorithm automatically masks all the input photos removing clutter. The user then optionally performs a visual check and a manual refinement, then the masked photos – already coarsely aligned among themselves with SfM – are aligned with the geometry using the method by Pintus et al. [74]. The photos are finally mapped to the surface by color-blended projection [42, 43]. During the blending step colors are calibrated using a data extracted from the color checker and the differences in illumination caused by the flash used during photography are corrected using geometric information. Finally, an anisotropic diffusion process [46] is employed to perform a conservative inpainting of the areas left without colors due to occlusions.

It should be noted that the infilling and inpainting approaches employed in this work are minimalist. We do not aim at reconstructing high frequency details in large areas. Instead, we just smoothly extend color and geometry from the neighborhood of holes to avoid the presence of confusing and unattractive holes for public display applications.

Details on semi-automatic geometry and color masking, as well as scalable data consolidation and color mapping are provided in the following sections. The corresponding phases are highlighted in yellow in Fig. 3.7.

3.4.2 Data acquisition

While geometry acquisition is performed using the standard triangulation laser scanning approach, color acquisition is performed using an uncalibrated flash camera. n our context, flash illumination is a viable way to image the objects, as it provides us with sharp shadows together with information on the image-specific direction of the illumination. Since at color mapping-time the geometry of the image is known, we can correct each projected pixel according to the position of the surface on which it projects with respect to the camera and the flash's light, thus obtaining a reasonable approximation of the surface albedo (see Sec. 3.4.5). In addition, cluttering material – e.g., the supporting exostructure – generates sharp shadows which can be easily identified both by the masking process and by taking into account geometric occlusion in the color mapping process (see Sec. 3.4.3).

In contrast to previous work [38, 39, 41], we handle images directly in RAW format, which allows us to correct images without prior camera calibration. We experimentally measured that on a medium/high end camera, such as the Nikon D200 employed in our work, acquisition in RAW format produces images with pixel values proportional to the incoming radiance at medium illumination levels (as also verified elsewhere [75, 40]), and that the flash emits a fairly uniform light within a reasonable working space.

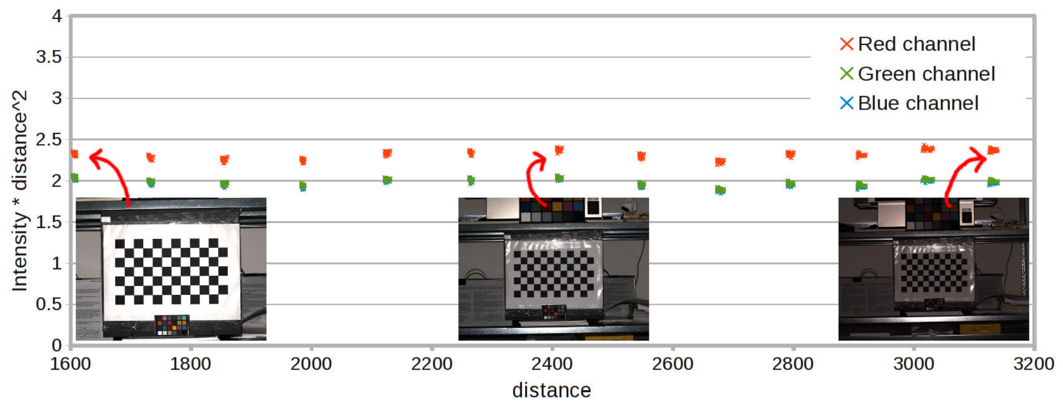


Fig. 3.8.: **Flash illumination.** Using RAW camera data distance-based scaling provides a reasonable correction. Balance between color channels can then be ensured using color-checker-based calibration.

Sensor near-linearity has been verified by taking images of a checkerboard in a dark room with $t = 1/250s\ f/11.0 + 0.0, ISO400$. As shown in the graph in Fig. 3.8 values (measured on the white checkerboard squares) are proportional to $1/d^2$, where d is the distance from the flash light. Distance-based scaling can be thus exploited at color mapping time to provide reasonable correction, while balance between color channels can be ensured using color-checker-based calibration (see Sec. 3.4.5). The characteristics of flash illumination have been verified by taking a photograph of a white diffuse material at 2m using the same 50mm lens used for photographing the statues. After correcting for lighting angle and distance the illumination varies only by a maximum of 7.6% within the view frustum.

While more accurate results may be obtainable with calibration techniques, even the most accurate ones performed off-site [41, 38, 39, 75] do not perfectly match local shading and illumination settings since, in particular, indirect illumination is not taken into account and the photographed materials are (obviously) not perfect

Lambertian scatterers. We thus consider this uncalibrated approach to be suitable for practical use. It should be noted that whenever needed these alternate techniques can be easily performed during post-processing using the same captured data. Indeed, the availability of RAW images in the captured database grants the ability to perform a variety of post-process enhancements [40].

3.4.3 Semi-automatic geometry and color masking

Our masking process aims to separate the foreground geometry (the object to be modeled) from the cluttering data (in particular, occluding objects), under the assumption of different appearances – as captured in the reflectance and color signals. Starting from a manual segmentation of a small set of examples (see Sec. 3.4.3) we train a histogram-based classifier of the materials (see Sec. 3.4.3), which is then refined by finding an optimal labeling of pixels using graph cuts (see sec. 3.4.3). Then, a final (optional) user-assisted revision can be performed using the same tool used for manual segmentation.

Manual segmentation. To perform the initial training the user is provided with a custom segmentation tool, with the same interface for range maps and color images. The tool allows the user to visually browse images/scans in the acquisition database, visually select a small subset (typically, less than 5%) and draw a segmentation in the form of a binary mask – using white for foreground and black for background. The mask layer is rendered on top of the image layer and the user can vary the transparency of the mask to evaluate the masking results. In addition to using standard draw/erase brushes, our tool supports interactive grab-cut segmentation [60] in which the user selects a bounding box of the foreground object to initialize the segmentation method.

Histogram-based classification. The user-selected small subset of manually masked images and range maps is used to learn a rough statistical distribution of pixel values that characterize foreground objects. For artifacts made of fairly uniform materials – e.g., stone sculptures – 3-4 range maps and 4-5 images are typically sufficient.

For the range maps we build a 1D histogram of reflectance values, quantized to 32 levels, by accumulating all pixels that were marked as foreground in the user-defined mask. On the other hand, for the color images we use a 2D histogram based on hue and saturation, both quantized to 32 levels. Ignoring the value component is more

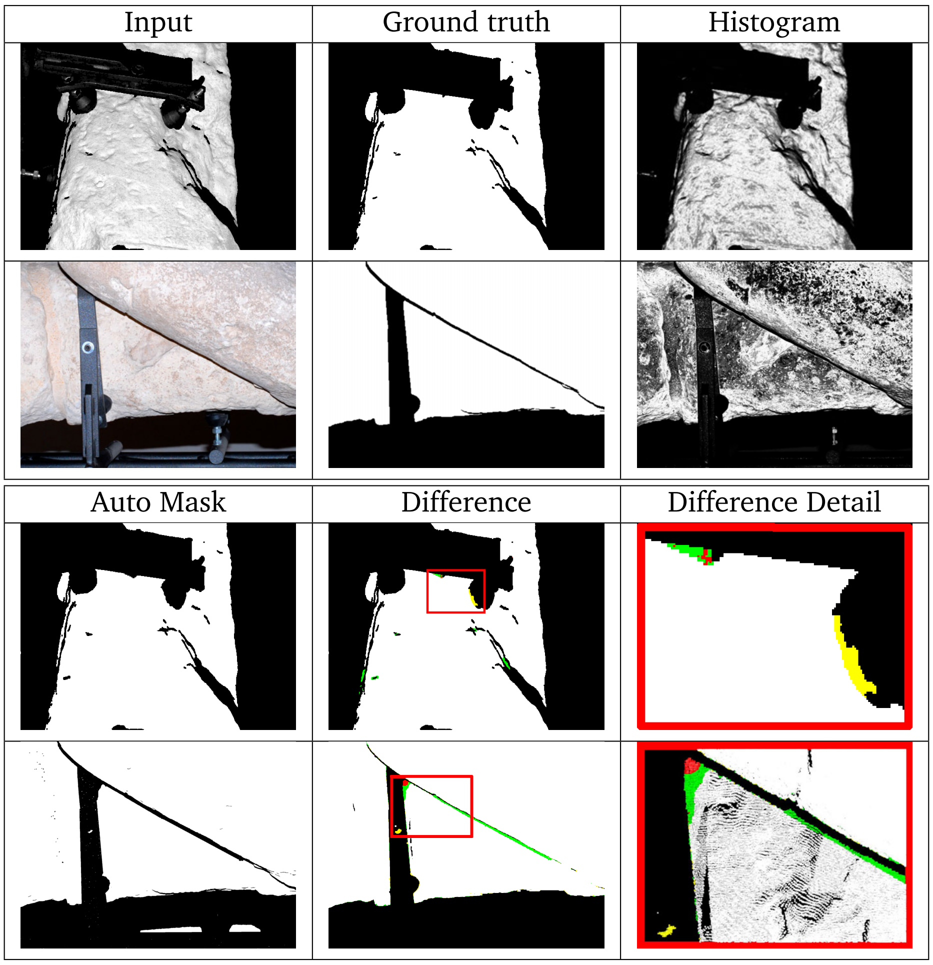


Fig. 3.9.: **Automatic masking.** Geometry (top row) and color (bottom row) results for a single image. From left to right: acquired reflectance/color image; user-generated ground truth masking; mask generated by histogram-based classification; final automatically generated mask; difference to ground truth; magnified region of difference image. In the difference image, black and white pixels are perfect matches, while yellow pixels are false positives, green pixels are false negative points on this image, and red pixels are real false negative points considering the entire dataset.

robust to shading variation due to flash illumination and variable surface orientation. The process is repeated for all manually masked images, thus accumulating histogram values before a final normalization step. The histogram computed on the training set can be used for a rough classification of range map/image pixels based on reflectance/color information. This classification is simply obtained by back-projecting each image pixel to the corresponding bin location and interpreting the normalized histogram value as a foreground probability.

It is worth noting that whether the histogram is computed from foreground or clutter data is not important; as long as the rest of the pipeline is consistent the only constraint is the aforementioned assumption that the two appearances are reasonably well separable.

Graph cut segmentation. As illustrated in Fig. 3.9, third column, the histogram-based classification is very noisy but roughly succeeds in identifying the foreground pixels, which are generally marked with high probabilities. This justifies our use of histograms for the rough classification step rather than the more complex statistical representations typically used in soft segmentation [76, 56].

Segmentation is improved by using the rough histogram-based classification as starting point for an iterated graph cut process. We initially separate all pixels in two regions: probably foreground for those with normalized histogram value larger than 0.5, and probably background for the others. We then iteratively apply the GrabCut [60] segmentation algorithm using a Gaussian Mixture Model with 5 components per region and estimating segmentation using min-cut. As illustrated in Fig. 3.9, column 4, the process produces tight and well regularized segmentation masks.

Morphological post-pass. Since masks must be conservative, especially at silhouette boundaries where a small misalignment is likely to occur, we found it useful to post-process the masks using morphological filters. After denoising the mask using a small median filter (5x5 in this work) we perform an erosion of the mask using an octagon kernel (4x4 in this case). This step has the effect of eliminating small isolated spots and to avoid being too close to the silhouettes, and the removal does not create problems given the large overlap of images that cover the model.

3.4.4 Data consolidation and editing

The final result of the automatic masking step is a mask image associated to each range map and color image. These masks are used for pre-filtering the geometry and color information before further processing. The remaining processing steps, optionally including color mapping (see Sec. 3.4.5), are performed on the point cloud structure.

The final colored point clouds can then be further elaborated to produce seamless surface models. To produce consolidated models represented as colored triangle meshes, the most common surface representation, there exist a number of state-of-the-art approaches [66, 65, 64]. We have adopted the recent screened Poisson surface reconstruction approach [45] which produces high-quality watertight reconstructions by incorporating input points as interpolating constraints, while reasonably infilling missing areas based on smoothness priors. Because the Poisson approach does not handle colored surfaces we incorporate color in a post-processing phase (see Sec. 3.4.5).

3.4.5 Color correction, mapping, and inpainting

The color attribute is obtained first by projecting masked photos onto the 3D model reference frame and then performing seamless texture blending of those images onto the surface [68, 43]. In contrast to previous work, we blend and map images directly to the out-of-core structure and perform color correction starting from captured RAW images during the mapping operation.

Streaming color mapping. Our streaming photo blending implementation closely follows our previous work [68, 43], which we have extended to work on the multiresolution point cloud structure. We associate a blending weight to each point, which is initialized at zero. We then perform photo blending adding one image at a time. For each image, we start rendering the point cloud from the camera point of view, using a rendering method that performs a screen-space surface reconstruction and adapting the point cloud resolution to 1 projected point/pixel. We then estimate a per-pixel blending weight with screen-space GPU operations that take as input the depth buffer as well as the stencil masks (see Pintus et al. [43] for details). In a second pass on the point cloud, we update the point colors and weights contained in the visible samples of the leafs of the multiresolution structure. Once all images are updated we consolidate the structure recomputing bottom-up the colors and weights of inner node samples using averaging operations. As a result, the colored models are available in our out-of-core multiresolution point cloud structure for further editing.

In order to apply this same process to triangulated surfaces, such as those coming out of the Poisson reconstruction, we import the surface vertices in our octree, perform the mapping, and then map the color back to the triangulated surface. In this manner we can use the spatial partitioning structure for view-frustum and occlusion culling during mapping operations.

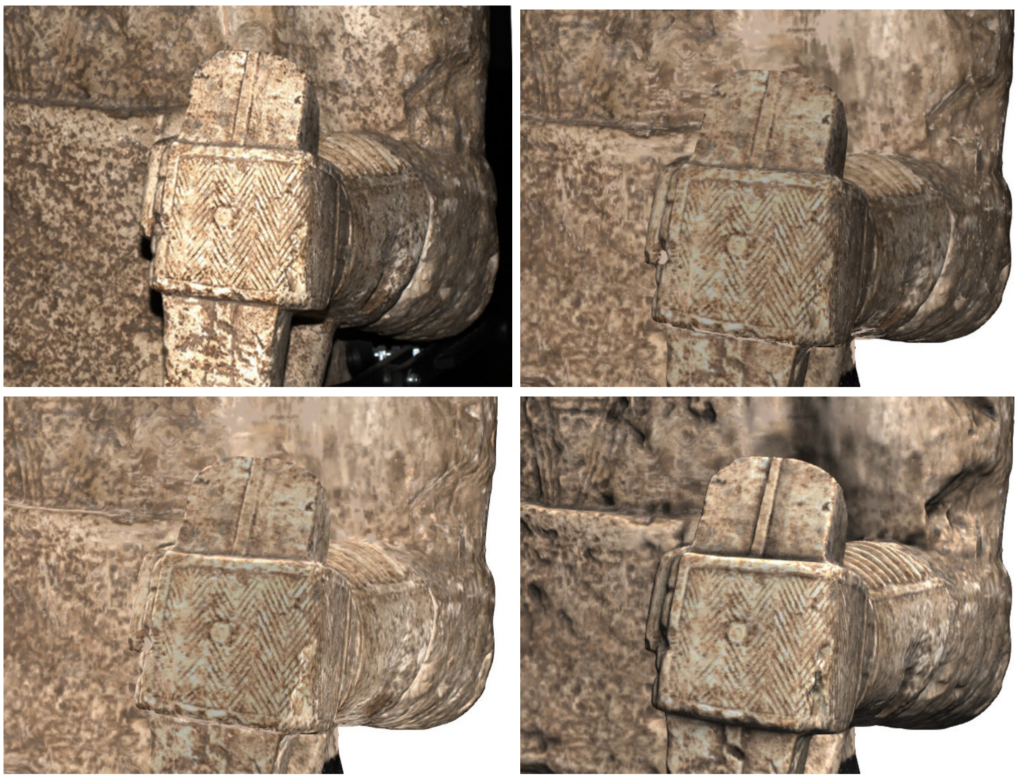


Fig. 3.10.: **Color correction and relighting.** Top-left: original image under flash illumination; note the sharp shadows and uneven intensity. Top-right: projected color with distance-based correction and no synthetic illumination; notice that the flash highlight has been removed, but a darker shade is on the slanted surface. Bottom-left: projected color with distance-based and orientation-based correction and no synthetic illumination; note the even distribution and good approximation of surface albedo. Bottom-right: synthetically illuminated model based on recovered albedo using a different lighting setup.

Flash color correction. Color correction happens at color blending time during color mapping operations. At this phase of the processing the color mapping algorithm knows the color stored in the corresponding pixel of the RAW image (the *apparent color* $\mathbf{C}_{(raw)}$), the camera parameters (camera intrinsic and extrinsic parameters as well as flash position), and the geometric information of the current sample (position and normal stored in the corresponding pixel of the frame buffers used to compute blending weights).

As we verified, the RAW data acquisition produces images where each pixel value is proportional to incoming radiance and the flash light is fairly uniform (see Sec. 3.4.7). Therefore, we apply a simple color correction method based of first principles, similar to the original approach of Levoy et al. [41], but without per-pixel calibration of flash illumination and camera response. The results presented in this section assume that the imaged surface is a Lambertian scatterer so that the measured color, for a sufficiently distant illumination, can be approximated for each color channel i by

$$\mathbf{C}^{(i)}_{(raw)} \approx w^{(i)}_{(balance)} \frac{\mathbf{I}_{(flash)}}{d^2} \mathbf{C}^{(i)}_{(surface)} (\mathbf{n} \cdot \mathbf{l})^{+} \tag{3.1}$$

where $w^{(i)}_{(balance)}$ is the channel's scale factor used to achieve color balance, $\mathbf{I}_{(flash)}$ is flash intensity, $\mathbf{C}^{(i)}_{(surface)}$ is the diffuse reflectance (albedo) of the colored surface sample, $\mathbf{n}$ is the surface sample normal, $\mathbf{l}$ is the flash light direction, and d is the distance of the surface sample from the flash light. As in standard settings, the color balance factors are recovered by taking a single image of a calibration target (Macbeth charts in our case) and using the same setting used for taking the photographs of the artifacts.

Thus, to compute the color of the surface at color mapping time we consider a user-provided desired object distance d_o, using equation 3.1 to find

$$\mathbf{C}^{(i)}_{(mapped)} = \frac{d^2}{w^{(i)}_{(balance)} d_0^2 (\epsilon + (1-\epsilon)\tilde{\mathbf{n}} \cdot \mathbf{l})} \mathbf{C}^{(i)}_{(raw)} \tag{3.2}$$

where ϵ is a small non-null value (0.1 in our case) and $\tilde{n}$ is the smoothed normal (obtained in screen-space with a 5x5 averaging filter. Normal smoothing and dot product offsetting are introduced to reduce the effect of possible over-corrections in the presence of a small misalignment – particularly at grazing angles. It should be noted that, since the Lambertian model does not take into account the roughness of the surface, under flash illumination it tends to over-shadow at grazing angles. As noted by Oren and Nayar [77], this effect is due to the fact that while the brightness of a Lambertian surface is independent of viewing direction, the brightness of a rough surface increases as the viewing direction approaches the light source direction. The small angular weight correction thus also contributes to reduce the boosting of colors near silhouettes.

Figure 3.10 shows how a single flash image introduces sharp shadows and uneven intensity based on distance and angle of incidence. Shadows are removed by the color masking process described in Sec. 3.4.3 as well as by shadow mapping during color projection. Distance-based correction removes flash highlights but still produces darker shades on slanted surfaces. On the other hand, combining distance-based and orientation-based correction produces a reasonable approximation of surface albedo, thereby enabling a seamless combination of multiple images without illumination-dependent coloring. The resulting colored model can thus be used for synthetic relighting.

3.4.6 Inpainting

Points of contacts between supports and status generate small holes in the geometry, as well as missing colors due to occlusions and shadows (seen in white in Fig. 3.11 left). In order to produce final colored watertight models – useful, e.g., for public presentations – it is important to smoothly reconstruct these missing areas. We took the conservative approach of only using smoothness priors to perform geometry infilling and color inpainting, rather than applying more invasive reconstruction methods based on – for instance – non-local cloning methods. This conservative approach has the advantage of not introducing spurious details, while repairing the surface enough to avoid the presence of distracting surface and color artifacts during virtual exploration. Geometry infilling is simply achieved by applying a Poisson surface reconstruction method [45] to reasonably infill missing areas based on smoothness priors (see Fig. 3.11 center).

On the other hand, color inpainting uses an anisotropic color diffusion process [46] implemented in a multigrid framework. We employ a meshless approach that can be applied either to the vertices of the triangle mesh produced by Poisson reconstruction, or directly to a point cloud constructed from it. We assume that each color sample stores the accumulated color and weight coming from color blending. We first extract all points with a null weight, which are those requiring infilling. We then extract a neighbor graph for this point cloud (by edge connectivity when operating on a triangle mesh or by a k-nearest neighbor search, with k=8, when working on point clouds), growing the graph by one layer in order to include colored points in the neighborhood of holes. We then produce a hierarchy of simplified graphs using sequence coarsening operations on the neighbor graph, so that each level has only one quarter of the samples of the finer one. We stop simplification when the number of nodes is small enough (less than 1000 in this workr) or no more simplification edges exists. The graph is used to quickly compute anisotropic diffusion using a multigrid solver based on V-Cycle iterations. Boundary conditions are computed using the samples with non-zero weight that are included in the hierarchy. The anisotropic diffusion equations are then successively transferred to coarser grids by simple averaging and used in a coarse-to-fine error-correction scheme. Once the coarser grid is reached the problem is solved through Gauss-Seidel iterations and the coarse grid estimates of the residual error can be propagated down to the original grid and used to refine the solution. The cycle is repeated a few times until convergence (results in this work use 10 V-Cycle iterations). As illustrated in Fig 3.11 right, color diffusion combined with watertight surface reconstruction successfully masks the color and geometry artifacts due to occlusion and shadows. It is important to note that the original colors and geometry are preserved in the database and that these extra colors can be easily removed from presentation when desired.

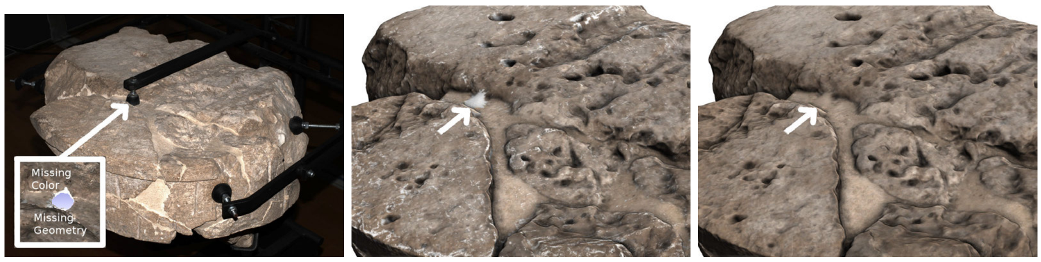


Fig. 3.11.: **Geometry infilling and inpainting.** Left: the points of contact between the support and the statue generate small holes in the geometry as well as missing colors due to occlusions (in white). Middle: Poisson reconstruction smoothly infills holes. Right: color is diffused anisotropically for conservative inpainting.

3.4.7 Implementation and results

We implemented the methods described in this work in a C++ software library and system running on Linux. The out-of-core octree structure is implemented on top of *Berkeley DB 4.8.3*, while *OpenMP* is used for parallelizing blending operations. The automatic masking subsystem is implemented on top of *OpenCV 2.4.3*. RAW color images from the camera are handled using the *dcraw 9.10* library. The SfM software used for image-to-image alignment is *Bundler 0.4.1* [78]. All tests were run on a PC with an 8-core Intel Core i7-3820 CPU (3.60GHz), 64GB RAM and an NVIDIA GTX680 graphics board.

Acquisition. The scanning campaign covered 37 statues, which were scanned and photographed directly in the museum. Fig. 3.12 summarizes the reconstruction results. The geometry of all the statues was acquired at a resolution of 0.25mm using a Minolta Vivid 9i in tele mode, resulting in over 6200 640x480 range scans. The number or scans includes a few (wide) coarse scans which fully cover the statue, that were acquired to help with global scan registration. The scanning campaign produced over 1.3G valid position samples. Color was acquired with a Nikon D200 camera mounting a 50mm lens. All photos were taken with a flash in a dark room, with a shutter speed of 1/250s, aperture f/11.0+0.0, and ISO sensitivity 400. A total of

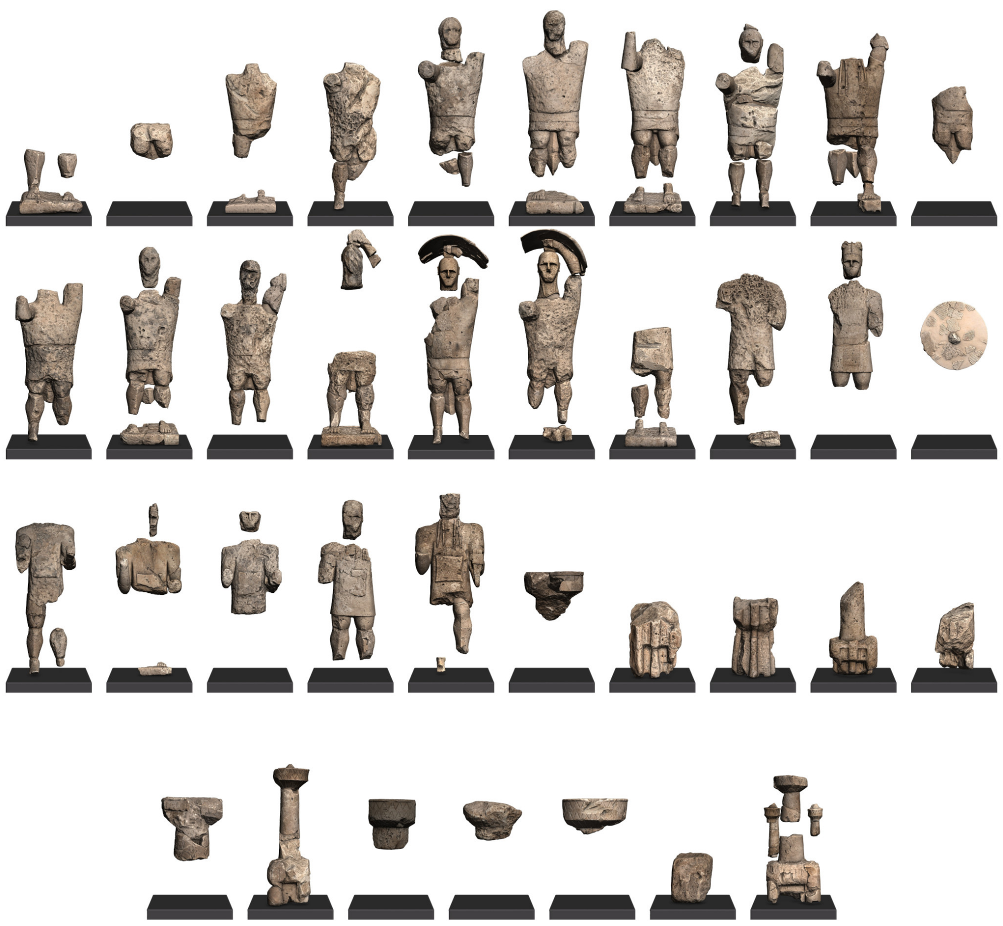


Fig. 3.12.: **Reconstructed Status of the Mont'e Prama complex.** Colored reconstructions of the 37 reassembled statues.

3817 10Mpixel photographs were produced. The on-site scanning campaign required 620 hours to complete for a team of two people, one camera, and one scanner. In practice, on-site time was reduced by parallelizing acquisition with two scanning teams working on two statues at a time. The acquisition time includes scanning sessions, flash photography sessions (in dark room), and coarse alignment of scans using our point cloud editor. Photo alignment using the SfM pipeline was performed after each flash acquisition session, and in parallel to the scanning session, in order to verify whether sufficient coverage had been reached. Average bundle adjustment time was of 2 hours/statue.

Automatic geometric masking.

The quality and efficiency of our automatic geometric masking process was extensively evaluated on a selected dataset, which was also manually segmented to create a ground-truth result. The digital acquisition of selected statue, named *Guerriero3* and depicted in Fig. 3.5, is composed by 226 range maps (54 of which containing clutter data).

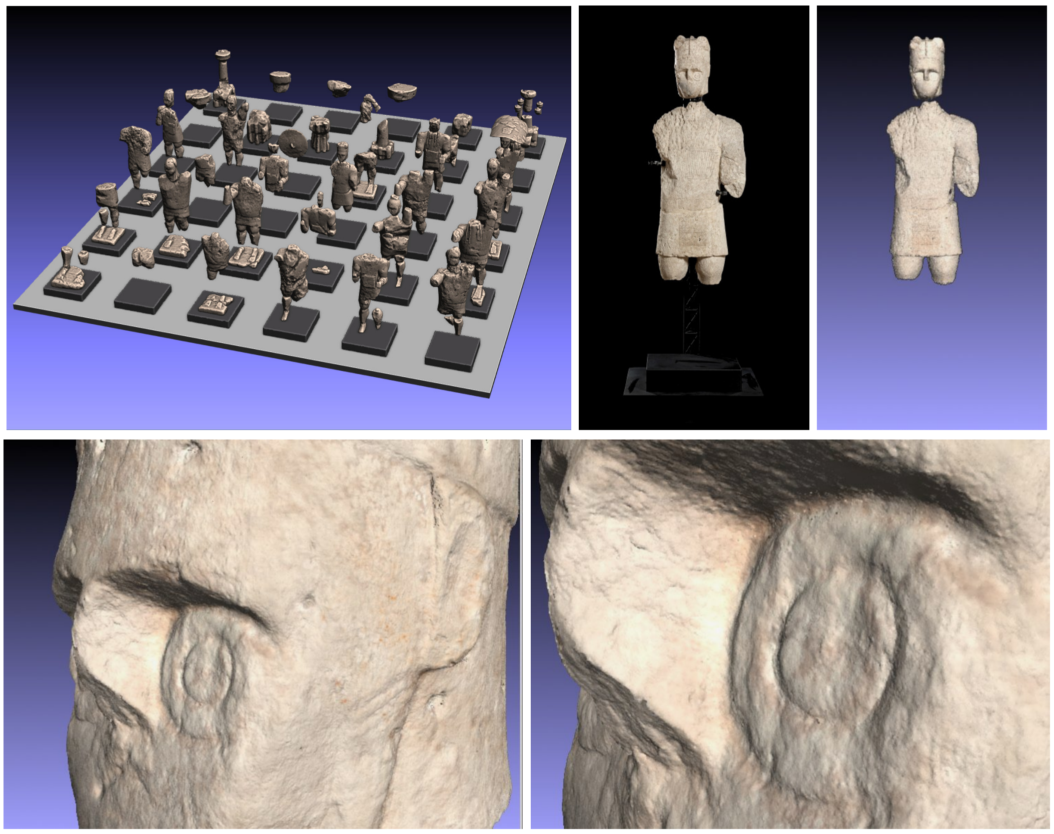


Fig. 3.13.: **Mont'e Prama complex.** From top left to bottom right: the full set of reconstructed statues; original image of the statue "warrior 3"; reconstructed model; closeup on the head of the reconstructed model; closeup on eye of reconstructed model.

Each ground-truth mask was created manually from the reflectance channel of the acquired range map using our interactive mask editor. An experienced user took about 330 minutes to complete the manual segmentation process for the entire statue. For the sake of completeness, we also measured the time required to remove clutter data from the 3D dataset by direct 3D point cloud editing, as done in typical scanning pipelines. Using our out-of-core point cloud editor this operation was completed by an experienced user in about 300 minutes, which is relatively similar to the time required for the manual 2D segmentation approach. By taking into account the relative complexity of the other statues, we can estimate a total time of about 130-150 man-hours for the manual cleaning of the entire collection of statues.

The automatic segmentation process was started by manually segmenting 5 reflectance images using the same editor used for manual segmentation. This training set was used as input for the automatic classifier. The entire process took 9 minutes for the creation of the training set and 6 minutes for the automatic computation of the mask on an 8-core processor. The automatically generated masks were then manually

	Samples	(%)
Model points	51.4M	
Clutter points	790K	
False-Positives	240 (486)	0.03 (0.06)
False-Negatives	35757 (11219)	4.53 (1.42)
True False-Negatives	5639 (2746)	0.68 (0.35)

Tab. 3.1.: **Evaluation of automatic geometric masking.** Results of manual segmentation of a single statue (*Guerriero3*) compared with the results produced by automatic masking. We report the number of range map samples labeled as model ("Model points") and clutter ("Clutter points") in the ground truth dataset, the samples erroneously labeled as statue ("False-positives") or clutter ("False-negatives") in the automatic method, as well as the number of false negative points that really lead to missing data in the combined dataset ("True False-negatives"). Values between parentheses compare the manually refined and the ground-truth datasets, instead of the purely automatic method. Percentages are computed with respect to the number of clutter points.

verified and retouched using our system. This additional step, which is optional, took about 30 minutes. Applying the automatic process to the entire statue collection only took 5 hours, excluding manual cleaning, and a total of 13.5 hours including the manual post-process cleanup: this result is a more than ten-fold speed-up with respect to the manual approaches.

The efficiency of the automatic masking method can be seen from the results presented in Fig. 3.1, which shows the results of the comparison tests between the automatically segmented masks (with and without post-process manual cleaning) and the ground-truth dataset.

More than 95.0% of the clutter samples are correctly labeled. False-positive samples represent extra points which can be easily identified and removed from the automated masks via 2D editing and they are only about 0.03% of the total clutter in the ground-truth dataset. False-negative points represent statue samples that have been erroneously masked; they are about 4.5% (1.4% in the clean-up dataset) of the total clutter in the ground-truth dataset. Since overlapping range maps typically acquire the geometry of same model region from multiple points of view, a false-negative sample is not a problem if its value is correctly classified in at least one mask covering the same area. By taking into account this fact, we verified that the points that were completely missed by the acquisition (*True false negative*) are only 0.68% of the total imaged clutter surface. This check was performed by searching in overlapping scans for samples within a radius of 1mm from each missing sample. Therefore, we can conclude that only a small portion of the surface is missed by the system. Further, Fig. 3.9 illustrates the position of the missing points; from the images it is easy to see that the points in question are often very sparse or represent small boundary area of the model. Thus, their overall effect on dataset quality is quite limited.

Automatic color masking. The quality and efficiency of the color masking process was evaluated in a manner analogous to the geometry masking procedure. The selected statue –"Guerriero3", depicted in Fig. 3.5 – was imaged by 68 photographs (33 of which containing clutter data). Manually masking the images took 181 minutes, while applying the automated process required 9 minutes to generate the training set, 15 minutes to automatically compute the masks on 8 CPU cores, plus a final 30 minutes for the optional manual post-process cleanup. The speed-up provided by our automated procedure is, again, substantial. The semi-automatic masking process for the entire set of statues only required a total of 41 hours (17 hours without the post-process cleaning). By taking into account the relative complexity of the other statues, we can estimate a total time of about 145 man-hours for the manual cleaning of the entire collection of statues.

	Samples	(%)
Model points	220.5M	
Clutter points	12.1M	
False-Positive	381K (334K)	3.16 (2.77)
False-Negative	263K (253K)	2.18 (2.09)
True False-Negative	8642 (7725)	0.07 (0.06)

Tab. 3.2.: **Evaluation of automatic color masking.** Results of manual segmentation of a single statue (*Guerriero3*) compared with automatic masking results. We report the number of colored samples labeled as model ("Model points") and clutter ("Clutter points") in the ground truth dataset, the samples erroneously labeled as statue ("False-positives") or clutter ("False-negatives") by the automatic method, as well as the number of false negative points that really lead to missing data in the combined dataset ("True False-negatives"). Values between parentheses compare the manually refined and the ground-truth datasets, instead of the purely automatic method. Percentages are computed with respect to the number of clutter points.

As illustrated in the table in Fig. 3.2, the color masking procedures achieves results similar to those obtained by geometry masking. Again, about 95.0% of the samples are labeled correctly. In this case, false-positive samples are points where clutter color could potentially leak to geometry areas. These represent about 3% of the clutter area – i.e., below 0.2% of the model area. Instead, false-negative points are statue samples that do not receive color by a given image since they have been erroneously masked; they are about 2.2% (2.1% in the cleaned-up dataset) of the total clutter in the ground-truth dataset, but reduce to negligible amounts when considering overlapping photographs. This is because of the large overlap between photos and the concentration of false negative in thin boundary areas covered from other angles. Sampling redundancy, required for alignment purposes, is thus also very beneficial to the automatic masking process.

Consolidation and coloring. The generated geometry and color masks were used to create digital 3D models of the 37 statues (see Fig. 3.12). After cleaning, all models were imported into our system based on forests of octrees, which was used for all the 3D editing and color blending. We use lossless compression when storing our hierarchical database, thus achieving an average cost of about 38B/sample, with per-sample positions, normals, radii, colors, and blending weights (including database overhead). Disk footprints for our multiresolution editable representation are thus similar to storing single-resolution uncompressed data.

We compared the performance of our system to the state-of-the-art streaming color blender [43]. Our pipeline required a total of 23 minutes for blending the *Guerriero3* statue; as we already mentioned, the pipeline works directly on the editable representation of the model and includes color correction for flash illumination. On the other hand, the streaming color blender required 2.5 minutes for pre-computing the Morton-ordered sample stream and the culling hierarchy, and 26 minutes for color blending. Therefore, the increased flexibility of our system does not introduce additional overhead in the form of processing time nor does it require additional temporary storage – all while supporting fast turnaround times during iterative editing sessions.

Flash color correction proved to be adequate in our evaluation. It produces visually appealing results without unwanted color variation and/or visible seams between acquisitions (see Fig. 3.10 for an example). It is important to note that, while no painting results are currently visible on the statues, including natural color considerably adds to the realism of the reconstruction, as demonstrated in Fig. 3.14.

3.5 Discussion

In this chapter, we have presented a general design for a scalable system for creating, coloring, analyzing, and exploring massive point clouds totally out-of-core. The structure presented is a variation of the Layered Point Clouds approach, which exploits a special ordering of the original points to construct a multiresolution

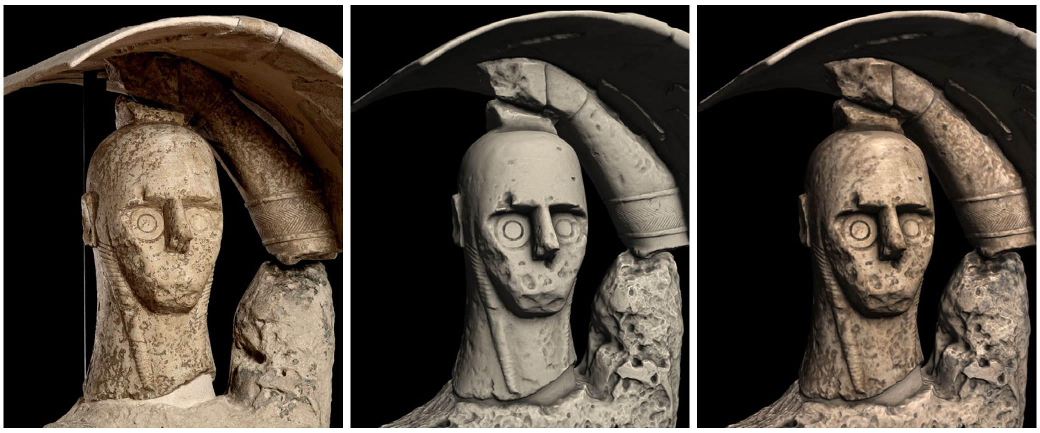


Fig. 3.14.: **Effect of color mapping.** From left to right: original photograph (boxer 16); virtual reconstruction without color; virtual reconstruction with color.

structure, assuming that the original point cloud is uniformly sampled. This is a reasonable assumption for typical scanning datasets. The presented implementation uses a caching system and an optimized I/O layer to support out-of-core algorithms in both CPUs and GPUs. The method, is, however, tuned only for static point clouds. Operations can be performed on it, but just adding or modifying attributes, and not moving or deleting points. This means that not all applications can be directly implemented by modifying the structure on the fly. However, the operations that just modify attributes can be implemented with maximum efficiency. This is an incremental contribution over the state-of-the-art, which was meant to support further research on scalable methods.

A very important example of operations that require enriching and consolidating point clouds arises in cultural heritage, when mixing geometric data acquired with laser scanners with color data acquired with digital photography. In this work, we have introduced an easy-to-apply acquisition protocol based on laser scanning and flash photograph to generate colored point clouds, as well as a novel and practical semi-automatic method for clutter removal and photo masking to generate clean point clouds without clutter using minimal manual intervention. In this approach, geometry masking is applied to the original range scans. The consolidated model is quickly regenerated in our multiresolution format after filtering. Color mapping, instead, can be directly applied to the multiresolution point cloud by updating the color attributed using a color blending approach. Our scalable implementation of the entire masking, editing, infilling, color-correction, and color-blending works fully out-of-core without limits on model size and photo number, as demonstrated on the *Mont ́e Prama* use case.

This chapter has mostly focused on scalable ways to support model creation, with a particular emphasis on point clouds mixed with photographic data. In the remainder of this thesis, I will focus, instead, on ways to scalably render massive models, going beyond those resulting from this pipeline based on real-world sampled data.

3.6 Bibliographical Notes

The point cloud structure presented here is more extensively described in [9, 15, 16]. The chapter of the book "*Effective Big Data Management and Opportunities for Implementation*" [10] describes further applications of this structure to filter the point cloud, as well as extract primary geometrical primitives to CAD models. Instead, The chapter of the book "*Handbook of Research on Visual Computing and Emerging Geometrical Design Tools*" [11] describes how the pipeline is applied to support VR applications in architecture and engineering. The proposed system for exploring point clouds was also shown to be able to run in a client-server configuration by streaming the data chunks and explore remotely the models in a web browser using WebGL [79].

The novel technique for scalable shape and color digitalization of cluttered artwork was published in the *ACM Journal on Computing and Cultural Heritage* [12]. That publication, in addition to the technique described here, also includes an analysis of requirements gathered from end users.

4

Improving scalability through adaptive batching: Coherent Hierarchical Culling for Ray Tracing

The interactive exploration of very large models, including, but not limited to, the high-density sampled models that can be created with the techniques presented in the previous chapter, requires specialized techniques to meet timing constraints. View-frustum and occlusion culling methods, in addition to multiresolution, are commonly used to load and process only the visible part of the scene and thus to make rendering output sensitive. These techniques are particularly effective with object-order rasterization on the standard GPU-pipeline, tuned for object-order streaming of primitive batches (in particular, triangles). In this chapter, we extend this approach to the more flexible ray-tracing setting, proposing a novel generalization of hierarchical occlusion culling in the style of the *CHC++* method. This novel approach exploits the rasterization pipeline and hardware occlusion queries in order to create coherent batches of work for localized shader-based ray tracing kernels. By combining hierarchies in both ray-space and object-space, the method is able to share intermediate traversal results among multiple rays. Then, temporal coherence is exploited among similar ray sets between frames and also within the given frame. A suitable management of the current visibility state makes it possible to benefit from occlusion culling for less coherent ray types like diffuse reflections. Since large scenes are still a challenge for modern GPU ray tracers, our method is most useful for scenes with medium to high complexity, especially since it inherently supports ray tracing highly complex scenes that do not fit in GPU memory. For in-core scenes our method is comparable to CUDA ray tracing and performs up to sabout six times better than pure shader-based ray tracing.

Depth-buffered rasterization and ray tracing are nowadays the two dominant techniques in real-time rendering. In its basic form, rasterization is an object-order approach that determines visible surfaces by going through scene primitives, projecting them to screen and maintaining the nearest surface for each pixel. Ray tracing, on the other hand, is an image-order approach that determines visible surfaces by computing ray-primitive intersections for each pixel.

In principle, rasterization offers more code- and data-cache coherence, because switching primitives and rendering attributes occurs much less frequently, and most operations work on an object-by-object basis on data residing in local memory, without the need to access the entire scene. This explains the success of massively parallel GPU rasterization hardware based on streaming architectures. In contrast, for ray tracing, in order to efficiently compute ray-primitive intersections, data is usually organized in space-partitioning data structures, and the traversal of these data structures results in non-streaming access patterns to the scene geometry. Even though current GPUs support general programming models and allow for programming acceleration data structures and complex traversal algorithms, efficient memory management and computation scheduling is significantly harder than for rasterization, leading to performance problems and/or complications when trying to integrate rasterization and ray tracing within the same application, e.g., to compute complex global illumination.

4.1 Contribution

We address these issues by proposing a ray-tracing technique that is designed to be integrated into the streaming rasterization pipeline. The core idea of the method is to exploit the rasterization pipeline together with occlusion queries in order to create coherent batches of work for GPU ray tracing. By combining hierarchies in both ray space and object space, and making use of temporal coherence, the ray-traversal overhead is minimized, and the method can concentrate on computing ray-object intersections for significantly reduced sets of rays and objects. This batched computation and memory-management approach makes it possible to use the same streaming schemes employed in current rasterization systems also for ray tracing. This opens the door to a flexible integration of rasterization and ray tracing, both for dynamic and out-of-core scenes. We show the efficiency of our method for several ray types like soft-shadow rays and diffuse interreflections. The main contributions of this work are:

- Occlusion culling for ray tracing using the rasterization pipeline, which is up to $6\times$ faster than standalone OpenGL-based ray tracing.
- A means for scheduling visible parts of the scene hierarchy for ray-triangle intersection on the GPU that allows a simple and natural extension to out-of core ray tracing.

Moreover, this chapter shows an efficient OpenGL implementation of the method, with a number of benefits:

- a novel algorithm for automatically identify rays sets that can be intersected by triangles;
- can be easily adapted for dynamic scenes (coarse and "approximate" hierarchy is sufficient);
- performs an efficient parallelization done through the rasterization hardware (so the scheduling and thus, the mapping to the stream cores if automatically manage by he hardware);
- allows to use the method on "legacy" hardware and consoles.

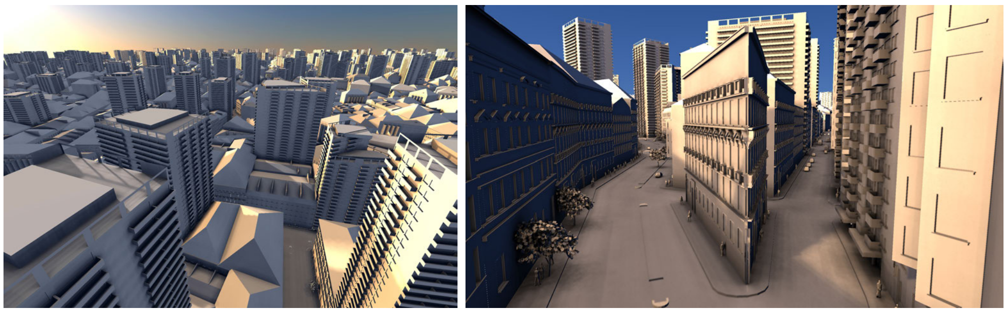
This technique was presented in a joint Eurographics 2015 paper [5]. As for the distribution of work, I contributed to the design of the method, fully designed and implemented the ray-tracing subsytem, and devised and implemented the majority of the evaluation.

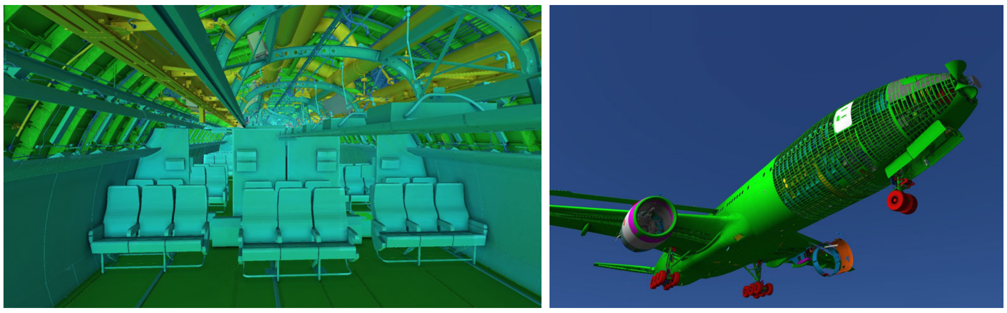
4.2 Related Work

Our work generalizes hierarchical occlusion culling, a technique traditionally used for accelerating *rasterization*, to incorporate *ray-tracing* effects. This allows to schedule the intersection work in batches, reduces the traversal stack size needs, achieve naturally out-of-core for massive models, and combine with am standard graphic pipeline in GPU. These topics have been studied extensively in the past. In the following, we discuss the most relevant work in these two well-studied fields, particularly those targeting the acceleration of both ray-tracing and rasterization techniques.

4.2.1 Ray tracing data structures and acceleration

Extensive research has been performed with the aim of accelerating the computation of intersections of rays with the scene. The commonly used acceleration data structures include uniform grids, octrees, kd-trees, and bounding-volume hierarchies (see established surveys for more details [80, 81]). One of the keys to efficiency is the quality of the acceleration data structure, which, for the case of hierarchies, is usually constructed according to the Surface Area Heuristics (SAH) [82]. Related to


(a) City-200 (138M triangles, 7.89GB)


(b) Boeing 777 (350M triangles, 18.9GB)

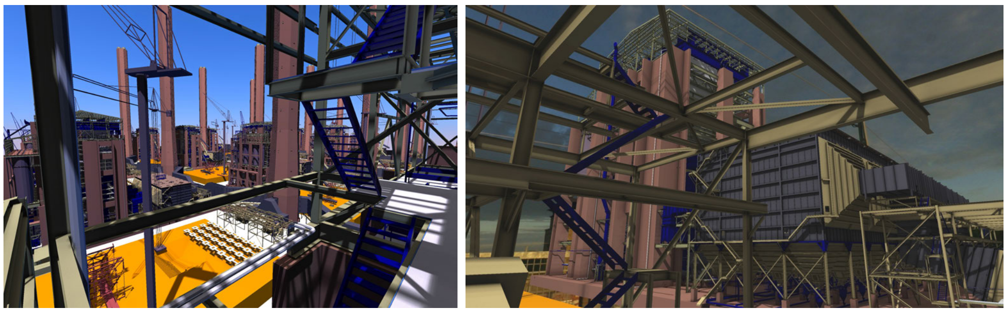

(c) Powerplat ×16 (205M triangles, 11.4GB)

Fig. 4.1.: **Massive 3D models frames rendered using our interactive OpenGL ray tracer using the CHC+RT algorithm.** Based on hierarchical occlusion culling allows a simple scheduling scheme for managing out-of-core scenes and also significantly accelerates OpenGL-based ray tracing in complex scenes.

our approach are the methods based on batched processing of rays, such as cone tracing [83], beam tracing [84, 85], or more generally the stream-ray architecture [86]. Mora [87] proposed a method which avoids organizing the scene in a spatial data structure, but instead sorts large groups of rays together with the scene geometry on the fly. The method of Bolous et al. [88] uses coarse-grained visibility tests to reduce the active ray set for CPU packet tracing, which have a similar purpose as the hardware occlusion queries used by CHC+RT. While our method shares the idea of packet tracing, it differs particularly in the fact that it is designed for integration with GPU-based rasterization and does not use explicit ray bounding primitives or other per-packet information.

By the other side, with the recent development in graphics hardware, a number of alternative methods directly implemented on the GPU have been proposed. Recent advances make possible to do real-time ray tracing on the GPU [89, 90, 91, 92]. In this scope, the traversal of spatial data structures becomes one of the most critical algorithms to be optimized, as could be completely related to the performance. While these methods are very fast, they usually require that the scene and the associated acceleration data structure is fully available in GPU memory, which makes it difficult to handle large scenes. Our technique, in contrast, naturally leads to more coherent data access patterns and to batch-based memory management.

4.2.2 Mixing ray tracing and rasterization

Several algorithms have tried to use the limited features of rasterization-based rendering for ray tracing. Most notably, Carr et al. [93] proposed the Ray Engine, which achieves ray tracing effects by rendering a screen-sized quad and computing ray intersections for each scene triangle. The brute-force version of this process is inefficient and uses huge amounts of fill rate. Roger et al. [94] improves on this method by building a hierarchy of cones over the rays and using them to reduce the number of computed intersections. In our algorithm, we conservatively cull those pairs of triangle batches and screen-space patches where the geometry is not intersected with respect to the screen-space patch. Also, several techniques have been proposed to compute approximate ray tracing effects on the GPU [95, 96].

Novak and Dachsbacher [97] use rasterization to construct a hierarchy containing resampled scene geometry that can be processed by standard ray tracing methods. Davidovic et al. [98] proposed a 3D rasterization method designed for coherent rays. The authors show that there exists no fundamental difference between rasterization and ray tracing of primary rays, but a continuum of approaches that blend seamlessly between both paradigms. Our algorithm further explores the space between both paradigms by using the fixed-function pipeline and the z-buffer for arbitrary rays. Recently, Zirr et al. [99] proposed a method for ray tracing in a rasterization pipeline, using a voxel scene approximation to accelerate the traversal. A voxel representation is also used by Hu et al. [100], using the A-buffer to search ray-triangle intersections in a shader. In contrast to these methods, we support casting arbitrary rays, compute exact ray-scene intersections, and support out-of-core rendering.

4.2.3 Out-of-core ray tracing

Most of the work on rendering large scenes has focused on combining CPU techniques with out-of-core data-management methods (see a survey on massive-model rendering [101]). Notable examples are methods using a scheduling grid for rays to improve the coherence of scene accesses (e.g., [102, 103]) and methods exploiting level-of-detail representations [104, 105, 106]. More recent work also combined CPU/GPU computation using distributed computing 5approaches [107, 108]. In this context, Pantaleoni et al. [109] proposed the PantaRay system, targeted at fast relighting of complex scenes based on occlusion caching. Garanzha et al. [110] used a complex data structure similar to PantaRay for CentiLeo, a commercial progressive out-of-core path tracer based on CUDA. Instead, our method subdivides the scene into adaptively sized batches of visible geometry by using occlusion culling, allowing simpler and more flexible data management that yields a natural out-of-core extension.

4.2.4 Visibility and rasterization methods

Numerous methods for the acceleration of rasterization using visibility have been designed. Scene simplification techniques compute different levels of detail (LOD), which allow limiting the complexity of the rendered scene, such as Adaptive Tetra-Puzzles [111], GoLD [112], or Far Voxels [113]. View-frustum and occlusion culling methods are commonly used to rasterize only the visible part of the scene and thus to make rendering output sensitive. In particular, hardware occlusion queries can be used to efficiently test the visibility of simple proxy objects, such as bounding boxes, against the depth buffer before rendering the real geometry [114, 115, 113, 116, 13]. Specific methods have been designed for accelerating rasterized shadows, as Bittner et al. [117] using occlusion culling to accelerate shadow-map rendering for complex scenes. A general technique commonly used to compute complex effects in the rasterization pipeline is deferred shading, proposed by Deering et al. [118] and generalized by Saito and Takahashi [119], and used in several methods discussed above. Recently, the concept of *deferred shading* has been improved by tiled and clustered shading by Olsson et al. [120]. Our novel algorithm also exploits this technique and generalizes the described culling methods by handling arbitrary primary and secondary rays with occlusion queries.

4.3 Overview

The diagram in Figure 4.2 provides an overview of our method. We first render the scene either by rasterization or tracing primary rays. Then the method applies a number of additional ray tracing-based shading passes, which add the required illumination effects to the rendered image. In each pass, we first generate the rays to be cast and store them in a (full-screen) ray texture with one ray per pixel. Thus, the rays are directly associated with the pixels they should contribute to. We generate 3 ray types in this phase: soft-shadow rays, ambient-occlusion rays, and diffuse rays. Note that while a ray is stored in the pixel it will finally contribute to, it could start anywhere in the scene. Each pass uses one ray texture, and thus evaluates one ray contributing to the pixel.

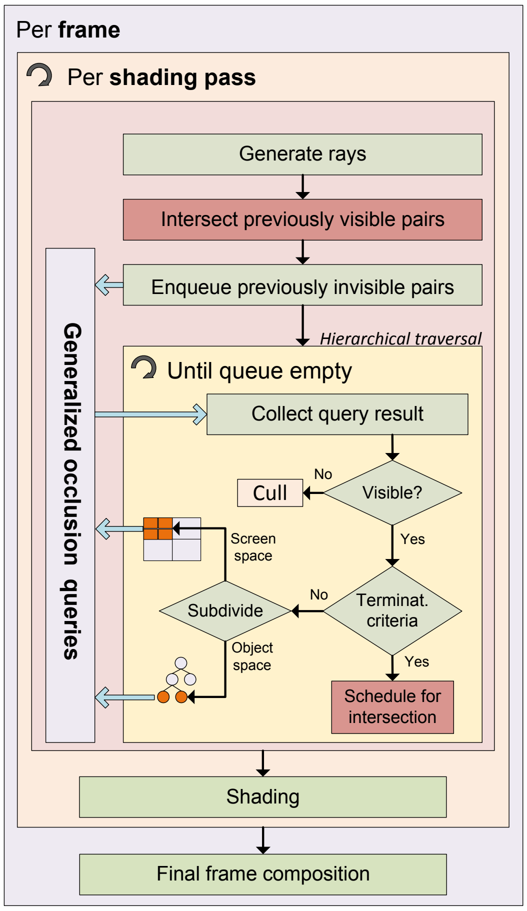


Fig. 4.2.: **Overview of the proposed algorithm, CHC+RT.**

The core part of our method is computing ray-triangle intersections in the rasterization pipeline using the given ray texture and a CPU-side scene hierarchy. The hierarchy can be a Bounding Volume Hierarchy (BVH) used by the CPU to perform view-frustum and occlusion culling. The basic operation that we use is determining whether rays corresponding to a certain screen-space tile intersect the bounding box of the given node of the BVH. We call this pair (the screen-space tile and the BVH node) a *query pair*. The algorithm starts with the query pair given by the tile representing the whole screen (all rays) and the bounding box corresponding to the root of the BVH (all triangles). The potential intersection of rays and the bounding box is evaluated in a shader that computes the nearest intersection of each ray with the given box. This distance is passed as a z-value to be compared with the already evaluated nearest distance using the hardware z-buffer. We detect the rays intersecting the box by issuing an occlusion query that encapsulates the query pair processing. If the result of the occlusion query indicates a non-zero number of intersections, we either proceed by subdividing the screen-space tile or the BVH node and repeating the process for the newly created query pairs. This hierarchical traversal is indicated by the yellow box in Figure 4.2. The subdivision is terminated when reaching tiles of a certain minimum size and when meeting a termination criterion of the BVH. Then the actual ray-triangle intersections are computed.

We exploit temporal coherence by maintaining generalized visible and invisible fronts for the current ray set (stored as *previously visible pairs* and *previously invisible pairs*, as shown in Figure 4.2). Using this method we can reduce the number of intersection tests and also eliminate stalls caused by the latency occlusion queries. One key to the efficiency of our method is that the occlusion-culling phase computes a relatively coarse-grained cut in the query-pair hierarchy. We chose to use GLSL shader-based traversal to compute the fine-grained ray-triangle intersections of each visible subtree of the BVH. GLSL is very efficient in rendering small subtrees due to the high cache coherence of the traversal stack. The parts where query pairs are scheduled for intersection are shown in red in Figure 4.2. While our method is conceptually similar to hierarchical occlusion culling for rasterization, the main difference is that the occlusion queries are generalized to arbitrary rays, and that we also maintain a hierarchy over screen space to localize the ray contributions.

4.4 Hierarchical Occlusion Culling for Ray Tracing

This section describes algorithmic and implementation details of the proposed method. We first describe the main components of the algorithm. Then we describe an optimized version of the method using temporal coherence.

4.4.1 Generalized Occlusion Queries

In our method, we use occlusion queries to cull those sets of rays and triangles that cannot intersect. The occlusion queries used in our method can be seen as a generalization of classical hardware occlusion queries [115]. Traditionally, occlusion queries handle visibility from the camera, and thus they deal with a well-defined set of primary rays enclosed in the viewing frustum. In ray tracing, we deal with arbitrarily distributed rays, and thus we have to be able to determine which rays intersect the given geometry using some other means than simple projection of the geometry and its rasterization.

Similar to classical rasterization, we use a depth buffer to store the nearest intersection of each ray with the part of the scene processed so far (recall that rays are associated with pixels). We subdivide the screen into tiles corresponding to packets of rays. For each tile, we use an occlusion query to check if the bounding volume of an object intersects the rays enclosed by the tile. If there is at least one intersection, the query returns a non-zero value and we proceed by calculating the actual ray-triangle intersections. This step can be easily evaluated using a fragment shader in which we pass the bounding volume (axis-aligned box) as a shader parameter. The shader evaluates the ray/box intersection and returns the distance of the intersection as the depth value of the fragment. The query can thus count the number of fragments having nearer intersections than those stored in the z-buffer so far. So the main difference to classical occlusion queries is that the z-buffer values do not represent camera depth values, but distances along rays. They are implemented for a particular query pair by rasterizing the screen-space quad and passing the AABB of the associated BVH node as shader parameter. Since the depth buffer is initialized with the intersection values of the previously visible geometry, occlusion culling can be performed by enabling the depth test in OpenGL.

4.4.2 Shader-based Ray-Triangle Intersection

The visibility in the occlusion-query stage corresponds to a coarse cut in the BVH, making the method less sensitive to spatially incoherent ray packets. Once we reach the termination criteria based on the optimal height and number of triangles in a subtree, it is subsequently (but not necessary immediately) a termination node in the BVH, the subtree is subsequently scheduled for intersection. The actual ray-triangle intersections are computed in a fragment shader executed for a given screen-space tile. The geometry (triangles) is stored in a texture buffer object and passed as a shader parameter. A naive version of our method would render for each triangle in the scene a full-screen quad covering all rays, which basically corresponds to the Ray Engine algorithm [93]. The shader evaluates the intersection of the ray with the triangle and passes the intersection distance as the depth value of the fragment. If this intersection is closer to the ray origin than the one computed so far, the depth buffer entry is automatically updated. The actual shading is deferred to the moment when all scene geometry is processed and the final nearest intersections have been determined.

4.4.3 Hierarchical Occlusion Culling

The two above-described principles (generalized occlusion queries and shader-based ray triangle intersection) can be used together in an algorithm which processes both the rays and the triangles hierarchically. The hierarchy of rays is defined implicitly by a quadtree-based screen-space subdivision, the hierarchy of triangles is defined by a bounding-volume hierarchy (BVH). We propose a generalization of hierarchical occlusion culling, with the main difference that the query objects are not BVH nodes, but *query-pairs* consisting of a BVH-node and a screen-space tile.

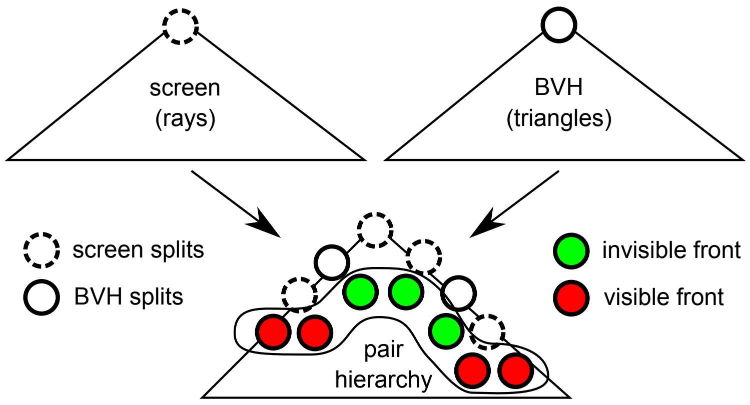


Fig. 4.3.: **Illustration of the query-pair hierarchy.** The interior nodes of the hierarchy correspond to either screen-space splits or object-space splits. The leaf nodes belong to either the visible front (rays and triangles that can intersect) or the invisible front (rays and triangles that cannot intersect).

The queries are always performed on query pairs defined by a screen-space tile and a world-space bounding box.

4.4.4 Traversing the Query-Pair Hierarchy

When the result of the query indicates an intersection, we have to subdivide the query pair and construct new query pairs to refine the intersection results. Here, we have to decide between two choices – subdividing in screen space, and creating 4 new query pairs, or subdividing in object space, and creating two new query pairs (see Figure 4.3). This decision influences in how many steps a particular subtree of the query-pair hierarchy can be culled as being invisible, and hence it is important for the performance of the traversal algorithm. A split in object space can potentially reduce the *intersection cost*, while a split in screen space can potentially reduce the area and hence the *query cost*. Note that a split in object space can potentially double the overdraw and thus the query cost since both child nodes have to be queried for the same tile. The area also roughly corresponds to the probability that a BVH node is intersected [82] and the query fails. Hence object space splits are preferred until the BVH nodes are relatively small.

We found that the best performance can be achieved by comparing the areas of the screen-space tile and the object-space node of a pair, which are connected to the query cost and intersection probability, respectively [82]. The areas are normalized by the area of the bounding box of the BVH root (A_{root}) and the full screen extent (A_{screen}). They are also weighted by a hardware-dependent factor t_q, which we set to 0.5 in all our comparisons (favoring object-space splits in the beginning). We always split in the domain where the corresponding ratio is larger, i.e.:

$$\frac{A_{bvh}}{A_{root}} > t_q * \frac{A_{tile}}{A_{screen}} \begin{cases} \text{true:} & \text{split in object space} \\ \\ \text{false:} & \text{split in screen space} \end{cases}$$

This heuristic aims to keep a rough balance between the extents of the screen-space and object-space domain within a query pair. Note that it would be more consistent to compare both areas in world space, but until we compute the intersections we do not know the world-space extent of the bounding volume of rays covered by a screen-space tile.

4.4.5 Exploiting Temporal Coherence

The hierarchical algorithm described above can be improved by exploiting temporal coherence among rendered frames. In particular, similar to occlusion-culling algorithms, we can initialize the content of the depth buffer by first evaluating intersections using all visible query pairs from the previous frame. Note that this assumption does not invalidate the correctness of the results since the actual ray-triangle intersections always use data for the current frame, i.e., rays generated for the current frame and triangles at correct positions for the current frame.

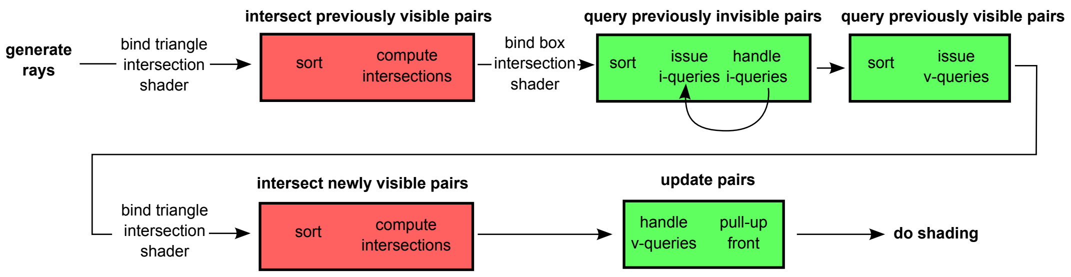

generate rays
bind triangle intersection shader
intersect previously visible pairs
sort
compute intersections
bind box intersection shader
query previously invisible pairs
sort
issue i-queries
handle i-queries
query previously visible pairs
sort
issue v-queries
bind triangle intersection shader
intersect newly visible pairs
sort
compute intersections
update pairs
handle v-queries
pull-up front
do shading

Fig. 4.4.: **Overview of the pipeline.** Parts of the method computing and storing the actual ray/triangle intersections are shown in red, while the steps dealing with ray traversal and culling are shown in green.

After processing previously visible pairs, we issue queries on previously invisible pairs to verify if they stay invisible. If any previously invisible pair becomes visible, we process it hierarchically and collect all newly visible pairs for which ray-triangle intersections should be computed. At the end of the frame, these new intersections are evaluated, and finally the visibility front consisting of both visible and invisible pairs is updated. An overview of the different steps of the coherence-based algorithm is shown in Figure 4.4.

4.5 CHC+RT Implementation

In this section we address the details regarding the actual OpenGL implementation of the method and its optimizations.

4.5.1 Hierarchical Traversal

The pseudo-code of our traversal algorithm is shown in Algorithm 4.7. In analogy to occlusion culling [115, 13], we talk about visible/invisible nodes. For a node in the query-pair hierarchy, this means that the occlusion-query result is either positive (there are potential intersections, hence visible) or zero (there are no intersections, hence invisible). Most optimizations proposed in the CHC++ algorithm [13] can also be used for CHC+RT for reducing the overhead of generalized occlusion queries. Our algorithm starts from the previous cut of visible leaf nodes and invisible (leaf or interior) nodes. It consists of three phases.

- **Phase 1.** All previously visible leaf nodes are scheduled for ray-triangle intersection. This initializes the z-buffer with the intersections from those query pairs which have been visible in the previous frame and allows us to exploit ray occlusions.
- **Phase 2.** The visibility status is queried for the previously visible and invisible pairs. First, all the previously invisible nodes are queried (line 4) and enqueued in the so-called *query queue*. Since the visibility status of the previously visible nodes could have changed from the previous frame, we lazily query them and update their visibility status at the end of the frame. Analogous to CHC++, we use a stratified sampling scheme by randomizing the first frame where the node is queried (between $1..n$ frames). Thereafter, the node is queried every n frames. In our tests, we set n to 3.
- **Phase 3.** The actual hierarchical traversal is where newly visible pairs are detected and the invisible front is updated for the next frame. It starts by fetching the query results one at a time: If the visibility did not change from the previous frame, we are finished. If a node turned visible, we further subdivide the node and enqueue the child nodes, until either a node is found to be invisible or a termination criterion is met (in which case we set it to visible). During the traversal, whenever we have compiled more than m visible nodes (where $m = 16$ in our tests), we compute the ray-triangle intersections of the nodes found newly visible (line 22).

Each frame, invisibility information is pulled up in the hierarchy. This means, if all child nodes are invisible, the parent node is set to invisible and the child nodes can be deleted. This can be continued recursively until we encounter a visible child node, but we restricted it to at most one level per frame to avoid fluctuations. Note that for this purpose we maintain a query-pair hierarchy in order to recall the history of the subdivision and quickly determine which pairs to merge during the pull-up phase.

A useful optimization for previously visible node queries (line 6) are the so called *multi-queries* [13]. Their purpose is to reduce the overdraw caused by queries overlapping in screen space, which we identified as the main source of performance

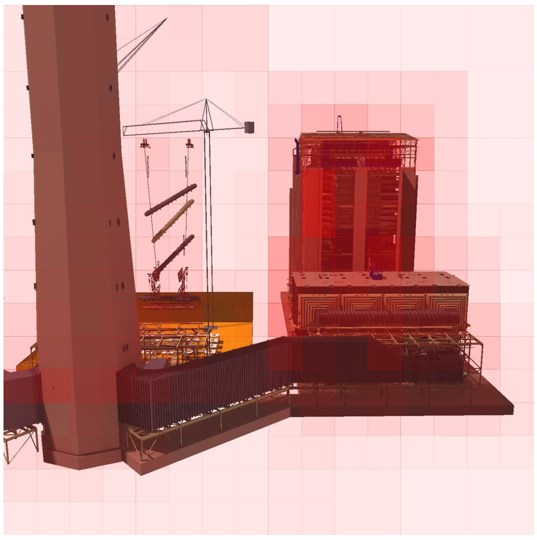
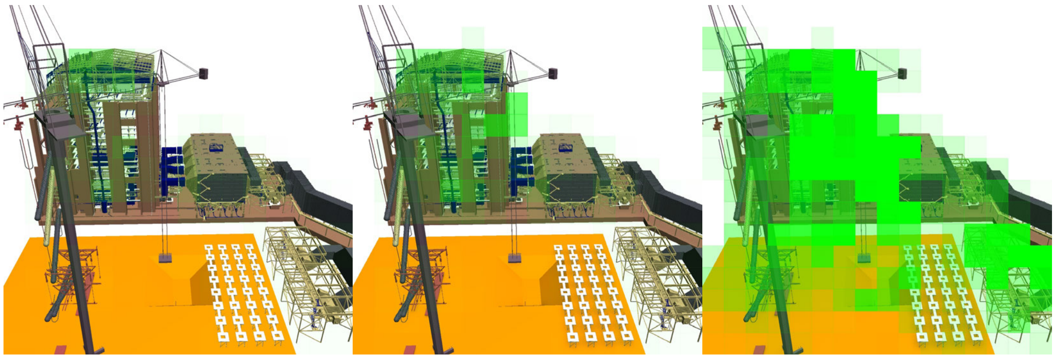


Fig. 4.5.: **Occlusion-query overdraw** for primary rays, where reddish regions (those with the highest geometric complexity) have high overdraw.



(a) Ray length: 1 (b) Ray length: 10 (c) Ray length: 1000

Fig. 4.6.: **Reduced spatial coherence using diffuse rays.** Illustration of the reduced spatial coherence using different diffuse rays lengths, 1, 10 and 1000, respectively. An opaque green screen-space tile means that its rays potentially intersect $> 2M$ triangles.

overhead (see Figure 4.5). Multi-queries compile many previously invisible pairs projecting to the same screen-space tile into a single occlusion query over multiple bounding boxes. If this query is successful and all nodes stay invisible, many nodes have been handled in a single shader pass, which is more efficient than using a separate pass for each individual node. Note that in case the query fails (line 14), multi-queries have to be handled differently. In particular, all nodes have to be queried individually, since we don't know which of the nodes has become visible. We also exploit the tighter-bounds optimization of CHC++ by always querying the bounding boxes of the two children of a BVH node instead of the node itself.

```
// Phase 1: intersect visible pairs
sort previously visible pairs by tile
for all previously visible pairs do
    compute ray-triangle intersections;
end
// Phase 2: query pairs
sort previously invisible pairs by tile
for all previously invisible pairs do
    compile (multi-)query and enqueue;
end
for previously visible leaves do
    update visibility status every $n$ frames;
end
// Phase 3: recursive traversal
while not query queue empty do
    fetch next query result;
    if query result == visible then
        if terminationReached(query pair) then
            add node to newly visible leaves;
            if newly visible leaves $> m$ then
                for newly visible leaves do
                    compute ray-triangle intersections;
                    clear newly visible leaves;
                end
            end
        end
    else
        subdivide(query pair);
        for all children do
            issue occlusion query and enqueue;
        end
    end
end
// intersect remaining visible leaves
for newly visible leaves do
    compute ray-triangle intersections;
end
```

Fig. 4.7.: **Pseudo-code of the traversal algorithm** described in 4.5.1.

4.5.2 Ray-Triangle Intersections

The ray-triangle intersection is the last stage of our algorithm. As we compute ray-triangle intersections for localized subsets of our scene geometry, we can achieve good data-access coherence and can employ streamlined acceleration data structures. The efficiency of the implementation of the proposed method greatly depends on the CPU/GPU data management, i.e., the way in which we pass the shader data, the actually used intersection algorithm, and how we organize the rendering calls of the shader kernels.

GLSL shader. We chose to use a GLSL shader-based traversal algorithm for the final intersections of the termination nodes in the BVH. The shader uses the speculative while-while ray traversal proposed by Aila and Laine [89] for CUDA-based ray tracing. The traversal always fetches both child nodes and traverses the nearer child first in order to exploit occlusion. We cache the intersected leaves for delayed coherent ray-triangle intersection. We observed that the optimal value for this leaf cache was 2 on our hardware. The cache size is small but nevertheless crucial, as omitting the cache causes a slowdown by approximately 30%.

In order to show the flexibility of our method, we selected for this paper the simplest approach of performing the testing at the granularity of BVH leaves and to just stream over the contained triangles for each ray. While this stage is comparable to the Ray Engine algorithm [93], our algorithm has three important optimization steps:

1. we do not apply the method to the individual triangles, but we always query batches of n triangles at once (n 200)
2. In the shader, we test the intersection of a ray against the bounding box first before actually testing the triangles – this process is efficient since often we can skip the fragment program before entering the loop and neighboring rays usually have enough coherence to keep the thread divergence low. This basic approach can be easily extended by coarsening the granularity of our intersection queries (using small subtrees instead of leaves), and using flat acceleration structures.

CPU-GPU transfer. We pass the geometry to the shader using texture buffer objects in GLSL. For the in-core version, we simply allocate two texture buffers: one for the BVH and one for the geometry. Since we currently only allow diffuse materials, we store a diffuse color per triangle in the alpha channel of the RGBA texture used for storing the geometry. Through the use of uniform buffer objects (UBOs), and managing the UBOs as a cache, we can gracefully move from pure direct rendering using a streaming model to a pure retained graphics model, in which all the geometry is defined up-front in a set of UBOs.

Termination criteria. We use 3 different termination criteria for the traversal of the query-pair hierarchy. One is connected to the termination in the screen-space hierarchy, the other two to the termination in the BVH hierarchy. In our experiments, the optimal size of a screen-space tile seemed independent of the chosen resolution of the actual render target. In our case, we set the minimum tile size to 200^2 pixels in

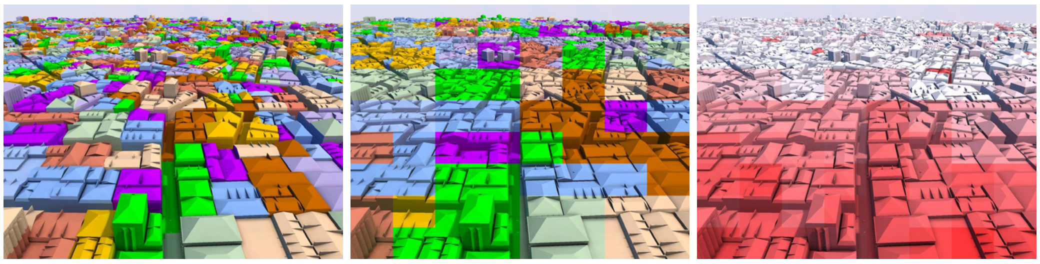


Fig. 4.8.: Left: Visualization of the BVH subtrees that will be scheduled for intersection in the fragment shader. Center: Visualization of the screen space subdivisions. Right: Visualization of the number of subtrees compiled in each batch (1 (red) – 24 (white)) for per-tile based batching.

all experiments, meaning that each tile covers less than 2% of the screen at Full-HD resolution. The key termination criterion in the BVH hierarchy turned out to be the maximum subtree height (i.e., the number of traversal steps until the farthest of the leaf nodes can be reached). The reason is that the GLSL shader-based traversal is very sensitive to the maximum stack size, which has to be at least as large as the maximum subtree height. The optimal granularity of the subtrees depends on various factors like the presence of occlusion. In our experiments, we found that setting the maximum subtree height to 24 levels works well in many cases. Figure 4.8 shows the subtrees induced by our termination parameters. Another termination criterion is the maximum number of triangles per subtree, which becomes important in the out-of-core scenarios. We set it to $1M$ triangles in all our tests.

By-Tile sorting. In our experiments it turned out to be inefficient to schedule the visible subtree nodes of the BVH separately for ray-triangle intersection. Instead, the GPU is better utilized if the contribution of several subtrees to the same screen-space tile is computed in a single shader call. For this purpose we sort the nodes scheduled for intersection by screen-space tile (line 2 in Algorithm 4.7) and then pass an array with the maximum number of 24 node ids to the shader, together with their bounding boxes. The shader then tests the bounding boxes for intersection and starts the traversal for all nodes that pass the intersection test. A visualization of this method is shown in Figure 4.8. The right image visualizes the number of subtrees that can be handled in a single shader pass, and how this number increases with distance. Apart from the better shader utilization, another benefit of this approach is that we can better exploit occlusion within the shader. For certain ray types like shadow or primary rays, this approach can be optimized further by passing the nodes in an approximate front-to-back order.

4.5.3 Ray Generation and Scheduling

In each render pass we generate a single ray direction for primary rays as well as shadow, ambient occlusion, and diffuse rays. The performance of our method benefits from *temporal coherence* and to a lesser degree *spatial coherence*. Less spatial coherence leads to less efficient pruning of invisible subtrees, as shown in Figure 4.5. We take this into account already during ray generation. An alternative possibility would have been to use ray sorting on the generated rays.

Spatial coherence. For both ambient occlusion (or diffuse rays, respectively) and shadow rays, we generate the samples in a stratified fashion. Using a Halton sequence, the same ray direction is generated for each pixel and perturbed with a random per-pixel offset. The degree of randomization depends on the number of rays shot. To achieve this for ambient occlusion and diffuse rays, we apply a random per-pixel rotation to the ray in tangent space, as proposed by Mittrig et al. for SSAO [121]. The maximum angle is chosen so that the samples can cover the whole hemisphere.

Temporal coherence. For primary and shadow rays, ray directions are usually sufficiently coherent so that we maintain a single visibility status for all shadow rays. For ambient occlusion rays and diffuse rays, the ray directions exhibit more variation. We can nevertheless enforce temporal coherence by using a separate visibility status per ray direction, which contains all query pairs in the visibility front. Since we use a coarse hierarchy in screen space and object space, it is easy to keep track of many such cuts.

4.5.4 Out-of-Core Ray Tracing

The method naturally allows for out-of-core ray tracing, with the possibility of rendering potentially unbounded scenes. This is difficult to achieve with current GPU ray-tracing architectures. In the best case, we assume that (most of) the working set required for computing a given frame fits in GPU memory, while the entire scene does not. By using a cache of recently used termination nodes on the GPU, we can avoid transferring to the GPU the geometry that is already in the cache. Note that this sort of memory management requires only minor modifications to the method shown in Algorithm 4.7, and can be managed inside the *compute ray-triangle intersections* function. In our current implementation, we use a simple round-robin style cache management for the BVH and geometry data of the visible subtrees. When caching an out-of-core node, the data is simply written in the next free slot in the texture buffer. When the end of the buffer is reached, we start overwriting the data from the beginning and mark the overwritten nodes as out-of-core. As the only extension to the core algorithm shown in Algorithm 4.5.2, we sort geometry that is scheduled for intersection by their out-of-core status, i.e., visible nodes that have their data currently cached on the GPU are scheduled for intersection first. Note that we still use per-tile sorting among cached nodes.

4.6 Analysis

At the core of our technique is a novel scheduler and memory manager for fine-grained ray-tracing computations that exploits coarse-grained hierarchies in both object space and screen space. The screen-space hierarchy significantly improves scheduling and speeds up rendering. E.g., for the Powerplant scene it results in a speedup by a factor of 3 for Full-HD. Moreover, a screen-space hierarchy makes the method well suited to the current trend towards larger resolution displays (4K and above). By working at a coarse grain, we can amortize the cost of taking decisions over a large number of ray-primitive intersection queries, and use an efficient and flexible adaptive-loading architecture working on optimized geometry batches.

4.6.1 Problem-domain pruning

A good insight into the principle of the method and its potential strengths and weaknesses can be obtained by analyzing the coverage of the whole ray-triangle intersection domain by the query pairs. For this analysis, we express this domain using a matrix in which each ray corresponds to a row in the matrix, while each triangle corresponds to a matrix column. When computing the nearest intersections of rays and triangles, there will be a single unique intersection in each row of the matrix, while there can be many intersections for each column (a triangle can define a nearest intersection for many rays). Our query pairs need to be constructed in a way that every potential intersection is correctly determined. In other words, the query pairs have to fully cover the whole matrix. We can observe an example of such a matrix and its coverage by query pairs in Figure 4.9. This matrix is generally very sparse. The coverage of the matrix by query pairs depends on two main factors: (1) the coherence of intersections and (2) on how densely the rays sample the scene. The triangles are sorted in the BVH and therefore, for similar rays the intersections

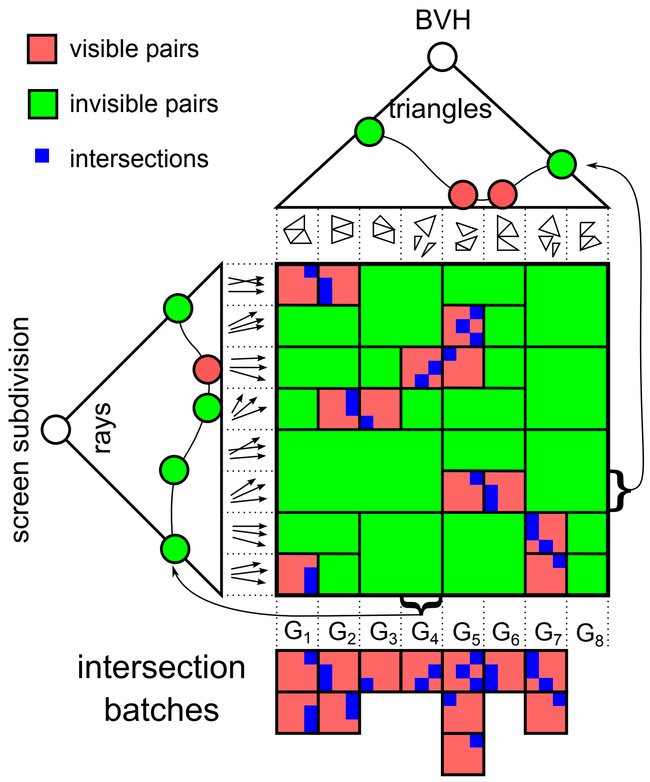


Fig. 4.9.: Intersection between object-space and ray-space subdivisions. Illustration of the coverage of the whole domain of ray-triangle intersection by constructed pairs. Pairs indicating potential ray-triangle intersections are shown in red, while the pairs for which the geometry bounding boxes do not intersect the rays are shown in green. The actual ray-triangle intersections are shown in blue. The figure shows batches of visible pairs (G_1-G_8) used for computing ray-triangle intersections. Note that the geometry G_8 is not used in any batch, meaning that it is not intersected by any ray and will not be scheduled for intersection. Note the two example cuts in the hierarchies: the cut in the screen-space subdivision shows query pairs corresponding to a BVH termination node, and the cut in the BVH shows query pairs corresponding to a given screen-space tile.

should cluster around similar triangles, creating a compact intersection cluster which can be covered by a few query pairs. However, if the rays are highly incoherent, the coverage by query pairs will become more complex. Note that even then, the matrix will be sparse, and at some point we will be able to prune most of the intersection domain if enough query pairs are used.

Another interesting observation follows from the analysis of rows and columns of the matrix. In particular, the number of query pairs covering a row of the matrix directly corresponds to the overdraw of the corresponding screen pixel. The visible query pairs (shown in red) will cause ray/triangle intersections to be executed for this pixel and passed to the z-buffer, while the invisible query pairs (green) will execute the ray/box intersection verifying the invisibility of the associated geometry by the given ray. Looking at the columns of the matrix, we can observe the number of query pairs (screen-space tiles) needed to handle the given geometry (triangle batch). The visible query pairs (red) are those for which the actual ray-triangle intersections are computed, while the invisible query pairs show the occlusion queries issued for rays which do not intersect the triangle batch. Note that the size of the rectangles shown in the matrix depends on the depth of the corresponding query pair in the two hierarchies. Thus the coverage of the matrix by query pairs also visualizes the double-hierarchical cut on which our algorithm operates.

4.6.2 GLSL rendering

Our method is not necessarily bound to a specific implementation, and a CUDA version of the algorithm is definitely possible. Nonetheless, focusing on GLSL in this work provides specific advantages. First of all, by explicitly using a rasterization platform, we better convey the underlying idea that there is a continuum between rasterization and ray-tracing approaches. We aim to foster further research in the area of hybrid rendering by showing how techniques from the rasterization world, such as coherently scheduled visibility algorithms, batched computation and out-of-core rendering, can successfully improve ray tracing. Second, GLSL simplifies the implementation through features of the fixed-function pipeline. For instance, we can use automated shader scheduling instead of implementing explicit schemes, and, while Z-buffering and visibility queries can be realized in CUDA, using GLSL avoids the need to craft efficient synchronization methods using atomic operations. Finally, a GLSL implementation has the additional benefit to be less hardware dependent with respect to CUDA and to simplify integration into classic OpenGL rendering pipelines.

4.6.3 Limitations

Our framework currently only supports static scenes, but an extension to fully dynamic scenes would be possible without changes to the core of the algorithm. While different ray types are supported by our method and we present techniques for enforcing more coherence, it is still true that the method becomes less efficient for fully incoherent ray patterns. We focus on the overall method and its capability of handling large scenes using single-bounce illumination. For simple multi-bounce illumination, e.g., Whitted-style ray tracing, there is enough coherence even for secondary bounces, and our method can use a separate hierarchy cut for each such bounce. For a full path-tracing solution, our plan is to have a sorting step after each rendering pass that would reorder rays in a more coherent order using a space-filling curve based on scene hit points, along the lines of Moon et al. [103]. Other authors have already done this sorting on the GPU [122], so we are confident that real-time performance is possible. This pass would generate a linear order, and our screen-space hierarchy will become a ray-space hierarchy built on reordered rays.

Multiple Bounces. Each bounce (either refractive or reflective) is handled in a separate deferred shading pass. The contribution of a pass is accumulated to the overall illumination. The new ray-termination positions are then used to initialize the ray textures in the next pass. We use a shared query-pair front for all bounces, meaning that we assume certain coherence of the results of one bounce to the other. An alternative would be to maintain a query-pair front for each bounce separately and share it among different frames. This in most cases improves the coherence at the cost of increased memory consumption.

Object-Order Scheduling. By using object-order scheduling, our method is capable of increasing the coherence of data accesses. We can use concepts widely used in the rasterization pipeline, such as binding large high resolution textures when processing a particular geometry, or applying on-the-fly geometry tessellation. This can be achieved while minimizing the memory consumption of the temporary data and the number of state changes required for switching between rendering different primitives. Such behavior is not possible to achieve with fully hierarchical ray traversal methods, which lead to unpredictable accesses to the scene geometry and materials including the methods using ray sorting and packeting.

Fully Dynamic Scenes. Applications like traffic simulation or games perform complex simulations on the CPU and offload rendering to the GPU. Our method inherently supports managing the scene data and the BVH on the CPU side. Only those parts that were modified and are *actually needed* for rendering will be transferred to the GPU.

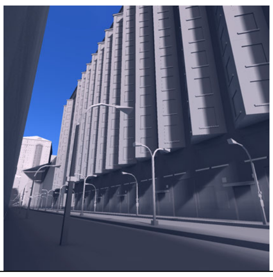
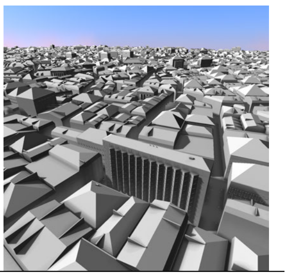
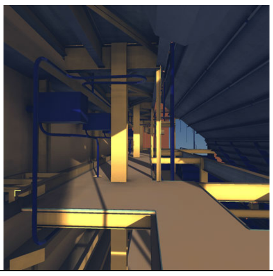
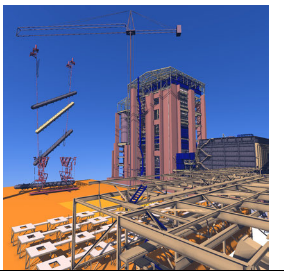
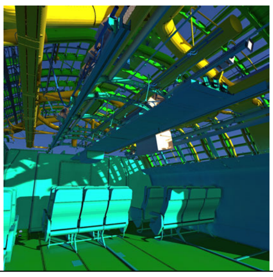
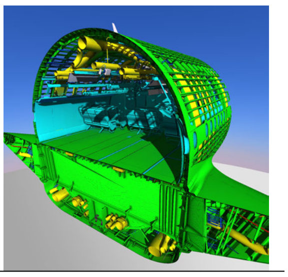
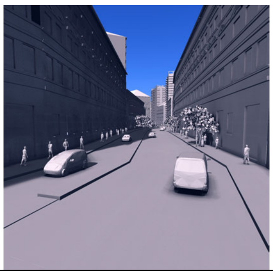
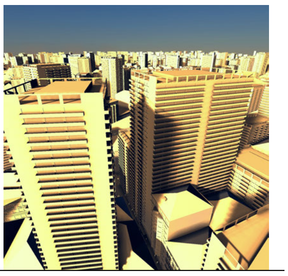
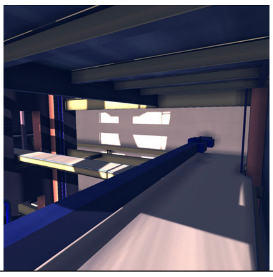
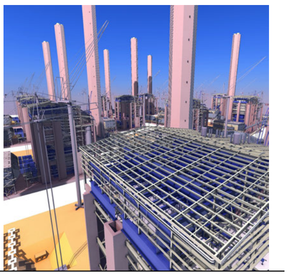
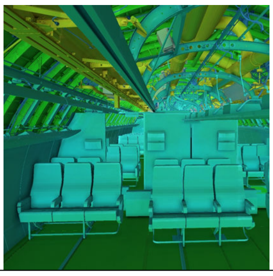
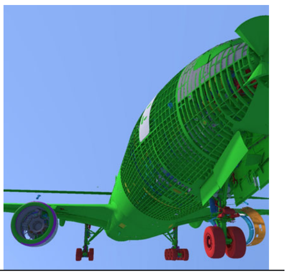
Support for legacy hardware. Some current hardware architectures like game consoles or mobile devices do not allow to implement stack based ray traversal on their GPU due to limited shader capabilities of these devices. When using a naive intersection shader in the leaves of the query-pair hierarchy our method can be used for implementing GPU ray tracing even on such devices. We consider the actual porting of our method to this type of hardware as an interesting practical topic to be addressed in future work. While this opens the possibility of application on architectures with limited shader capabilities such as game consoles, it hinders the ray tracing performance on state of the art GPUs. It should be noted that a leaf-level acceleration structure can be introduced to speed-up intersection queries without any modification to the global architecture.

4.7 Results

For our experiments we use an Intel i7-3770 CPU with 3.5GHz (using one core), 8 GB RAM, a resolution of 1920×1080 and an NVidia Titan GPU with 6 GB of video memory. However, as there seems to be a limitation at 4 GB for use with a single thread, we are only able to allocate a maximum of 3.7 GB for the data (BVH, geometry, materials) of our out-of-core scenes.

Table 4.1 shows the models used in our experiments. We use 3 scenes for in-core ray tracing and 3 out-of-core scenes. City-10 and City-200 are a typical $2.5D$ city models, generated with the City Engine [123] in two levels of detail. They offer a high degree of occlusion for near views and a high degree of regularity. The Powerplant model is considerably less regular, and the created BVH is deep. For out-of-core ray tracing, we use 16 copies of the Powerplant. The complex 777 model is a standard scene for testing out-of-core methods. We also extracted a section of the 777 model for in-core use. Note that the 777 is a challenging scene for our method because many small complex details are visible most of the time. For shading we use the diffuse materials defined in the original scenes if available and a Preetham Skylight model [124] for the City models. In order to keep the memory footprint as low as possible for out-of-core rendering, we generate the face normals on the fly in the intersection shader.

We use a deferred shading approach to compute the shading contribution of each ray in in a post-processing step. Tables 4.2 and 4.3 shows numerical results for our benchmarks using the described technique. In each scene we provide 2 walkthroughs roughly corresponding to a sequence of near-view points (e.g., on street level) and far-view points (e.g., bird's-eye view). We test the proposed method for primary rays, 20 short ambient-occlusion rays (0.5 units long) which sample the hemisphere, and 20 diffuse reflection rays where the maximum ray length is set to cover the full scene extent. These ray types cover many cases typically encountered in ray-tracing applications. Our method uses all optimizations described in Section 4.4.5 in order to

	Near view	Far view
City-10 Num. Triangles: 11.7M Geometry size: 537MB BVH Size: 140MB In-core: 100%		
Powerplant Num. Triangles: 12.8M Geometry size: 584MB BVH Size: 126MB In-core: 100%		
Boeing 777 section Num Triangles: 21.5M Geometry size: 984MB BVH Size: 244MB In-core: 100%		
City-200 Num. Triangles: 139M Geometry size: 6.35GB BVH Size: 1.54GB In-core: 47%		
Powerplant x16 Num. Triangles: 205M Geometry size: 9.34GB BVH Size: 2.02GB In-core: 32%		
Boeing 777 Num. Triangles: 350M Geometry size: 16.1GB BVH Size: 2.80GB In-core: 20%		

Tab. 4.1.: Used models showing near-view and far-view.

Scene	Ray type	Near-View (ms) Primary		AO		Diffuse	
City-10	GLSL	18.22	(1.00×)	186	(1.00×)	493	(1.00×)
	CHC+RT	**11.3**	**(1.61×)**	**153**	**(1.22×)**	**384**	**(1.28×)**
	CUDA	12.5	(1.46×)	311	(0.60×)	615	(0.80×)
Powerplant	GLSL	69.9	(1.00×)	772	(1.00×)	10064	(1.00×)
	CHC+RT	12.8	(5.46×)	**173**	**(4.46×)**	1700	(5.92×)
	CUDA	**11.8**	**(5.92×)**	310	(2.49×)	**1152**	**(8.74×)**
Boeing 777-Section	GLSL	78.1	(1.00×)	516	(1.00×)	5464	(1.00×)
	CHC+RT	24.2	(3.23×)	**236**	**(2.19×)**	2729	(2.00×)
	CUDA	**12.5**	**(6.25×)**	277	(1.86×)	**1264**	**(4.32×)**
City-200	CHC+RT	**25.9**	(-)	**325.7**	(-)	**2720**	(-)
Powerplant ×16	CHC+RT	**18.7**	(-)	**232**	(-)	**1855**	(-)
Boeing 777	CHC+RT	**134**	(-)	**5175**	(-)	**30134**	(-)

Tab. 4.2.: **Comparison for near of our method (CHC+RT) with shader-based ray tracing (GLSL) and CUDA-based raytracing [92] (CUDA) using a resolution of 1080p.** We trace either primary rays or 20 samples per pixel of secondary rays. The numbers in bold identify the best method in terms of the overall frame time. The numbers in parenthesis show the speedup with respect to GLSL.

Scene	Ray type	Far-View (ms) Primary		AO		Diffuse	
City-10	GLSL	26.3	(1.00×)	**277**	**(1.00×)**	586	(1.00×)
	CHC+RT	23.4	(1.12×)	286	(0.97×)	655	(0.89×)
	CUDA	**14.9**	(1.77×)	278	(1.00×)	**422**	**(1.39×)**
Powerplant	GLSL	82.5	(1.00×)	588	(1.00×)	2722	(1.00×)
	CHC+RT	15.0	(5.50×)	**154**	**(3.82×)**	843	(3.23×)
	CUDA	**10.6**	**(7.78×)**	246	(2.39×)	**501**	**(5.43×)**
Boeing 777-Section	GLSL	108	(1.00×)	781	(1.00×)	3752	(1.00×)
	CHC+RT	29.8	(3.62×)	**270**	**(2.89×)**	2382	(1.58×)
	CUDA	**16.4**	**(6.59×)**	316	(2.47×)	**809**	**(4.64×)**
City-200	CHC+RT	**87.8**	(-)	**766.7**	(-)	**2492**	(-)
Powerplant ×16	CHC+RT	**270**	(-)	**1559**	(-)	**25975**	(-)
Boeing 777	CHC+RT	**333**	(-)	**1961**	(-)	**16441**	(-)

Tab. 4.3.: **Comparison for far views of our method (CHC+RT) with shader-based ray tracing (GLSL) and CUDA-based ray tracing [92] (CUDA) using a resolution of 1080p.** We trace either primary rays or 20 samples per pixel of secondary rays. The numbers in bold identify the best method in terms of the overall frame time. The numbers in parenthesis show the speedup with respect to GLSL.

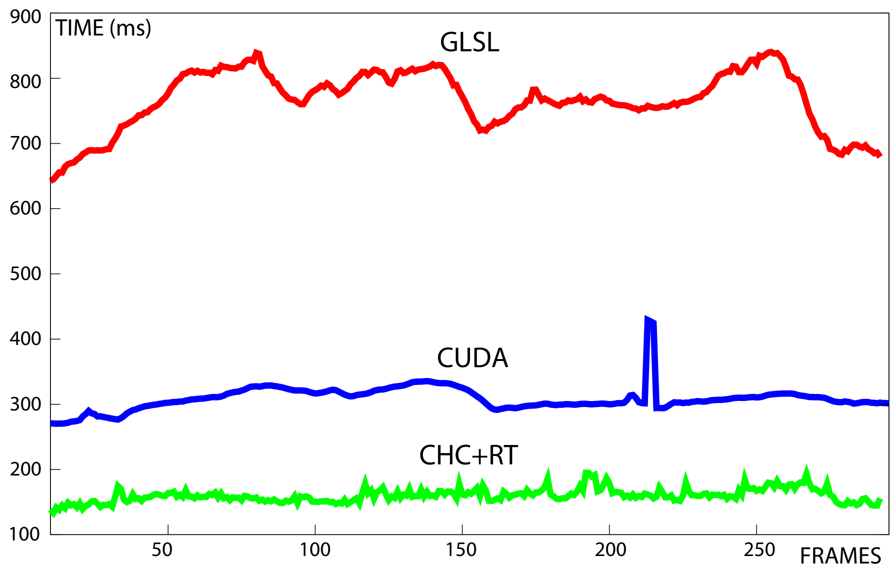


Fig. 4.10.: **Comparison of CHC+RT with GLSL and CUDA** This comparison was recorded during a walk-through, near-view over the Powerplant model for ambient-occlusion rays.

fully exploit temporal and spatial coherence. The BVH in our tests is constructed on the CPU using SAH and optimized using the insertion-based BVH optimization [125]. As termination criteria for the traversal in CHC+RT, we set the maximum subtree height to 24 and the maximum number of triangles to 1M.

We compare our method against standalone GLSL shading-based traversal without occlusion culling (simply traversing the BVH from the root node) and the state-of-the-art CUDA ray tracer of Aila et al. [92]. Note that we disabled ray sorting in CUDA tracing since the overhead significantly outperformed the gain in traversal time, while for our method the rays are already generated in a more coherent fashion.

CHC+RT usually works best for occluded walkthroughs (near views in City scenes and Powerplant). For the City near view, it is faster than both GLSL and CUDA tracing for all ray types. In the Powerplant model, CHC+RT is significantly faster than GLSL and mostly comparable with CUDA. This can also be observed in the frame-by-frame comparison shown in the plot of Figure 4.10. The sources of the speedup with respect to GLSL are that occlusion can be efficiently exploited in Powerplant for both near and far views, and that the deep BVH in Powerplant can be better handled by our method.

In less occluded views, the overhead due to the occlusion queries can sometimes outweigh the benefit for CHC+RT (e.g., City far view). The 777 model is a challenging case for any rendering algorithm, and the 777 Section exhibits similar properties. Since parts of the hull have been removed, many complex details are visible most of the time. This is a good case for the CUDA ray tracer, which is indeed the best method for primary and diffuse rays. On the other hand, CHC+RT shows better overall frame times for ambient-occlusion rays due to the smaller setup time. Also note that CHC+RT is able to reduce the performance gap to CUDA in this scene by a large margin.

The CUDA ray tracer is generally faster in terms of pure traversal times than both GLSL and CHC+RT. But since a higher constant cost is involved in the setup of each frame for CUDA, GLSL is competitive for scene configurations where the ray traversal time is short (e.g., for AO rays and highly occluded scenes). We made the observation that GLSL is much more sensitive to the stack size than CUDA, and this becomes a bottleneck for deep hierarchies. On the other hand, CHC+RT does not suffer from this problem. Indeed, the scheduled subtrees have a bounded traversal height and hence the stack size can be bounded.

The method scales well to large, possibly out-of-core scenes if sufficient occlusion is available. The performance of the near view in City-200 is comparable to the near view in City-10 in spite of the over 12$\times$ larger scene and the out-of-core overhead, and similar to the performance in the 777 Section. The same is true for the near views in Powerplant and Powerplant$\times$16. As can be observed for the far view of Powerplant$\times$16 and for 777, diffuse rays in open view-points in the large out-of-core scenes are quite challenging for our method, but can be improved using the aggressive version of our algorithm discussed below.

Figure 4.11 visualizes the timings of the different phases of the algorithm as listed in Algorithm 4.7. Phase 1 corresponds to the intersection of previously visible nodes, while Phase 2 and 3 correspond to the overhead caused by occlusion culling. Phase 2 evaluates the current visibility status using queries for previously visible and invisible nodes. Phase 3 traverses the hierarchy in response to a change in visibility. Interestingly, the time spent in Phase 3 relative to the other phases increases for the out-of-core scenes. The reason is that the overhead of Phase 3 corresponds to changes in visibility. Even if the currently visible scene fits completely in-core and there are no node fetches during Phase 1, nodes that become newly visible will be uploaded to the GPU in this phase. Table 4.4 shows some interesting statistics for the out-of-core models. For all models, the primary ray rendering can be done

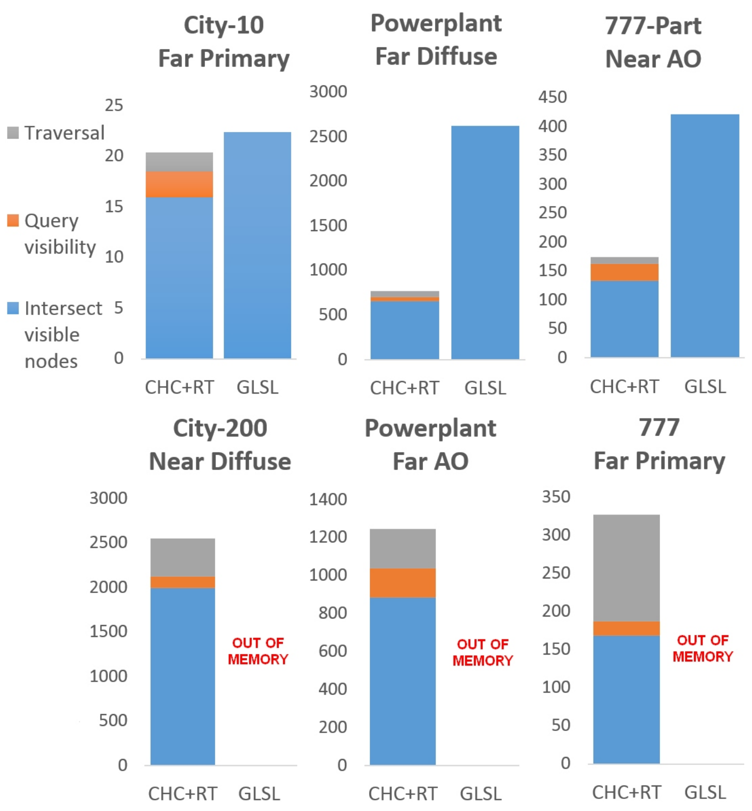


Fig. 4.11.: **Traversal-time comparison** of CHC+RT with GLSL split into the different phases of the algorithm.

Scene Ray type		Near-View Prim	 AO	 Diff	Far-View Prim	 AO	 Diff
City-200	Mrays/sec	80.1	127	15.2	23.6	54.1	16.6
	Queries	822	13.6K	61.2K	2.46K	42.9K	117K
	Trans. MB	0	5.85	383	0.61	0.11	1.73
	BVH	0	3.00	94.9	1.96	0.30	2.04
	Geom	0	2.85	288	2.58	0.41	3.77
PP×16	Mrays/sec	111	179	22.4	7.68	26.6	1.60
	Queries	793	13.2K	32.3K	3.63K	61.7K	312K
	Trans. MB	0.11	0.03	0.55	170	43.6	2310
	BVH	0.04	0.01	0.13	85.8	34.7	663
	Geom	0.07	0.02	0.42	83.8	8.85	1647
777	Mrays/sec	15.4	8.01	1.38	6.23	21.1	2.52
	Queries	1.76K	123K	501K	6.31	85K	278K
	Trans. MB	11.7	419	1442	355	96.7	1073
	BVH	2.64	139	595	70.1	25.0	253
	Geom	9.14	281	847	285	71.7	820

Tab. 4.4.: Per-frame statistics for the out-of-core models.

predominantly in-core. Ambient occlusion and diffuse interreflections in particular require significantly larger transfer rates between CPU and GPU memory.

		City-200		Powerplant $\times 16$		Boeing 777	
Pixels	%Tile	Near	Far	Near	Far	Near	Far
0	0.000	2720	2492	1855	25975	30134	16441
20	0.005	780	2019	1682	9651	11544	4807
100	0.025	738	1751	1553	4727	8081	2712
200	0.050	715	1611	1489	3578	6804	2028

Tab. 4.5.: **Timings for the aggressive version of CHC+RT,** using 20 diffuse reflection rays. %Tile shows the error in % of the termination size of the screen-space tiles. Visually the images look similar and the speedup is significant ($x3.2$).The mean absolute error for the pixel colors is 9.04.

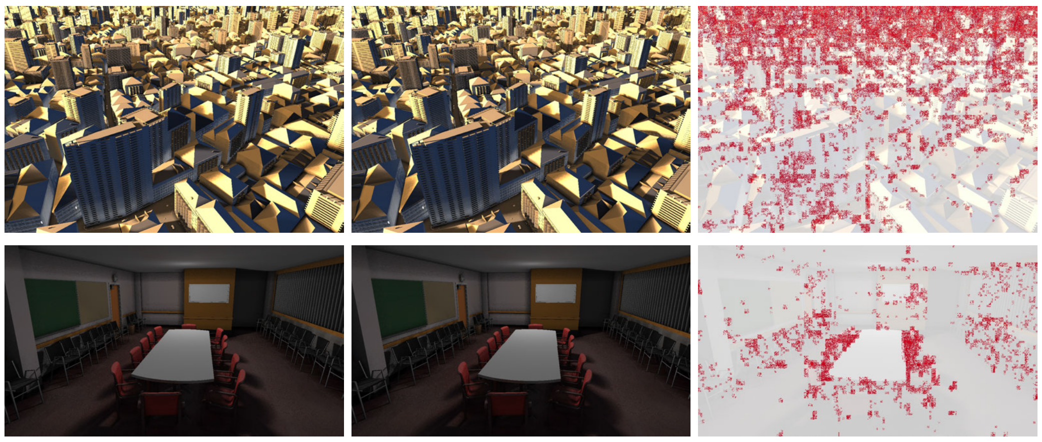


Fig. 4.12.: Comparison of the conservative (left) with the aggressive version of our method (middle) allowing 20 pixels of error for diffuse rays. Right: Pixel differences are mostly in the background.

The proposed algorithm is conservative because the occlusion-query result (the number of visible pixels) is used for a binary decision. As an alternative, this number can be used for a simple LOD mechanism that culls all nodes whose contribution to a screen-space tile is less than a visible pixel threshold. As can be seen in Table 4.5, the aggressive algorithm is especially useful for reducing the computational complexity of diffuse reflections, where many nodes contribute to only a few pixels. As shown in Figure 4.12, allowing for example an error of 20 pixels per query can reduce the render time by a factor of 3 with only a minor decrease in accuracy, with a mean absolute pixel error of 9.04. A detailed plot of one of the walkthrougs can be seen in Figure 4.10 in terms of million rays per second and number of occlusion queries issued in each frame.

Table 4.6 shows the influence of spatial coherence during ray generation on the performance of our method. This is achieved by increasing the value for the maximum angle for the random kernel rotation per-pixel. Diffuse reflections slow down by

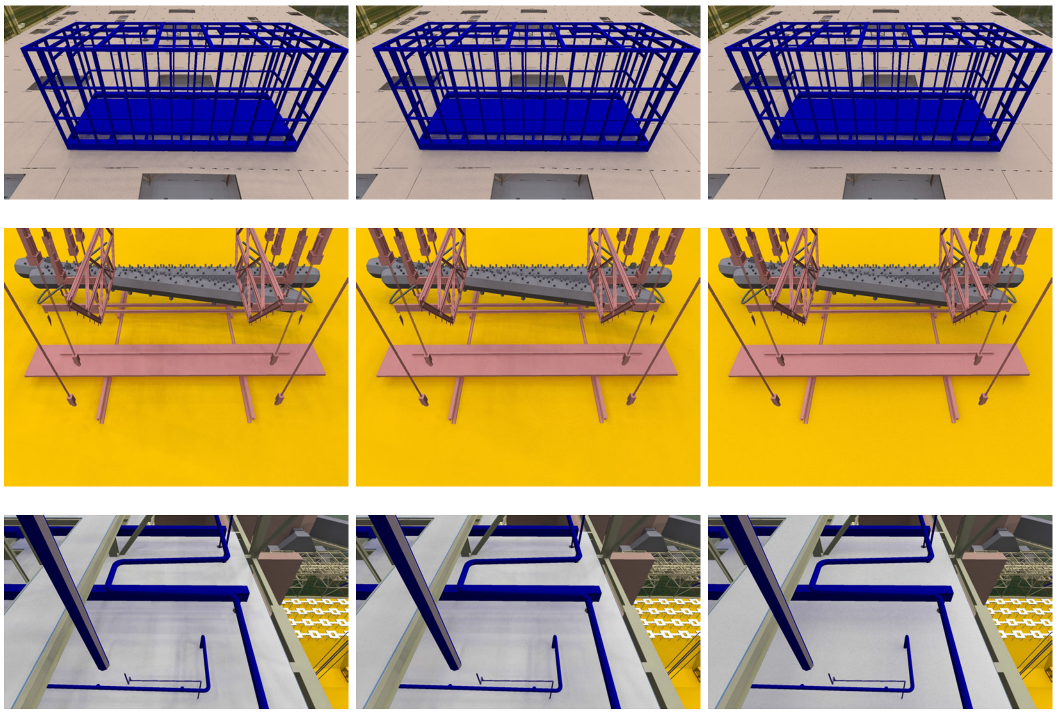


Fig. 4.13.: **Examples of the effects of the random rotation** on diffuse color bleeding using 20 samples for $18°$ (left), $36°$ (middle), and full randomization (right) (zoom in to see the differences).

		Randomization			
		None	18°	36°	Full
AO	City-10 Near (ms)	136	145	156	193
	Powerplant Near (ms)	153	167	175	190
Diffuse	City-10 Near (ms)	228	323	384	805
	Powerplant Near (ms)	664	1334	1700	4125

Tab. 4.6.: **Effect of the per-pixel random rotation of the diffuse sampling kernel** on the coherence and frame time of 20 diffuse reflection rays in two selected models.

a factor of over 4–$5\times$ when going from no randomization to a fully randomized rotation, whereas the frame times for AO rays are affected much less. Note that the frame times using full randomization are still comparable to GLSL. The *temporal coherence* can be maintained by storing the visibility status for each ray direction. In our results we use a per-pixel rotation of $36°$ for 20 samples, which provides good quality and maintains a sufficient degree of spatial coherence (as shown in Figure 4.13). It also has the benefit to eliminate some temporal noise in moving frames.

4.8 Discussion

In this chapter I presented a novel use of hierarchical occlusion culling for accelerating OpenGL-based ray tracing. Our method exploits the rasterization pipeline and hardware occlusion queries in order to create coherent batches of work for the GPU ray-tracing kernel. By generalizing occlusion culling to arbitrary rays through a combined hierarchy in both ray space and object space, we are able to share the intermediate traversal results among multiple rays, leading to a simple and efficient implicit parallelization using rasterization hardware. Through novel means for scheduling GLSL ray tracing kernels using the coarse-grained hierarchy over screen- and object-space, we are able to support rendering of out-of-core ray tracing using GPU memory as a cache. Our method narrows the gap between OpenGL-based ray tracing and CUDA ray tracing by a significant amount and is able to outperform CUDA ray tracing in some cases.

The current implementation is based on OpenGL, and, given the evolution of the graphics APIs, it would be interesting to evaluate how the same concepts can be efficiently implemented on other platforms, such as WebGL and Vulkan.

While the method fully exploits the hierarchy in object space, we could extend it to better exploit the image-space hierarchy and thus to improve the algorithmic efficiency. We implicitly assume that issuing occlusion queries is the dominant cost. The cost of the queries depends, however, on the size of the active ray set (screen-space rectangle area), since each ray is processed individually. An interesting future work concerns devising methods for efficiently sorting and grouping rays to compute queries in constant time.

Currently only single bounce rendering of diffuse reflections is supported, thus, one of the most interesting evolutions concerns the extension to multiple bounces of arbitrary rays in a complete path tracing application.

4.9 Bibliographical notes

Most of the content of this chapter was presented in our Eurographics 2015 contribution and published in the *Computer Graphics Forum* journal [5], describing the novel CHC+RT algorithm, its implementation and results.

A number of recent relevant publications have presented follow-ups of this work, extending it in different directions. For instance, Wald et al. [126] use an adaptation of the kd-tree for raytracing large amounts of particles from molecular dynamic simulations, astrophysics, etc. Kostas et al. [127] presented in 2016 the *DIRT* system for computing interactive image-space ray tracing, with the aim of improving our ray coherence model. In 2017 Barringer et al. [128] presented *Ray Accelerator*, an heterogeneous raytracing system which similarly to our approach subdivides the rays in large packages and schedules their computation. They manage to distribute the work between GPU (for visibility) and CPU (mostly for shading), using shared memory for communication. Recently Kol et al. [129] have presented *MegaViews*, a scalable architecture to render complex scenes from many viewpoints, with applications to real-time computation of global illumination solutions and complex shadowing. They use also a double scene-view hierarchy, updating every frame a queue of node-pairs to compute. Also, very recently, Hynt et al. from Oculus Research [130] have proposed the use of our approach as a future improvement of their Hierarchical Visibility for Virtual Reality (HVVR) method.

5

Improving scalability through compression: Symmetry-aware Sparse Voxel DAG

In the previous chapter, I proposed a method to improve rendering performance by subdividing work and data into small batches, dynamically selecting at runtime, based on visibility considerations, what data to load and what computation to perform. Such a method makes it possible to render very large scenes, exceeding GPU memory, using adaptive loading. In this chapter, instead, I will explore a fully orthogonal solution, which tackles the problem of massive model rendering by aggressively compress data so that it fits, in fully renderable format, on GPU memory. The presented method is targeted to improve performance on voxelized representations of complex 3D scenes, which are widely used to accelerate visibility queries in many GPU rendering techniques. Since GPU memory is limited, it is important that these data structures can be kept within a strict memory budget. Recently, directed acyclic graphs (DAGs) have been successfully introduced to compress sparse voxel octrees (SVOs), but they are limited to sharing identical regions of space. In this chapter, we show that a more efficient lossless compression of geometry can be achieved while keeping the same visibility-query performance. This is accomplished by merging subtrees that are identical through a similarity transform and by exploiting the skewed distribution of references to shared nodes to store child pointers using a variable bit-rate encoding. We also describe how, by selecting plane reflections along the main grid directions as symmetry transforms, we can construct highly compressed GPU-friendly structures using a fully out-of-core method. Our results demonstrate that state-of-the-art compression and real-time tracing performance can be achieved on high resolution voxelized representations of real-world scenes of very different characteristics, including large CAD models, 3D scans, and typical gaming models, leading, for instance, to real-time GPU in-core visualization with shading and shadows of the full Boeing 777 at sub-millimeter precision.

WITH the increase in performance and programmability of graphical processing units (GPUs), GPU raycasting is emerging as an efficient solution for many real-time rendering problems. In order to handle large detailed scenes, devising compact and efficient scene representation for accelerating ray-geometry intersection queries becomes paramount, and many solutions have been proposed (see Sec. 5.2). Among these, sparse voxel octrees (SVO) [131] have provided impressive results, since they can be created from a variety of scene representation, they efficiently carve out empty space, with benefits on ray tracing performance and memory needs, and they implicitly provide a levels-of-detail (LOD) mechanism. Given their still relatively high memory cost, and the associated high memory bandwidth required, these voxelized approaches have, however, been limited to moderate scene sizes and resolutions, or to effects that do not require precise geometric details (e.g., soft shadows). While many extremely compact representations for high-resolution volumetric models have been proposed, especially in the area of volume rendering [132], the vast increase in compression rates of these solutions is balanced by increased decompression and traversal costs, which makes them hardly usable in general settings. This has triggered a search for simpler scene representations that can provide compact representations within reasonable memory footprints, while not requiring decompression overhead. Kämpe et al. [14] have recently shown that, for typical video-gaming scenes, a binary voxel grid can be represented orders of magnitude more efficiently than using a SVO by simply merging together identical subtrees, generalizing the sparse voxel tree to a directed acyclic graph (SVDAG). Such a representation is compact, as nodes are allowed to share pointers to identical subtrees, and remains as fast as SVOs and simple octrees, since the tracing routine is essentially unchanged.

In this work, we show that efficient lossless compression of geometry can be combined with good tracing performance by merging subtrees that are identical up to a similarity transform, using different granularity at inner and leaf nodes, and compacting node pointers according to their occurrence frequency. The resulting structure, dubbed *Symmetry-aware Sparse Voxel DAG* (SSVDAG) can be efficiently constructed by a bottom-up external-memory algorithm that reduces an SVO to a minimal SSVDAG by alternating different phases at each level. First, all nodes that represent similar subtrees are clustered and replaced by a single representative. Then, pointers to those nodes in the immediately higher level are replaced by tagged pointers to the single representative, where the tag encodes the transformation that needs to be applied to recover the original subtree from the representative. Finally, representatives are sorted by their reference count, which allows for an efficient variable-bit-rate encoding of pointers. We show that, by selecting planar reflections along the main grid directions as symmetry transform, good building and tracing performance can be achieved.

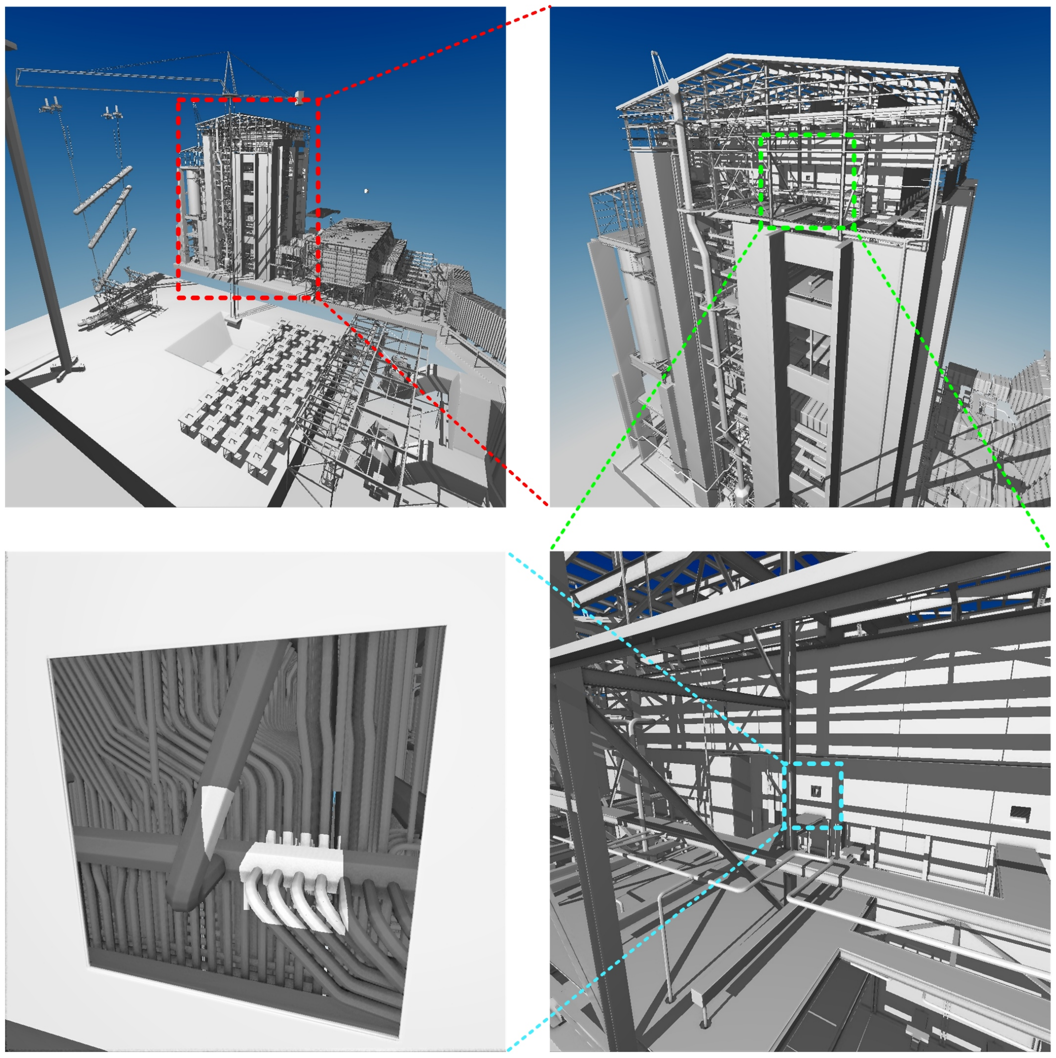


Fig. 5.1.: **Compressed Power Plant model, zoom-in details.** Our algorithm is able to compress and render at full frame-rate this large CAD model, at $1M^3$ voxelization resolution. It has 1.6 billion of full voxels, which compressed with SSVDAGs occupies 3.2GB, fitting in nowadays GPUs memory. Thus, the compression rate is of 0.017 bits/voxel.

5.1 Contribution

The main contribution of this research are:

- A compact representation of a Symmetry-aware Sparse Voxel DAG that can losslessly represent a voxelized geometry of many real-world scenes within a small footprint and can be efficiently traced;
- An out-of-core algorithm to construct such representation from a SVO or a SVDAG; we describe, in particular, a simple multi-pass method based on repeated merging operations;

- A clean modification of standard GPU raycasting algorithm to traverse and render this representation with small overhead. We describe, in particular the details of a GPU tracing method based on a multi-resolution Digital Differential Analyzer (DDA), implemented with a full stack. This sort of approach has been proven effective in previous work on SVOs and SVDAGs [131, 14], and is extended here to handle graphs with reflective transformations.

Our reduction technique is based on the assumption that the original scene representations is geometrically redundant, in the sense that it contains a large amount of subtrees which are similar with respect to a reflective transformation. Our results, see Sec. 5.6, demonstrate that this assumption is valid for real-world scenes of very different characteristics, ranging from large CAD models, to 3D scans, to typical gaming models. This makes it possible to represent very large scenes at high resolution on GPUs, and to support precise geometric rendering and high-frequency phenomena, such as sharp shadows, with a tracing overhead of less than 15%. Similarly to other works on DAG compression [14, 133, 134], we focus in this chapter only on geometry, and not on non-geometric properties of voxels (e.g., material or reflectance properties), which should be handled by other means. A recent example on how to associate attributes to the original SVDAGs has for instance been recently presented by Dado et al. [135].

This technique was presented in joint papers at *i3D* conference [6] and in the (JCGT) journal [7]. SSVDAGs is the main contribution of this thesis, since I co-designed the method and its efficient implementation, fully implemented all the system components, and performed the full evaluation.

5.2 Related Work

Describing geometry for particular applications and devising compressed representation of volumetric models are broad research fields. Providing a full overview of these areas is beyond the scope of this thesis. We concentrate here on methods that employ binary voxel grids to represent geometry to accelerate queries in GPU algorithms. We refer the reader to a recent survey [132] for a more general overview in GPU-friendly compressed representations for volumetric data.

Starting from more general bricked representations proved successful for semitransparent GPU raycasting [136, 137], Laine and Karras [131] have introduced Efficient

Sparse Voxel Octrees (ESVOs) for raytracing primary visibility. In their work, in addition to employing the octree hierarchical structure to carve out empty space, they prune entire subtrees if they determine that they are well represented by a planar proxy called contour. Storing the proxy instead of subtrees achieves considerable compression only in scenes with many planar faces, and introduces stitching problems as in other discontinuous piecewise-planar approximations [138]. Crassin et al. [139] have shown the interest of such approaches for secondary rays, computing ambient occlusion and indirect lighting by cone tracing in a sparse voxel octree. Their bricked structure, however, requires large amounts of memory, also due to data duplication at brick boundaries.

A number of works have thus concentrated on trying to reduce memory consumption of such voxelized structures while maintaining a high tracing performance. Crassin et al. [137] mentioned the possibility of instancing, but rely on ad-hoc authoring for fractal scenes, rather than algorithmic conversions. Compression methods based on merging common subtrees have been originally employed in 2D for the lossless compression of binary cartographic images [140], and extended to 3D by Parker and Udeshi [141] to compress voxel data. These algorithms, however, are costly and require fully incore representations of voxel grids. Moreover, since voxel content is not separated from voxel attributes, only moderate compression is achieved. Recently, Dado et al. [135] presented a compressed structure able to encode-decode voxels attributes, such as color or normals in another auxiliary structure. This work is orthogonal to ours. Hoetzlein [142] created GVDB, a hierarchy of grids of volumetric models with dynamic topology, with the scope of fast decoding in GPU. The traversal algorithm is similar to our DDA-based one.

High Resolution Sparse Voxel DAGs (SVDAG) [14] generalize the trees used in Sparse Voxel Octrees (SVOs) to DAGs, allowing the sharing of common octrees. They can be constructed using an efficient bottom-up algorithm that reduces an SVO to a minimal SVDAG, which achieves significantly reduced node count even in seemingly irregular scenes. The effectiveness of the method is demonstrated by ray-tracing high-quality secondary-ray effects using GPU raycasting from GPU-resident SVDAGs. This approach has later been extended to shadowing by voxelizing shadow volumes instead of object geometry [133, 134], as well as for time-varying data [143]. We improve over SVDAGs by merging subtrees that are identical up to a similarity transform, and present an efficient encoding and building algorithm, with an implementation using reflective transformations. The idea of using self-similarity for compression has also found application in point cloud compression [144], where, however, the focus was on generating approximate representations instead of lossless ones.

In addition to reducing the number of nodes, compression can be achieved by reducing node size. As pointers are very costly in hierarchical structures, a number of proposals have thus focused on reducing their overhead. While pointerless structures based on exploiting predefined node orderings have been proposed for offline storage [145], they do not support efficient run-time traversal. The optimizations used for trees, such as grouping children in pages and using relative indexing within pages [131, 146] are not applicable to our DAGs, since children are scattered throughout the structure due to sharing. By taking advantage of the fact that the reference count distribution of shared nodes is highly skewed, we thus employ a simple variable bit-rate encoding of pointers. A similar approach has been used by Dado et al. [135] for compression of pointers in their auxiliary structures for storing voxel attributes. That system has been independently developed in parallel to ours.

5.3 Overview

A 3D binary volumetric scene is a discretized space subdivided in N^3 cells called voxels, which can be empty or full. Since this structure grows cubically for every subdivision, is is hard to achieve high resolutions. SVOs compactify these representation using a hierarchical octree structures of nodes arranged in a number of levels (L), with $N = 2^L$, and most commonly represented using a children bitmask per node as well as up to eight pointers to nodes in the next level. When one of those children represents an empty area, no more nodes are stored under it, introducing sparsity and thus efficiently encoding whole empty areas of the scenes. The structure can be efficiently traversed on the GPU using stackless or short-stack algorithms [131, 147], which exploit sparsity for efficient empty-space skipping.

SVOs and grids can be directly created from a surface representation of the scene through a *voxelization* process, for which many optimized solutions have been presented (see, e.g., [148]). In this work, we use a straightforward CPU algorithm that builds SVOs using a streaming pass over a triangle soup, inserting triangles in an adaptive octree maintained out-of-core using memory-mapped arrays. Using other more optimized solutions would be straightforward.

SVDAGs optimize SVOs by transforming the tree to a DAG, using an efficient bottom-up process that iteratively merges identical nodes one level at a time and then updates the pointers of the level above. The resulting structure is more compact than SVOs, and can be traversed using the exact same ray-casting algorithm, since node sharing is transparent to the traversal code.

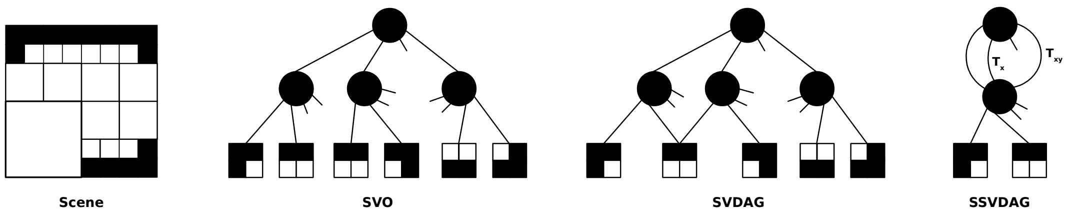


Fig. 5.2.: Example 2D scene transformed into different structure, with children are ordered left-right, top-bottom. The Sparse Voxel Octree (SVO) contains 10 nodes. The Sparse Voxel Directed Acyclic Graph (SVDAG) finds one match and then shares a node, meaning 9 nodes. The presented Symmetry-aware Sparse Voxel Directed Acyclic Graph (SSVDAG) finds reflective matches in two last levels, and reduces the structure to 4 nodes.

The aim of this work is to obtain a more compact representation of the volume, while keeping the efficiency in traversal and rendering. We do this by merging self-similar subtrees (starting from an SVDAG or an SVO), and by reducing node size through an adaptive encoding of children references.

Among the many possible similarity transformations, we have selected to look for *reflective symmetries*, i.e. mirror transformations along the main grid planes. We thus consider two subtrees similar (and therefore merge them) if their content is identical when transformed by any combination of reflections along the principal planes passing through the node center. Such a transformation $T_{x,y,z}$ has the advantage that the 8 possible reflections can be encoded using only 3 bits (reflection X,Y,Z), that the transformation ordering is not important, as transformations along one axis are independent from the others, and that efficient access to reflected subtrees, which requires application of the direct transformation $T_{x,y,z}$ or of its inverse $T^{-1}_{x,y,z} = T_{x,y,z}$, can be achieved by simple coordinate (or index) reflection. This leads to efficient construction (see Sec. 5.4.1) and traversal (see Sec. 5.5). In addition, since the transformation has a geometric meaning, the expectation, verified in practice, is to frequently find mirrored content in real-world scenes (see a 2D example in Fig. 5.2). The output of the merging process is a DAG in which non-empty nodes are referenced by tagged pointers that encode the transformation $T_{x,y,z}$ that needs to be applied together with the child index. Further compression is achieved by taking advantage of the observation that not all subtrees are uniformly shared, i.e., some subtrees are significantly referenced more than others. We thus use a variable bit-rate encoding, in which the most commonly shared subtrees are references with small indexes, while less common subtrees are referenced with more bits. This is achieved through a per-level node reordering process, followed by a replacement of child pointers by indices. The encoding process, as well as the resulting final encoding is described in Sec. 5.4.3.

5.4 Construction and encoding

A SSVDAG is constructed bottom-up starting from a voxelized representation (SVDAG or SVO). We first explain how a minimal SSVDAG is constructed by merging similar subtrees, and then explain how the resulting representation is compactly encoded in a GPU-friendly structure. While the original work [6] described a fully out-of-core implementation based on external-memory arrays, we propose in this version a simpler approach that proceeds by repeated in-core reductions of subtrees.

5.4.1 Bottom-up construction process

Constructing the SSVDAG requires to efficiently find reflectively-similar subtrees. Since explicitly checking similarity in subtrees would be prohibitively costly for large datasets, we use a bottom-up process that iteratively merges similar nodes one level at a time. This requires, however, some important modifications to the original SVDAG construction method. In our technique, we use arrays encoding level-by-level the existing nodes, with one array per level. For construction, each array element contains an uncompressed node description containing for leaf nodes a bitmask while for inner nodes 8 (possibly null) child pointers and a bitmask to take into account invariance with respect to transformation during inner nodes clustering (see below). We start the construction process from the finest level $L-1$, and proceed up to the root at level 0.

Our construction code is capable to perform a transformation into a DAG with or without symmetries, and works, for compatibility with previous encoding methods, using a leaf size of 2^3. Grouping into larger leaves is performed in post-processing during our encoding phase (see Sec. 5.4.3). At each level, we first group the nodes into clusters of self-similar nodes, then select one single representative per cluster and associate to others the transformation that maps them to the representative. The surviving nodes are reordered for compact encoding (see Sec. 5.4.3) and stored in the final format. Child pointers of nodes at the previous level are then updated to point to the representatives, and the process is repeated for all levels up to the root.

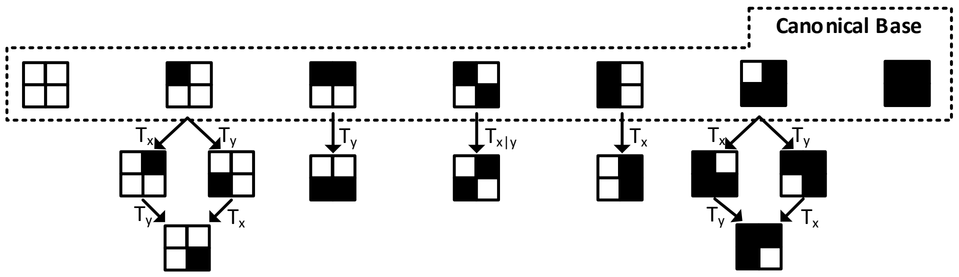


Fig. 5.3.: **2D canonical base of transformations.** Example of all the 2D canonical symmetry transformations of a small voxel grid into a set of base representatives. The transformation maps clusters of self-similar grids to a unique representative.

The matching process at the core of clustering is based on the concept of reordering the nodes at a given level so that matching candidates are stored nearby. Clustering and representative selection is then performed during a streaming pass. Leaf nodes and inner nodes, must use, however, different methods to compute ordering and perform matching.

Leaf nodes clustering In order to efficiently match leaf nodes in the tree, we must discover which representation of small voxel grids remain the same when one of the possible transformations is applied. Considering that each grid G can be represented by a binary number $B(G)$ by concatenating all the voxel occupancy bits in linear order, we define a mapping of each possible grid G to another grid $G^\star = T^\star_{x,y,z}(G)$, such that the *canonical transformation* $T^\star_{x,y,z} = \arg\max_{T_{x,y,z}} B(T_{x,y,z}(G))$. $G^\star$ is the *canonical representation* of G, and represents, among all possible reflections of G, the one with the largest integer value. Geometrically, it is the one that attracts most of the empty space to the origin (see Fig. 5.3). This transformation is precomputed in a table of 256 entries that maps all the possible combinations of 2^3 voxels to the bitcode representing the canonical transformation as well as to the unique canonical representation (one of the 46 possible ones). Given this transformation, two nodes are self-similar if their canonical representation is the same. Clustering can thus be performed in a single streaming pass after sorting leaves using the canonical representation as a key. Nearby nodes sharing the same canonical representation are merged into a single representative, pointers at the upper level are then updated to point to the representative, and pointer tags are computed so as to obtain the original leaf from the representative.

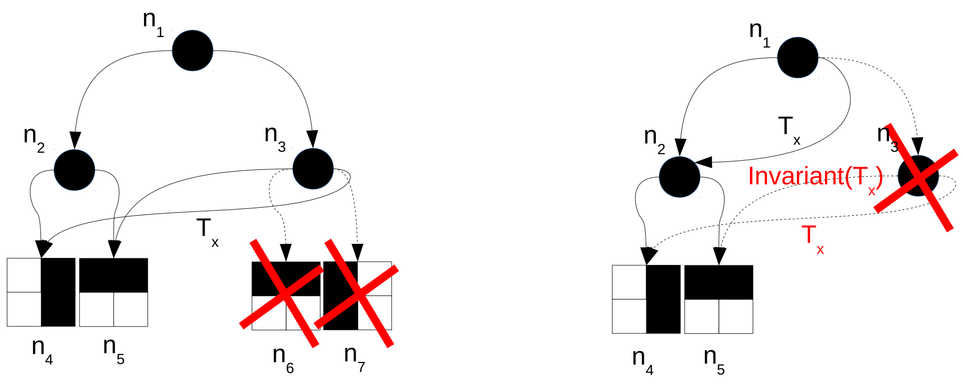


Fig. 5.4.: Clustering of nodes and invariants check. On the left, during leaf clustering, references to leaf node n_6 are replaced by references to n_5, which is identical, while references to n_7 are replaced with references to n_4 transformed by transformation T_x. On the right, inner node n_3 is replaced with n_2 through transformation T_x since its left child n_5 is invariant to transformation T_x and is identical to the right child of n_2, while its right child matches the left child of n_2 through the same transformation T_x.

Inner nodes clustering. While for leaf nodes we can detect symmetries by directly looking at their bit representation, two inner nodes n_1 and n_2 must be merged if they represent the exact same *subtrees* when a transformation T is applied. The first trivial condition to be checked is that child pointers must be the same. We thus sort the inner nodes using the lexicographically sorted set of pointers to children as key. Since after sorting all self-similar nodes are positioned nearby, as they are among those that share the same set of pointers, we perform merging by creating, during a streaming pass, one representative per group of self-similar nodes. The self-similarity condition must be verified without performing a full subtree comparison. Given the properties of our reflective transformations, we have thus to verify that, when the two nodes n_1 and n_2 are matched for similarity under a candidate transformation T, every tagged pointer (tag, p) of n_1 is mapped to $(T(tag), p)$ in n_2 if p is not invariant to the transformation T, or it is mapped to $(T(tag) \bigvee \neg T(tag), p)$ if it is invariant (see Fig. 5.4). This means, for instance, that, when looking for a match under a left-right transformation T_x, the left and right pointers must be swapped in n_2 with respect to n_1, and the pointed subtrees must be equal under a left-right mirroring, The latter condition is verified if the pointed subtree has a left-right symmetry, or if, for each matched pair of tagged child pointers, the left-right transformation bit is inverted while the other bits are the same. This process does the clustering for one particular transformation T, and is repeated for each of the 8 possible reflection combinations, stopping at the first transformation that generates a match with one of the currently selected representatives, or creating a new representative if all tests are unsuccessful. In order to efficiently implement invariance checks, we thus associate three invariant bits (one for each mirroring direction) at each of the leaves when computing their canonical representations, and pull them up during construction at inner nodes by suitably combining the invariant bits of pointed nodes at each merging step. For instance, an inner node is considered invariant with respect to a left-right transformation if all its children are invariant with respect to that transformation, or the left children are the mirror of the right ones.

5.4.2 Out-of-core implementation

The construction process described above constructs an SSVDAG starting from a voxelized representation (SVO or SVDAG), and must be made scalable to massive models.

Our original work[6] used a direct implementation of the described method, which performed the reduction in a single pass, using external memory structures to store, by level, all the required data. These structures were per-level memory-mapped arrays storing the voxel tree. Scalability was thus achieved by relying on operating system features to handle virtual memory. This required, however large (out-of-core) temporary storage space to store the fully constructed tree.

In this work. we employ, instead, a simpler solution based on building and merging bottom-up portions of the dataset fully fitting in core memory.

It should be noted that, unlike what it has been seen before in this thesis, the goal of the method shown in this chapter, the SSVDAG reduction, is to produce, even from very massive inputs, an end results that needs to fit within GPU memory constraints. A recursive merging solution, which reduces voxelized representations bottom-up until they fit into memory thanks to partial reduction is therefore applicable. Even if such a construction approach assumes that the final reduction can be performed fully in-core, this is a safe assumption in our context, since the final reduced SSVDAG produces a data structure that should be stored in GPU memory, which is typically much smaller than available RAM.

When starting from a model whose representation exceeds the available memory for construction, we therefore construct an SSVDAG of depth L using the following process:

1. we estimate the level $L_0 < L$ of the octree structure at which all subtrees are deemed small enough to fit into memory once transformed into SSVDAGs;

2. for each of the voxels at level L_0, we perform a separate and independent reduction process confined to their spatial regions by applying in sequence the following steps:
    a) we create, in-core, a voxelized representation of depth $L - L_0$ of the portion of the model contained within the voxel; if the representation is already available, the relevant voxel data is just loaded from disk, otherwise it is computed on-the-fly through a voxelization process; in the latter case, for a mesh, this can be done by streaming over the input model's triangles, without the need to load the entire model in memory;
    b) we reduce, in-core, the voxelized representation to an SSVDAG using the process described in Sec. 5.4.1, and we store the resulting SSVDAG on disk.
3. in order to compute the final DAG of depth L, we recursively merge bottom-up the reduced representations computed in the previous phase. For each reduction step, this is achieved by
    a) we load all the SSVDAGS associated to the voxels of the relevant subtrees in memory and construct a single DAG by creating octree nodes above them until the common root is reached;
    b) we apply the reduction process described in Sec. 5.4.1 to the union of SSVDAGS instead of an octree, and we store the resulting SSVDAG on disk;
4. we repeat the process bottom up, until we have reduced the root of the original octree to a single SSVDAG; at this point, the stored SSVDAG is the final model.

Separately processing subtrees (or sub-DAGs) during the reduction phases performed at a given octree level will reduce voxel counts by finding symmetries only in the spatial region represented by each subtree, which are a subset of the total ones. It is important to note that this is not a limitation, since the overall global reduction will be obtained when the separately reduced subtrees are merged in subsequent bottom-up reduction phases. This is because each subtree reduction always restarts from the leaf level L of the merged subtrees (or sub-DAGS) and recurses up to their root. We exploit this fact to parallelize the construction process, in order to perform several reductions in paralell. On a machine with *num_procs* processors, we thus give as maximum memory budget for subtree construction the maximum in-core memory divided by *num_procs*, and then perform *num_procs* reductions in parallel.

5.4.3 Compact encoding

The outlined construction process produces a DAG where inner nodes point to children through tagged pointers that reference a child and encode the transformation that has to be applied to recover the original subtree. We encode such a structure in a GPU friendly format aimed at reducing the pointer overhead, while supporting fast tracing without decompression. We achieve this goal through leaf grouping, frequency-based pointers compaction, and memory-aligned encoding.

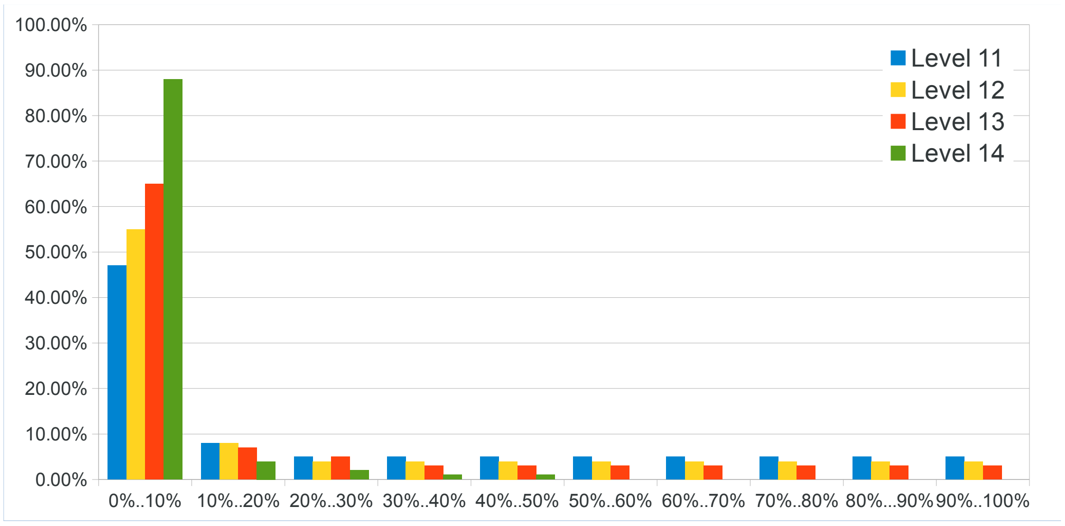


Fig. 5.5.: Histograms of the references to nodes in the Powerplant dataset voxelized at $64K^3$ resolution, with nodes sorted by reference counts. As we can see, the distribution is highly skewed, and the most popular 10% of the nodes account for most of the references.

Leaf grouping. While 2^3 allow for an elegant construction method using table-based clustering, such a level of granularity leads to a high structure overhead, since rays have to traverse deep pointer structures to reach small 8-voxel grids, and the advantage of clustering is offset by the need to encode pointers to these small nodes. For final encoding, we have thus decided to coarsen the construction graph by one level, encoding as simple grids all the 4^3 grids, and to store them in a single array of bricks, each occupying 64 bits. Note that this decision does not require performing new matches on 4^3 leaves, since we just coarsen the graph obtained with the bottom-up process described in Sec. 5.4.1, which uses a table-based matching on 2^3 leaves at level $L-1$ to drive the construction of 4^3 inner nodes at level $L-2$.

Frequency-based pointer compaction. We have verified that in our SSVDAG the distributions of references to nodes is highly skewed. This means that there typically is a small groups of node referenced by a lot of parents nodes, while many others are referenced much less. Fig. 5.5, for instance, shows the histogram of the distribution of reference counts in the Powerplant dataset of Fig. 5.1, where the most common 10% of the nodes is referenced by nearly 90% of pointers at level 14 (nearly 50% at level 11). We have thus adopted an approach in which frequently used pointers are represented with less bits than more frequent ones. In order to do that, for encoding, we reorder nodes at each level using the number of references to it as a key, so that most referenced nodes appear first in a level's array. We then replace pointers with offsets from the beginning of each level array, and chose for each offset the smallest number of bits available in our encoded format (see below).

Memory aligned encoding. While leaf nodes are all of the same size (64 bits), the resulting inner node encoding produces variable-sized records (which is true also for other DAG formats with variable child count, e.g., SVDAG [14]). We have decided, in order to simplify decoding, to use half-words (16 bits) as the basis for our encoding. Our final encoding includes an indexing structure, an array of inner nodes, and an array of leaf nodes. The indexing structure contains the maximum level L and three 32-bits offsets in the inner level array that indicate the start of each level. The layout of inner-level nodes is depicted in Fig. 5.6. For each node, we store in a 16-bit header a 2-bit code for each of the 8 potential children. Tag 00 is reserved to null pointers, which are not stored, while the other tags indicate the format in which child pointers are stored after the header. Children of type 01 use 16 bits, with the leftmost 3 bits encoding the transformation, while the remaining 13 bits encode the offset in number of level-words from the beginning of the next level, where a level-word is 2 bytes for an inner level and 8 bytes for the leaf level. Thus, reflections and references to nodes stored in the first 2^{14} bytes of an inner level's array or in the first 2^{16} bytes of the leaf level can be encoded with just two bytes. Less frequent children pointers, associated to header tags 10 and 11, are both encoded using 32 bits, with the leftmost 3 bits encoding the transformation, and the rightmost ones the lowest $29bits$ of the offset. The highest bit of the offset is set to the rightmost bit of the header tag. We can thus address more than $2GB$ into an inner level, and $8GB$ into leaves.

5.5 Ray-tracing a SSVDAG

The SSVDAG structure can be efficiently traversed using a GPU-based raytracer by slightly adapting other octree-based approaches to apply the transformation upon entering a subtree. In order to test several approaches within the same code base, we have implemented a basic GPU-based raytracer to traverse both SVO, SVDAG, and our SSVDAG. While the structure alone, similar to SVDAG [14], is mostly

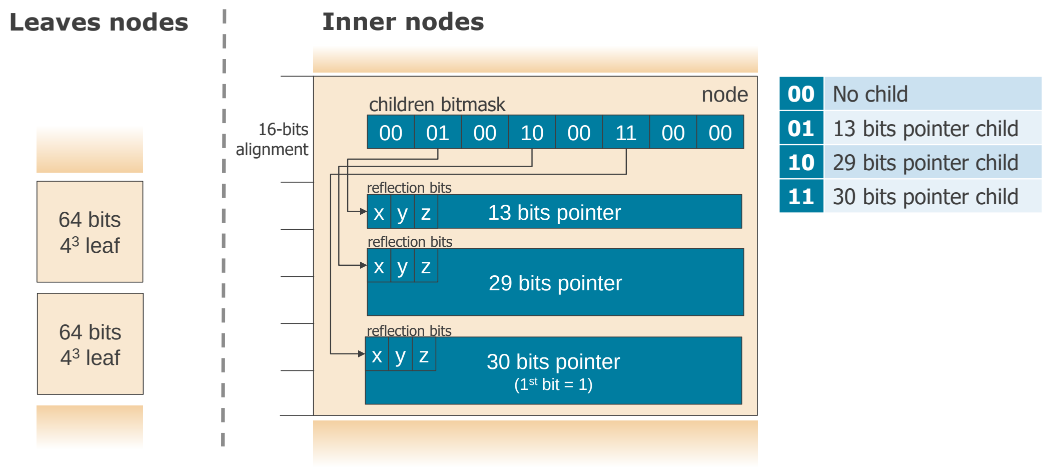


Fig. 5.6.: **Layout of inner nodes in the compact representation.** Leaf nodes are binary coded in a $4x4x4$ grid. Inner nodes have a 16bits alignment with the first byte used as a voxel mask, but also coding information about the size of the variable-length pointers.

useful for visibility queries and/or secondary rays effects, in order to fully test the structure in the simplest possible setting, we use the raytracer both from primary rays (requiring closest intersections) and for hard shadows for the view samples of a deferred rendering target. We focus on these effects, rather than soft shadows and ambient occlusion, since they are the ones where voxelization artifacts are most evident. In the following, we first explain how standard octree traversal code can be minimally transformed to support reflections, and then discuss integration in our sample raytracer to fully render geometric scenes.

5.5.1 Traversal

Many octree raycasting approaches can be adapted to trace SSVDAGs, which differ from regular SVOs or SVDAGs only because they need to apply reflection transformations every time a pointer is followed. As for all trees containing geometric transformations, this can be achieved by one of the two complementary approaches of applying the inverse transformation to the ray or the direct geometric transformation to the voxel geometry.

Transforming the ray. The ray-transformation approach is extremely simple to integrate in stackless traversal approaches [137], which cast rays against a regular octree using kd-restart algorithm [149], and traverse encountered leaves as uniform 3D grids. In this case, once the tagged pointer to the child is traversed, the associated transformation is extracted, and ray reflection is obtained by conditionally performing, for each axis in which the transformation is applied, a mirroring with respect to the node's center c of the ray's current origin together with a sign change of the ray direction. This conditional code can be concisely implemented as follows:

```
org = c + s * (org-c)
dir *= s;
```

where $s \in \mathcal{R}^3$ is equal to -1 in the coordinates in which a mirroring is applied, and $+1$ otherwise. Since the node's center is already maintained in kd-restart algorithms during tree descent in order to locate leaves in the octree, the overhead of mirroring is thus minimal. Traversal stops when-a non-empty voxel is found or the ray span is terminated.

In more optimized implementations, such as *pushdown* or *short-stack*, which avoid restarting from the root of the tree [150], in addition to applying this transformation one would, however, store in the restart cache or stack the modified local ray together with the restart node, which could put more pressure on registers or local memory and reduce parallelization efficiency on GPUs. Moreover, handling changes of ray directions during traversal due to the reflections would increase book-keeping costs in methods not simply based, such as kd-restart, on repeated octree point location.

Transforming the geometry. Directly transforming the geometry allows the ray-tracer to reduce book-keeping costs during traversal, and therefore leads to a more optimized implementation. In this work, we thus discuss how one can traverse our voxel structure using a depth-first visit of the SSVDAG based on a multi-resolution Digital Differential Analyzer (DDA), implemented with a full stack. This kind of DDA-based traversal approach has been proven effective in previous work on SVOs and SVDAGs [131, 14]. All the rendering results in this work are obtained with this implementation.

Listing 5.7 succintly describes our traversal algorithm, and shows the modifications to the standard octree DDA traversal needed to navigate between nodes with symmetries, using only three extra bits to encode reflections (variable `mirror_mask` in the code).

This 3-bit transformation status indicates which reflections must be applied to the indices used to access child pointers in inner nodes or voxel contents in leaf nodes.

```
1  vec3 trace_ray(Ray r, float proj_factor) {
2      t = ray.t_min; mirror_mask = 0; level = 0; cell_size = 0.5;
3      (voxidx, ...) = DDA_init();
4      node_idx = 0; leaf_data = 0;
5      inner_hdr = fetch_inner_hdr(node_idx);
6      mirrored_voxidx = mirror(mirror_mask, voxidx, level);
7      do {
8          bool is_full_voxel = (level<MAX_LEVEL) ?
9              inner_voxel_bit(mirrored_voxidx, inner_hdr) :
10             leaf_voxel_bit(mirrored_voxidx, leaf_data);
11         if (!is_full_voxel) {
12             // Empty -- try to move forward at current level
13             (voxidx, ...) = DDA_next();
14             if (!is_in_bounds(voxidx, level)) {
15                 if (stack_is_empty()) {
16                     return NO_INTERSECTION;
17                 } else {
18                     // Move up and forward at upper level
19                     old_level = level:
20                     (node_idx, inner_hdr, mirror_mask, level) = stack_pop();
21                     cell_size *= (1 << (old_level-level));
22                     (voxidx, ...) = DDA_next();
23                 } // if stack empty
24             } // if !in bounds
25         } else {
26             // Full - return intersection or refine
27             if (level == MAX_LEVEL || resolution_ok(t, cell_size, proj_factor)) {
28                 return t; // INTERSECTION FOUND
29             } else {
30                 // Go down
31                 if (is_in_bounds(DDA_next())) {
32                     stack_push(node_idx, inner_hdr, mirror_mask, level);
33                 }
34                 (m, node_idx)=fetch_tagged_ptr(inner_hdr, node_idx, mirrored_voxidx);
35
36                 // Update DDA and fetch next node
37                 ++level;
38                 cell_size *= 0.5;
39                 (voxidx, ...) = DDA_down();
40                 if (level < MAX_LEVEL-1) {
41                     inner_hdr = fetch_inner_hdr(node_idx);
42                 } else {
43                     // Leaves are 4x4 - must refine DDA
44                     ++level; cell_size *= 0.5;
45                     (voxidx, ...) = DDA_down();
46                     leaf_data = fetch_leaf_data(node_idx);
47                 }
48             }
49         }
50         mirrored_voxidx = mirror(mirror_mask, voxidx, level);
51     } while (t < ray.t_max);
52     return NO_INTERSECTION;
53 }
```

Fig. 5.7.: **Optimized Octree DDA traversal algorithm for SSVDAGs.**

The transformation status is initialized at 0 (line 2), is updated each time we descend in a child (line 40), and is pushed to the stack together with the current node to be able to restore it upon moving up in the hierarchy (line 37).

When descending, as in SVDAG [14], we must access the i-th children pointer by computing an offset within the header equal to the size of all pointers in the interval

$[0, i-1]$, with the only difference that in SVDAG the only two size possibilities are 0 and 4, while in our case tagged pointers can be stored using 0, 2, and 4 bytes. This computation is performed in our shader using a manually unrolled loop within routine `fetch_tagged_ptr` in line 39. Once the tagged pointer to the child is found, the associated transformation code is given by the 3 highest bits, and should be applied to all the nodes in the subtree, which is achieved by xor-ing it with the current `mirror_mask`.

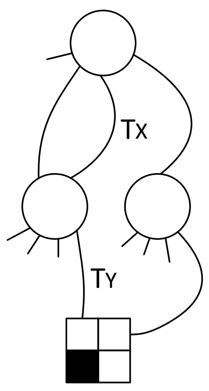
In order to support SSVDAGs, index reflection has applied when accessing voxel occupancy bit at inner and leaf nodes, as well as child pointers associated to a non-empty inner node voxel (line 8) . This is done by transforming a local 3D index v to a node's voxel index that takes into account the current mirroring transformation (function `mirror` in the pseudo-code). This can be obtained by computing the mirrored coordinates as

$$((1,1,1) - 2 \cdot M) * v + M \cdot (S-1)$$

where M is the 3-bits mirroring vector, v the child coordinates and S the cubical size of the voxel, that is 2 for inner nodes, and 4 for the leaves. The difference in size between inner and leaf nodes is also taken into account during the down phase, executing two `DDA_down()` steps instead of one at the last level (lines 47-51).

It should be noted that, in order to minimize memory pressure, similarly to previous work [131], we don't push a node to the stack if the next DDA step would cause the ray to exit from it. This makes it possible to skip useless steps when moving back up in the hierarchy (lines 23-27), but forces us to also put the level in the stack. Moreover, in order to avoid refetching a parent node's header when going up from the child, the header contents is also pushed. With this organization, data is fetched from global memory only when going down, either to fetch the pointer to follow (line 39), or the data associated to inner node (line 46) or leaf node (line 51). Fig. 5.8 shows an example execution of the traversal algorithm, illustrating a full traversal with conditional stack push.

During traversal, moreover, we maintain the voxel cell size, which is updated at the initialization (line 2), when moving up (line 26), and when moving down (line 43 as well as line 49 for leaf nodes). Maintaining this size makes it possible to stop the traversal when the projected size of a voxel goes below a threshold (line 32), which makes it possible to effectively implement levels of detail.



N.It.	step	mirror mask	stack size
	init	X 0000 Y 0000	0
0	↓	X 0010 Y 0000	1
1	↓	X 0010 Y 0010	2
2	↗	X 0010 Y 0010	2
3	↑ ↗	X 0010 Y 0000	1
4	↗	X 0010 Y 0000	1
5	↑ ↗	X 0000 Y 0000	0
6	↓	X 0000 Y 0000	1
7	↗	X 0000 Y 0000	1
8	↓	X 0000 Y 0010	2
9	**hit**		

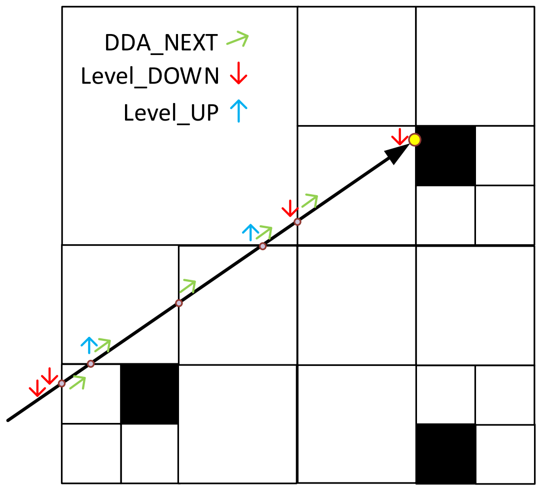


Fig. 5.8.: **Example of ray traversal through the SSVDAG structure.** At left, the example tree, with mirror tags on some pointers. At center, the table of iteration with main actions, current node mirror mask and stack size. At right, a representation of same traversal, with the ray passing though the structure up to find the first intersection.

5.5.2 Scene rendering

Our implementation uses OpenGL and GLSL. The dataset is fully stored in two texture buffer objects, one for the inner nodes, and one for the leaves. For large datasets that exceed texture buffer objects addressing limits, we use 3D textures.

Sparse voxel DAGs structures are typically used to accelerate tracing for secondary rays, while camera rays are rendered using other structures capable to store normals and material properties (see, e.g., [14], which uses rasterization for the camera pass). The high compression factor makes it possible, however, to faithfully represent geometry also for primary rays. We have thus implemented a simple deferred shading renderer which uses only our compressed structure to navigate massive models.

In the first pass, the depth buffer is generated by tracing rays from the camera using the algorithm in Listing 5.7, computing the `proj_factor` parameter of the tracing routing from the camera perspective transformation in order to stop traversal when projected voxel size fall below the prescribed tolerance (1 voxel/pixel in our results). The normals required for shading are then obtained in a second pass by finite differences in the depth buffer using a discontinuity preserving filter. Other shading passes are then performed to compute hard shadows and/or ambient occlusion, as shown in Fig.5.9. While in other settings other structures can be used for storing normals and material properties (see, e.g., [14, 135]), this approach also shows

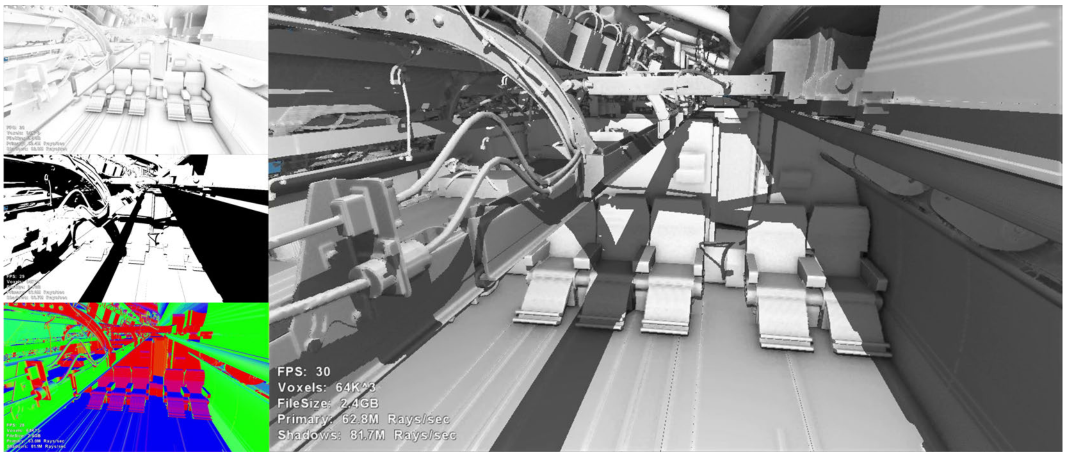


Fig. 5.9.: **Different render layers and final frame.** In the left, from top to bottom: screen space ambient occlusion layer, hard shadows layer, computed screen space normals. In the right, the final frame, shaded using the previous frame buffers plus the depth buffer.

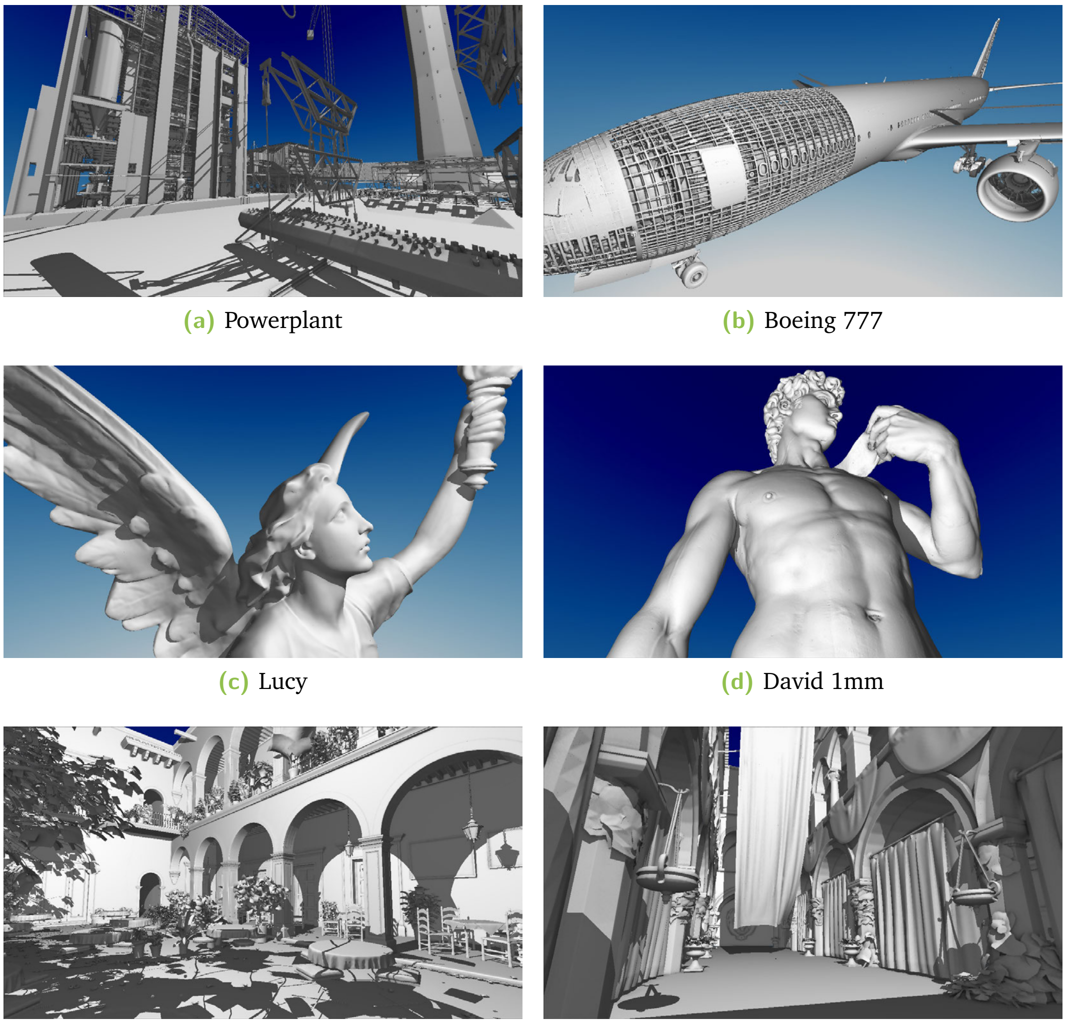
a practical way to implement real-time navigation of very large purely geometric scenes from a very compressed representation.

5.6 Results

An experimental software library, preprocessor and viewer application have been implemented on Linux using C++, OpenGL and GLSL shading language. All the processing and rendering tests have been performed on a Desktop Linux PC equipped with an Intel Core i7-4790K, 32 GB of RAM and an NVIDIA GeForce 980 GTX with 4GB of video memory.

5.6.1 Datasets

We have extensively tested our system with a variety of high resolution surface models. Here we present six models which have been selected to cover widely different fields: CAD, 3D scans and video-gaming (see Fig. 5.10). The CAD models, the Powerplant (12 MTriangles) and the extremely large and complex Boeing 777 (350M triangles) have been chosen to prove the effectiveness of our method with extremely high resolution datasets with connected interweaving detailed parts of complex topological structure, thin and curved tubular structures, as well as badly tessellated models that do not always create closed volumes. The 3D Scans represent, Lucy (28M triangles) and Michelangelo's David 1mm (56M triangles), are representatives of dense high


(a) Powerplant (b) Boeing 777

(c) Lucy (d) David 1mm

(e) San Miguel (f) Crytek Sponza

Fig. 5.10.: **The scenes used in our experiments.** All images are interactively rendered using our raytracer from fully GPU-resident data using deferred shading with screen-space normal estimation and hard shadows.

resolution scans of man-made objects with small details and smooth surfaces. The fourth and fifth model are the San Miguel dataset (7.8M triangles) and the Crytek Sponza dataset (282K triangles)), which are similar to what can be found on a video-game settings, and, together with Lucy, also provide a direct comparison point with the work on SVDAGs [14] .

5.6.2 DAG reduction speed

The preprocessor transforms a 3D triangulation into a SVO stored on disk and then compresses it using different strategies. In our out-of-core implementation, the SVO

		$2K^3$	$4K^3$	$8K^3$	$16K^3$	$32K^3$	$64K^3$
Powerplant	MVoxels	4	17	72	310	1336	5827
	SVDAG Time (s)	0.1	0.4	1.3	3.1	8.6	29.1
	SSVDAG Time (s)	0.3	0.8	2.5	6.3	17.4	52.8
Boeing 777	MVoxels	12	57	268	1242	5699	24633
	SVDAG Time (s)	0.3	1.6	7.2	29.6	121.5	495.71
	SSVDAG Time (s)	1.3	6.0	25.4	107.5	447.1	1846.2
Lucy	MVoxels	6	25	99	395	1580	6321
	SVDAG Time (s)	0.2	0.6	2.3	9.3	37.7	131.7
	SSVDAG Time (s)	0.6	2.1	8.8	37.8	146.2	472.9
David 1mm	MVoxels	4	16	64	257	1029	4116
	SVDAG Time (s)	0.1	0.4	1.7	6.7	25.8	91.2
	SSVDAG Time (s)	0.4	1.5	5.9	25.5	101.2	342.7
San Miguel	MVoxels	12	46	187	750	3007	12045
	SVDAG Time (s)	0.1	0.4	1.6	6.0	19.4	70.4
	SSVDAG Time (s)	0.4	1.4	5.3	19.1	65.8	232.8
Crytek Sponza	MVoxels	40	160	641	2568	10276	41124
	SVDAG Time (s)	0.2	0.7	2.5	10.0	33.5	116.0
	SSVDAG Time (s)	0.6	1.9	6.5	22.5	73.8	226.6

Tab. 5.1.: **Comparison of compression reduction timings.** Resolutions are stated in the top row. For each dataset, the first row is the count of non-empty voxels, the second row is the time taken to reduce the SVO to SVDAG, and the third row is the time taken to reduce the SVO to SSVDAG.

construction using an octree rasterizer can also be interleaved with DAG compression using the multipass strategy.

Preprocessing statistics for the various datasets at different resolutions are reported in Table 5.1, which indicates the time taken to compress datasets from SVO to SVDAG, as well as to SSVDAG. These times do no include rasterization computation. The compressor uses OpenMP to parallelize the code, and 8 parallel processes were active in parallel to reduce subtrees. As one can see from the table, the SSVDAG code is slower on factor ranging over $1.8\times$ and $3.8\times$. with respect to SVDAG, which is due to the overhead caused by computing self-similarity.

As a comparison, Crassin et al. [148] report for the Crytek Sponza dataset a building time of 7.34ms for the resolution 512^3 on an NVIDIA GTX680 GPU, and Kämpe et al. [14] report 4.5s for building an SVDAG from a SVO at resolution $8K^3$ on an Intel Core i7-3930, while the out-of-core scalable voxelizer of Pätzold and Kolb [151] builds a $8K^3$ SVO for the Crytek Sponza dataset in 98s.

Our conversion times are thus similar to previous node reduction works, and are in any case about faster than the first rasterization step required for creating an out-of-core SVO from the original dataset. In addition, about 4% of the time in our conversion is due to the frequency-based pointer compaction step, which requires a reordering of nodes.

	Technique	**Bitrate** ($64K^3$)
Powerplant (12 MTri)	SSVDAG	0.123
	USSVDAG	0.188
	ESVDAG	0.156
	SVDAG	0.241
	SVO	2.390
Boeing 777 (350 MTri)	SSVDAG	0.788
	USSVDAG	1.250
	ESVDAG	1.467
	SVDAG	0.241
	SVO	2.365
Lucy (28 MTri)	SSVDAG	0.852
	USSVDAG	1.215
	ESVDAG	1.190
	SVDAG	1.608
	SVO	2.666
David 1mm (56 MTri)	SSVDAG	0.992
	USSVDAG	1.459
	ESVDAG	1.393
	SVDAG	1.913
	SVO	2.667
San Miguel (7.8 MTri)	SSVDAG	0.206
	USSVDAG	0.355
	ESVDAG	0.260
	SVDAG	0.433
	SVO	2.660
Crytek Sponza (282 KTri)	SSVDAG	0.064
	USSVDAG	0.089
	ESVDAG	0.085
	SVDAG	0.115
	SVO	2.665

Tab. 5.2.: **Comparison of bitrates in bits/voxel for $64k^3$ resolutions**: the proposed Symmetry-aware Sparse Voxel DAG (SSVDAG) is compared with the original sparse voxel DAG (SVDAG), and the pointerless SVO. In order to evaluate the effects of the different optimizations, we also provide results for a version of SSVDAG without pointer compression (USSVDAG) and without symmetry detection (ESVDAG).

5.6.3 Compression performance

Tables 5.3, 5.4 and 5.2 provide detailed information on processing statistics and compression rates of all the test models. We compare our compression results to SVDAG [14], as well as to the pointerless SVOs [145], where each node consumes one byte, a structure that cannot be traversed in random order but useful for off-line storage. For a comparison with ESVO [131], please refer to the original paper on SVDAGs [14]. It should be noted that the slight differences in number of non-empty

		Total number of nodes in millions					
	Structure	$2K^3$	$4K^3$	$8K^3$	$16K^3$	$32K^3$	$64K^3$
Powerplant	SSVDAG	0.1	0.2	0.4	1.0	2.3	5.4
(12 MTris)	SVDAG	0.1	0.2	0.5	1.2	2.9	7.0
	SVO	1.1	5.0	22.0	94.4	404.9	1741.0
Boeing 777	SSVDAG	0.3	0.9	3.2	11.3	40.0	140.0
(350 MTri)	SVDAG	0.4	1.3	4.3	14.4	48.3	164.4
	SVO	3.3	15.6	72.8	341.0	1582.8	7282.0
Lucy	SSVDAG	0.1	0.4	1.4	4.8	14.4	40.3
(28 MTri)	SVDAG	0.2	0.5	1.7	5.7	18.3	52.9
	SVO	2.0	8.2	32.9	131.6	526.6	2106.8
David 1mm	SSVDAG	0.1	0.3	1.1	3.6	11.2	31.8
(56 MTri)	SVDAG	0.1	0.4	1.3	4.2	13.8	41.5
	SVO	1.3	5.4	21.5	85.9	343.2	1372.4
San Miguel	SSVDAG	0.1	0.3	0.9	2.6	7.7	21.6
(7.8 MTri)	SVDAG	0.1	0.3	1.1	3.1	9.1	26.5
	SVO	3.8	15.3	61.7	248.2	997.8	4004.4
Crytek Sponza	SSVDAG	0.1	0.4	1.1	2.9	7.6	19.7
(282 KTri)	SVDAG	0.2	0.5	1.4	3.7	9.8	25.5
	SVO	12.8	52.6	212.6	853.9	3421.5	13697.4

Tab. 5.3.: **Comparison of node reduction for various data structures**: the proposed Symmetry-aware Sparse Voxel DAG (SSVDAG) is compared with the original sparse voxel DAG (SVDAG) and the SVO.

nodes in the SVO structure with respect to Kämpe et al. [14] is due to the different voxelizers used in the conversion from triangle meshes.

Memory consumption obviously depends both on node size and node count. We therefore include for our structure results using uncompressed nodes (USSVDAG in Table 5.4), with the same encoding employed for SVDAGs [14], as well as results using our optimized layout (SSVDAG in Table 5.4). SVDAGs cost 8 to 36 bytes per node, depending on the number of child pointers. Our uncompressed SSVDAGs have the same cost, since symmetry bits are stored in place of padding bytes. On the other hand, our compressed SSVDAGs cost 4 to 34 bytes per node, depending both on the number of child pointers and their size, computed according on the basis of a frequency distribution. In order to assess the relative performance of our different optimizations, we also include results obtained without including symmetry detection but encoding data using our compact representations (ESVDAG).

As we can see, all the DAG techniques outperform the pointerless SVO consistently at all but the lower resolutions, even though they offer in addition full traversal capabilities. Moreover, our strategies for node count and node size reduction prove successful. The USSVDAG structure, on average, occupies at $64K^3$ resolution only 79.6% of the storage required by the SVDAG structure, thanks to the equivalent reduction in the number of nodes provided by our similarity matching strategy. An

		Memory consumption in MB					
	Technique	$2K^3$	$4K^3$	$8K^3$	$16K^3$	$32K^3$	$64K^3$
Powerplant	SSVDAG	0.7	2.0	5.2	13.3	33.8	85.8
(12 MTri)	USSVDAG	1.6	4.0	9.7	23.1	54.9	130.5
	ESVDAG	0.8	2.4	6.3	16.2	41.9	108.6
	SVDAG	1.9	4.9	11.8	28.7	69.9	167.3
	SVO	1.1	4.8	20.9	90.05	386.2	1660.3
Boeing 777	SSVDAG	3.0	11.7	43.1	162.9	616.2	2314.4
(350 MTri)	USSVDAG	6.9	24.8	83.8	295.0	1042.8	3671.5
	ESVDAG	3.9	15.2	54.0	199.1	731.1	2740.7
	SVDAG	9.2	33.8	112.9	376.7	1260.6	4307.9
	SVO	3.1	14.9	69.4	325.2	1509.5	6944.6
Lucy	SSVDAG	1.5	5.3	19.1	67.2	213.6	642.2
(28 MTri)	USSVDAG	2.9	9.2	32.3	110.3	332.4	915.5
	ESVDAG	2.1	6.8	24.0	86.5	295.1	896.5
	SVDAG	4.0	11.3	36.8	127.3	419.0	1212.0
	SVO	2.0	7.8	31.3	125.5	502.2	2009.2
David 1mm	SSVDAG	1.1	3.9	13.6	48.1	159.2	486.7
(56 MTri)	USSVDAG	2.2	7.0	23.3	79.7	252.7	716.1
	ESVDAG	1.5	5.0	17.1	61.3	214.3	683.3
	SVDAG	3.0	8.8	27.3	91.8	308.0	938.6
	SVO	1.3	5.1	20.5	81.9	327.3	1308.8
San Miguel	SSVDAG	1.0	3.3	10.3	31.9	98.7	295.9
(7.8 MTri)	USSVDAG	2.1	6.9	21.3	61.8	181.1	509.8
	ESVDAG	1.1	3.7	12.0	37.6	118.7	373.0
	SVDAG	2.3	7.8	24.6	72.0	212.2	621.3
	SVO	3.6	14.6	58.9	236.7	951.6	3818.9
Crytek Sponza	SSVDAG	1.7	5.2	15.1	42.1	115.7	315.3
(282 KTri)	USSVDAG	3.4	9.7	25.8	67.4	172.3	436.8
	ESVDAG	2.1	6.4	18.8	53.6	151.3	417.3
	SVDAG	4.2	11.9	31.8	83.6	218.3	563.5
	SVO	12.21	50.2	202.7	814.3	3263.0	13062.9

Tab. 5.4.: **Comparison of data compression performance for various data structures**: the proposed Symmetry-aware Sparse Voxel DAG (SSVDAG) is compared with the original sparse voxel DAG (SVDAG), and the pointerless SVO. In order to evaluate the effects of the different optimizations, we also provide results for a version of SSVDAG without pointer compression (USSVDAG) and without symmetry detection (ESVDAG).

average reduction in size to about 52.4% of the SVDAG encoding is obtained by also applying the frequency-based tagged pointer compaction strategy. Such a strategy is particularly successful since, by matching more pointers, increased opportunities for referencing highly popular nodes arise. This is also proved by the results obtained by the ESVDAG techniques, which uses our data structure but only matches sub-trees if they are equal, as in the original SVDAG. The stronger compression of SSVDAGs makes it possible, for instance, to easily fit all the Boeing model into a 4GB graphics board (GeForce GTX 980) at $64K^3$ resolution. Since the Boeing 777 airplane has a length of 63.7m and a wingspan of 60.9m, using a $64K^3$ grid permits to represent details with sub-millimetric accuracy (see Fig. 5.11).

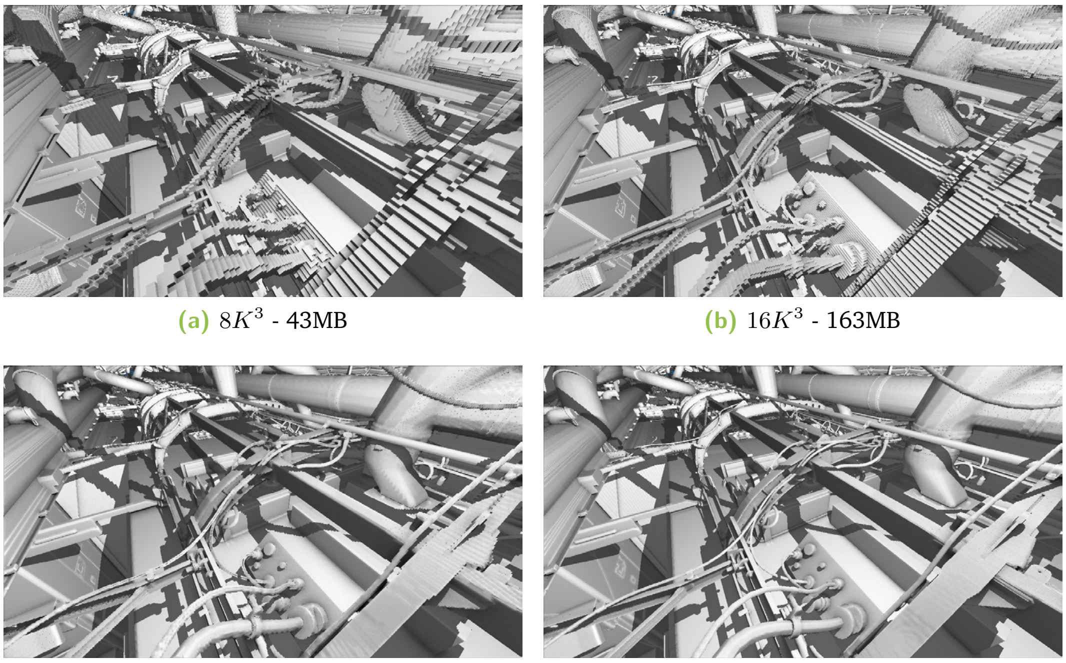

(a) $8K^3$ - 43MB
(b) $16K^3$ - 163MB
(c) $32K^3$ - 617MB
(d) $64K^3$ - 2331MB

Fig. 5.11.: **Detail view of the Boeing scene at different resolutions.** The compression performance of our method supports real-time rendering from GPU-resident data even at $64K^3$ resolution, while SVDAG memory requirements exceed on-board memory capacity on a 4GB board (see Table 5.4).

5.6.4 Rendering

Since our main contributions target compression, we have focused on verifying the correctness of the different DAG structures, as well as on being able to compare their relative traversal performances, rather than absolute speeds. We have thus implemented a generic shader-based raytracer that shares general traversal code based for the various DAG structures. The different structures are supported by simply specializing the general code through structure-specific versions of the routines that read a node structure, access a child by following a pointer, and applies reflections to 3D indices (see Sec. 5.5 for details), The SVDAG and USSVDAG version access nodes by fetching data from a GL_R32UI texture buffer object, while SSVDAG uses a GL_R16UI buffer because of the different alignment requirements. 3D textures are used in place of texture buffer objects when the data is so large to exceed buffer addressing limits. This happens only for the Boeing at $64K^3$ resolution for the results presented in this work. The code does not use any other acceleration or shading structure, and normals required for shading are generated in screen space. This simple setup also shows that it is possible to use such a terse structure to support interactive navigation of very large models compressed to the GPU.

We have introduced in the renderer two of the most used voxel-rendering optimizations, i.e., level-of-detail and the beam optimization introded by Laine and Karras [131]. For level-of-detail rendering, we simply stop traversal when the projected size of a voxel goes below a threshold (1 voxel/pixel in these results). This is done by computing the `proj_factor` parameter from the camera perspective transformation. For beam optimization, in a coarse rendering pass, we divide the image into 8x8 pixel blocks and cast a distance ray for the corners of these blocks. Traversal is stopped as soon as we encounter a voxel that is not large enough to certainly cover at least one ray in the coarse grid. This is simply done by adapting the `proj_factor` parameter to the coarse image. The subsequent depth pass then conservatively estimates the starting point of each ray from the four neighboring depth values in the coarse grid. Fig. 5.12 illustrates the results obtained during a real-time captured exploration sequence of Crytek Sponza at $64K^3$ resolution. All images are capture at 720p resolution. As one can see, we can trace approximately 100M primary rays/s on such a massive structure even without optimizations. LOD rendering proves dramatically effective in the overall views (boosting the performance to close to 300M rays/s), but, given the fine-grained structure, also provide a non-negligible performance boost even in the close-ups, where beam optimization also becomes effective. Their combination approximately doubles performance in the walkthrough of the inside of the model.

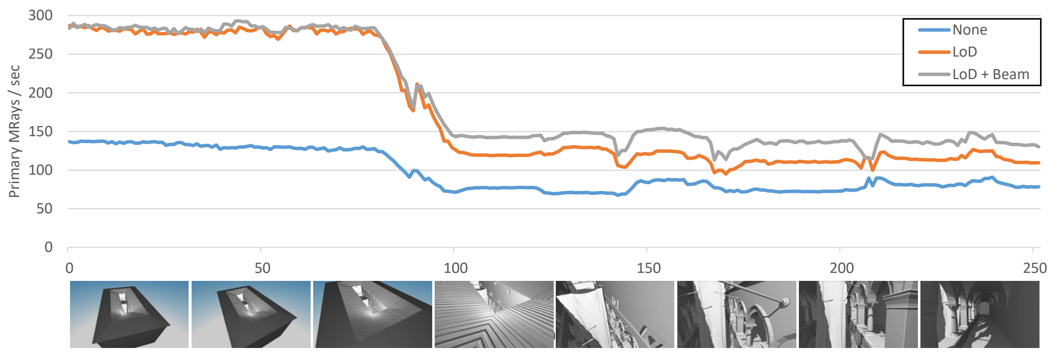


Fig. 5.12.: **Graph of the render performance** with different optimizations during a real-time captured exploration sequence of Crytek Sponza at $64K^3$ resolution, starting from an overall fly-over (frames 1-80), quickly moving inside the model (frames 80-100), and then performing a walkthrough (frames 100-250).

We have obtained similar relative performances for all the models. For instance, for the sample viewpoints in Fig. 5.1 of the Powerplant model at $1M^3$ resolution, the views are rendered from farthest to closest (left to right) ranging from 92 to 45 fps for SSVDAG, from 106 to 52 fps for USSVDAG, and from 107 to 53 fps for SVDAG.

All images are rendered in HD (720p) with screen-space normal estimation and hard shadows for one point light. Rendering performance is thus similar for the three implementations, demonstrating that the reduction in memory consumption does not come at the cost of much increased render times.

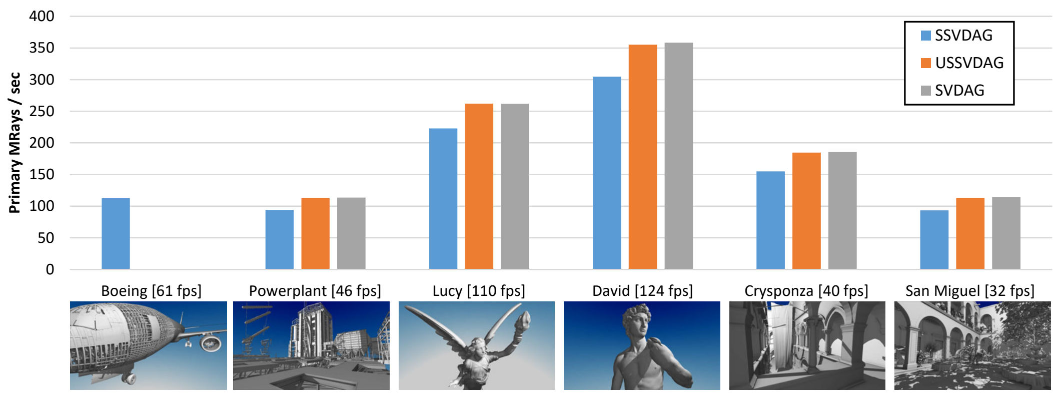


Fig. 5.13.: Comparison of primary rays performance for the different structures. Also the frames per second for the final frame (secondary rays, shading, etc.) is shown in each picture. Mirroring is not affecting traversal times, but encoding, even if its performance is still enough for real-time rates.

Fig. 5.13 compares the performance obtained when tracing primary rays for the different datasets at $64K^3$ resolution, for the reference images indicated below the graph. The Boeing 777 dataset does not fit into GPU memory for the other structures, and we does provide results only for SSVDAGs. As one can see, we can trace from 80M to 300M primary rays/s depending on viewpoint complexity, and the performance of the various structures, as for the Powerplant example discussed above, is consistently similar. It should in particular be noted that reflections impose a very little overhead, since USSVDAG is only 1%-2% slower than SVDAG, while variable-rate pointer compression proves a little bit more costly, since SSVDAG has an overhead of 14%-16% with respect to SVDAG. This is probably due to the fact that, while the extra computation required for implementing reflections is well hidden by memory latency, the more elaborate memory layout of pointer compression is more costly. This aspect leaves room for optimization.

Even with our unoptimized shader-based implementation, our SSVDAG structure supports real-time performance for very complex scenes. The Boeing 777 scene can be explored at $64K^3$ resolution in HD (720p) with shading and shadows, at about the same performance as the Powerplant model. Fig. 5.11 shows images taken from the same closeup viewpoint rendered with various voxel resolutions. It is evident how the small voxel dimensions enabled by our compression let appreciate important details that are lost at lower resolutions. The highest resolution is only possible with our SSDAG and USSDAG methods, which are the only ones capable to fit the entire model in-core in a 4GB board.

5.7 Discussion

We have shown that Symmetry-aware Sparse Voxel DAGs (SSDAGs), an evolution of Sparse Voxel DAGs, allow for an efficient lossless encoding of voxelized geometry representations, in which subtrees that are identical up to a similarity transformations appear only once. Our results demonstrate that this sort of geometric redundancy is common in all tested real-world scenes, ranging from complex CAD models to 3D scans to gaming models, leading to state-of-the-art lossless compression performance. The increased node size with respect to SVOs and SDAGs is quickly balanced by a sizeable reduction in node count. Moreover, pointer overhead is reduced by using fatter leaves and a simple entropy coding scheme. The resulting structure is compact and GPU-friendly, which makes it possible to trace very large scenes while maintaining the visibility acceleration structure fully resident in GPU memory. As the structure can be efficiently constructed from external memory, the resulting method is fully applicable to massive data sets, as demonstrated here on large scenes such as the Boeing 777, whose original description exceeds 350M triangles.

As for many data structures and acceleration techniques, our SSVDAG approach has also a number limitations. Handling these limitations indicates interesting areas for future work.

First of all, despite the fact that our compression scheme is lossless, relying on a voxelization scheme leads to a discretization of the original geometry, which, while typically very effective in terms of traversal speed, is not guaranteed to be the most effective in terms of compactness of representation, e.g., for low-poly scenes, or image quality, e.g., for extreme close-up views. These problems are not unique to our method, but typical of all pure voxelization techniques. By increasing compression rates, we significantly improve quality vs. memory costs, allowing for much deeper octrees, but do not eliminate blockiness, which may appear at extreme zoom levels.

A second current limitation of our method is that, while the concept of compression using symmetric DAGs is general, our current implementation is limited to handling only reflective symmetries, and some of the implementation choices explicitly take into account the properties of mirroring transformation (e.g., order independence).

While this approach simplifies implementation, it also likely reduces the compression potential, and leaves room for improvement. While the current implementation uses reflections only, an interesting avenue for future work would thus be to investigate other symmetries. A particularly interesting approach would be, moreover, to evaluate how the method could extend from lossless to lossy compression, by allowing for partial matches of subtrees instead of exact identity up to a transformation. The effect of the errors induced by such approximate techniques should be evaluated in terms of quality/cost ratio depending on the actual usage, which ranges from primary rays to hard and soft shadows).

Moreover, an additional limitation identified by our benchmarks is that, while the gain in compression rates due to merging symmetric subtrees appear to come at no rendering cost, a tracing overhead of up to 15% appears to be associated to the pointer compaction scheme. This is likely due to the overhead of fetching non-aligned data from GPU memory and to the need to perform bitwise operations for pointer decoding. It should be evaluated whether better layout schemes, or improved shader codes, could reduce this overhead. On the other hand, rearranging nodes based on reference frequency, which is at the core of the pointer compression techniques, has proven very effective, and it would be interesting to evaluate how such rearranging techniques could be further expanded, for example to achieve a better encoding in low-sharing areas.

Finally, similarly to other works on DAG compression [14, 133, 134], the scheme presented in this work only supports compression of geometry. While this is acceptable to compute occlusions and for shadowing, general use of the structure would require to map non-geometric properties to voxels (e.g., material or reflectance properties). A promising approach in that respect has been recently presented by Dado et al. [135] for graphs without symmetries and could be adapted to our technique. An alternative solution would be to totally decouple geometry from material representations, using compressed representations of volumetric textures to overlay a material layer on top of a geometric scene. For maximum compression, it would also be interesting to evaluate how shading normals could be evaluated by differentiating the geometry not only in screen space, but also for secondary rays, expanding techniques used in current volumetric renderers [147].

5.8 Bibliographical notes

An early version of this work was presented in the *ACM SIGGRAPH Symposium on Interactive 3D Graphics 2015* (i3D) conference [6]. A revised version, introducing a new construction methods and improved rendering techniques was published in 2017 in the *Journal of Computing Graphics Techniques* (JCGT) [7]. The work was also discussed in a tutorial presented at Eurographics 2018 [8]. Much of the contents of this chapter comes from the JCGT contribution. This chapter includes an extended evaluation with additional very large examples.

Several authors have proposed follow-ups that extend our method in different direction or exploit some of its components. In particular, different work have proposed to compress and encode also the attributes as colors or normals [135, 152], while using similar methods to compress pointers. *GVDB* [142] shows an alternative DDA algorithm to raytrace a hierarchy of grid containing volumetric scalar data. very recently, Duan et al. [153] introduced Exclusive Grouped Spatial Hashing (EGSH) to compresses repetitive data into tiny compact hash tables without while maintaining simple random access to the GPUs, rather than maintaining hierarchical access as in our case.

Summary and conclusions

6

This thesis has introduced scalable techniques that advance the state-of-the-art in massive model creation and exploration. In a preliminary work, concerning model creation, we have focused on methods for improving reality-based scene acquisition, processing, and creation, introducing an implementation of scalable out-of-core point clouds and a data fusion approach for creating detailed models from cluttered data acquisition. The enabling technology for the exploration of large datasets is the core of this thesis, which has introduced two orthogonal techniques for the high-quality exploration of very large models. The first is an adaptive out-of-core technique that supports non-local illumination using work-batches and visibility queries. The second is an aggressive compression method that exploits redundancy in large models to aggressively compress data so that it fits, in fully renderable format, in GPU memory. This final chapter summarizes the results obtained, and briefly discusses the most promising directions for future work.

6.1 Overview of achievements

THE availability of highly detailed 3D content is growing at fast pace thanks to the rapid evolution of 3D acquisition and 3D model creation techniques. Such detailed 3D models are becoming increasingly common and represent a very useful tool for many application domains, including ranging from architecture, engineering, and cultural heritage to simulation and gaming. The massive size of such models makes handling these datasets very complex, and requires scalable solutions at all processing stages.

In this thesis, in a preliminary work, I have focused on the problem of improving the creation of reality-based models, starting from massive amounts of data acquired using digital photography and range scanning. In this respect, the research carried out witin this thesis has led to the following achievements: have been the following:

- The introduction of a general and multiresolution design for a scalable system to create, colorize, analyze, and explore massive point clouds totally out-of-core. A GPU-accelerated implementation able to process and render a billion points dataset [9] and its application to fields like cultural heritage or engineering [10, 11].

- An easy-to-apply acquisition protocol based on laser scanning and flash photograph to generate colored point clouds [12], which introduces a novel semi-automatic method for clutter removal and photo masking to generate clean point clouds without clutter using minimal manual intervention. The multiresolution design previously introduced allows that the entire masking, editing, infilling, color-correction, and color-blending pipeline to work fully out-of-core without limits on model size and photo number.

This approach has been tested on several large world datasets, showing, in particular, the capability to be applied at a very large scale. The example presented (see Sec. 3.4.1) is an acquisition campaign that has covered 37 human-size statues mounted on metallic supports, with color and shape acquired at a resolution of 0.25mm for over 1billion geometric and colorimetric samples.

Following this initial work, I focused mainly on the exploration problem, with the goal of creating enabling technology to support real-time interactive exploration of massive models. This research has been carried out by exploring two orthogonal directions: smart work and data decomposition and extreme compression. In the first direction, the goal was to devise techniques capable to smartly decompose a scene into coherent batches, only loading and processing the minimum amount of data that we judge contributing to the current image, with the goal of considering both direct and indirect illumination. In the second direction, the goal, instead, was to exploit data redundancy to aggressively compress data so that geometry and acceleration structures fully fit, in fully renderable format, in GPU memory, thus making it possible to perform ray-tracing at GPU speed without data loading delays. The research work carried out on these subjects has led to the following achievements:

- A novel approach to exploit the rasterization pipeline and hardware occlusion queries in order to create coherent batches of work for localized shader-based ray tracing kernels [5]. By combining hierarchies in both ray-space and object-space, the method is able to share intermediate traversal results among multiple rays. Then, temporal coherence is exploited among similar ray sets between frames and also within the given frame. This scheduling architecture naturally allows for out-of-core ray tracing, with the possibility of rendering potentially unbounded scenes.

- A novel compression method called SSVDAG (Symmetry-aware Sparse Voxel DAG) [6, 7], which can losslessly represent a voxelized geometry of many real-world scenes, aside with an out-of-core algorithm to construct such representation from a SVO or a SVDAG, as well as a clean modification of standard GPU raycasting algorithm to traverse and render this representation with a small overhead. This technique has proven to compress up to a $1M^3$ voxel grid to fit completely in-core and render it in realtime.

As demonstrated in Sec. 3.4.7, Sec. 4.7 and Sec. 5.6, the results obtained matched the expected performance, as described in Sec. 2.4 for all the tested datasets.

6.2 Discussion and future work

The thesis has tackled both the problem of creation and exploration of massive models using scalable techniques. While my contribution to point-clouds are mostly at the practical level, showing a novel best-practice full-fledged implementation that solves very practical problems, the approaches studied for improving real-time exploration are more interesting in terms of potential for future work.

The batch-based technique presented a novel use of hierarchical occlusion culling for accelerating OpenGL-based ray tracing, exploiting the rasterization pipeline and hardware occlusion queries in order to create coherent batches of work for a GPU ray-tracing kernel. This work fits well in the area of massive-model rendering, since it makes it possible to exploit the visibility-driven adaptive-loading technique typical of current out-of-core rasterization renderers in a more flexible ray-tracing setting, opening the door to GPU-accelerated out-of-core rendering using shadows and global illumination. This is achieved in a very simple context, using a technique that can be easily integrated using current renderers. We have demonstrated non-local effects on large models through shadows and first-bounce diffuse illumination. It is interesting to note, as mentioned in Sec. 4.9 that the idea of visibility-driven acceleration using a dual hierarchy is currently being exploited for very advanced global illumination works, such as full global illumination using many-view rendering [129].

While the batch-based technique makes it possible to render very large scenes, exceeding GPU memory, using adaptive loading, my compression contributions tackle the problem by aggressively compressing data so that it fits, in fully renderable format, on GPU memory. I consider this approach my main research contribution.

Such an approach, in fact, fits extremely well with the current trend of increased memory budgets near the GPUs. Such memory budget, of several GBs, are still way too low to allow for rendering of uncompressed large models, but are starting to be non-negligible. With my work, I have proven that, by exploiting similarities and clever encodings, sparse voxel octrees, a very GPU-friendly, but memory-hungry representations, can be transformed to very compact Symmetry-aware Sparse Voxel DAGs for a large variety of model kinds, including laser scans, CAD models, and gaming models. Such very compact representations are hundreds of time smaller than the original voxel data, and, thus, require little off-line and GPU storage, as well as little bandwidth for the transmission up to the GPU. At the same time, the GPU-friendly encoding supports ray tracing without performance loss.

In this thesis, similarly to other works on DAG compression [14, 133, 134], I have mostly worked on compression of geometric data. Other authors have already shown how to associate to such compressed geometric data attributes such as colors or normals [135, 152]. Besides evaluating how these techniques can be improved by also taking into account symmetries, several interesting areas for future work can be identified.

First of all, the focus, so far, has been on lossless compression of voxelized representation. It seems very promising, instead, to also consider lossy (or near-lossless) approaches, while remaining in the same settings. Since voxelized representations are already discretization of another geometry (e.g., triangles, point clouds, implicit surfaces, higher order patches, ...) it seems reasonable to consider that it could be slightly varied to improve chances of finding similarities. This could be implemented using a prefiltering approach which reduces leaf-level variability (while conserving some errors) to increase the number of similar subtrees. Such an approach, which promises a much higher compression, has not been attempted so far.

A second very practical improvement would concern, instead, a system-level study of incremental loading using the compressed structure. So far, aggressive compression using a DAG representation has always been used for monolithic structures maintained in GPU. This is a very favorable situation, since full GPU residency supports very efficient rendering. It would be interesting to study, especially in the context of networked rendering, how these very compact representation could be used in an incremental loading context. This would require, at the system level, to implement a paging system, and at the compression level an improvement in data reordering to better support data locality, so that spatially close data is likely to be stored in the same page. Such locality is currently totally destroyed by the similarity-based reordering performed at compression level. It seems reasonable that one could find compromises to optimized at the same time for data and spatial similarity. Such an implementation would permit, for instance, to achieve a very efficient ray-based renderers on the Web (and, eventually, even on mobile devices), reducing data loading latency.

6.3 Bibliographical results

The scientific results obtained during this PhD work also appeared in related publications, sorted by their introduction in this thesis, listed below:

- ***A Multiresolution System for Managing Massive Point Cloud Data Sets***. *A. Jaspe Villanueva, Omar A. Mures, E. J. Padrón and J.R. Rabuñal. Tech Report ToVIAS project, Universidade da Coruña (2014)*
  – This is the original work introducing my implementation of the point-cloud subsystem.
- ***Point Cloud Manager: Applications of a Middleware for Managing Huge Point Clouds***. *O. A. Mures, A. Jaspe Villanueva, E.J-Padrón, J.R. Rabuñal. Chapter 13 of "Effective Big Data Management and Opportunities for Implementation" book. Pub. IGI Global. ISBN: 9781522501824 (2016)*
  – This work details the design and use of the point cloud manager for off-line and on-line operations.
- ***Virtual Reality and Point-based Rendering in Architecture and Heritage***. *O. A. Mures, A. Jaspe Villanueva, E.J- Padrón, J.R. Rabuñal. Chapter 4 of "Handbook of Research on Visual Computing and Emerging Geometrical Design Tools" book. Pub. IGI Global. ISBN: 9781522500292 (2016)*
  – This work discusses the integration of the point cloud manager into a VR/AR system and illustrates some test cases in architecture and cultural heritage.
- ***Mont'e Scan: effective shape and color digitalization of cluttered 3D artworks***. *F. Bettio, A. Jaspe Villanueva, E. Merella, F. Marton, E. Gobbetti, R. Pintus. ACM Journal on Computing and Cultural Heritage, Vol 8, Num 1 (2015)*
  – This work presents our acquisition protocol and processing method for creating seamless colored models from the fusion of photometric and range scan data, in the complex case of cluttered models.

- ***CHC+RT: Coherent Hierarchical Culling for Ray Tracing***. *O. Mattausch, J. Bittner, A. Jaspe Villanueva, E. Gobbetti, M. Wimmer, and R. Pajarola. Computer Graphics Forum Journal Vol 32, Num 2. Presented at Eurographics'15 (2015)*
  – This is the work that introduced our method for exploiting the rasterization pipeline and occlusion queries to efficiently implement out-of-core ray tracing.
- ***SSVDAGs: Symmetry-aware Sparse Voxel DAGs***. *A. Jaspe Villanueva, F. Marton, and E. Gobbetti. ACM SIGGRAPH i3D full paper (2016)*
  – This is the original work that introduced our technique to exploit symmetries for data-reduction of voxel DAGs.
- ***Symmetry-aware Sparse Voxel DAGs (SSVDAGs) for compression-domain tracing of high-resolution geometric scenes***. *A. Jaspe Villanueva, F. Marton, E. Gobbetti. Journal of Computer Graphics Techniques, Vol 6 Num 2 ISSN: 2331-7418 (2017)*
  – This is an extended version of the i3D contribution, which introduces improved construction and rendering techniques and presents an expanded evaluation.
- ***Voxel DAGs and Multiresolution Hierarchies: From Large-Scale Scenes to Pre-computed Shadows***. *U. Assarsson, M. Billeter, D. Dolonius, E. Eisemann, A. Jaspe Villanueva, L. Scandolo, E. Sintorn. Eurographics Tutorials (2018)*
  – This is a tutorial on the state-of-the-art of Voxel DAGs. I prepared and delivered the section on Advanced DAG Encoding and presented tests and evaluations on very large models.

In addition, during the course of my thesis, I have also contributed to the following related publications, which have not been included in this work:

- ***SOAR: Stochastic Optimization for Affine global point set Registration***. *M. Agus, E. Gobbetti, A. Jaspe Villanueva, C. Mura, and R. Pajarola. International Symposium on Vision, Modeling, and Visualization VMV'14 full paper (2014)*
  – This work introduce a stochastic algorithm for pairwise affine registration of partially overlapping 3D point clouds with unknown point correspondences. It is related to the problem of point-cloud data creation, but was not included here since it does not target massive data. I contributed mainly in the evaluation of the method using real-world and synthetic data.
- ***Practical line rasterization for multi-resolution textures***. *J. Taibo, A. Jaspe Villanueva, A. Seoane, Marco Agus, and L. A. Hernandez. STAG'14 full paper (2014)*
  – This work introduce a method for draping 2D vectorial information over

a multi-resolution 3D terrain elevation model using the OpenGL pipeline. I co-designed the techniques and worked on their implementation.

- ***Robust Reconstruction of Interior Building Structures with Multiple Rooms under Clutter and Occlusions***. *C. Mura, O. Mattausch, A. Jaspe Villanueva, E. Gobbetti, and R. Pajarola. CAD/Graphics'13 full paper (2014)*
  – This work also focuses on point cloud data, but with the goal to recover structural information. My contribution was mainly in the evaluation of the method using generated ground-truth data through virtual scanning.
- ***Reconstructing Complex Indoor Environments with Arbitrary Wall Orientations***. *C. Mura, O. Mattausch, A. Jaspe Villanueva, E. Gobbetti, and R. Pajarola. Eurographics Posters (2014)*
  – This is a follow-up work of the previous publication which introduced several speed-ups.
- ***Automatic room detection and reconstruction in cluttered indoor environments with complex room layouts***. *C. Mura, O. Mattausch, A. Jaspe Villanueva, E. Gobbetti, R. Pajarola. Computer & Graphics Journal Num 44 (2014)*
  – This is an extended journal version of the previous contributions.
- ***IsoCam: interactive visual exploration of massive cultural heritage models on large projection setups***. *F. Marton, M. Balsa, F. Bettio, M. Agus, A. Jaspe Villanueva, and E. Gobbetti. ACM Journal on Computing and Cultural Heritage, Vol 7, Num 2 (2014)*
  – This work focuses on user interfaces for the exploration of large models. My contribution was in the evaluation of the technique;
- ***ExploreMaps: Efficient Construction and Ubiquitous Exploration of Panoramic View Graphs of Complex 3D Environments***. *M. Di Benedetto, F. Ganovelli, M. Balsa, A. Jaspe Villanueva, R. Scopigno, and E. Gobbetti. Computer Graphics Forum Journal, Vol 33, Num 2. Presented at EuroGraphics'14 (2014)*
  – This work focused on the exploration of large rendered models using image-based methods. I contributed through the automatic generation of datasets and setting up the rendering back-end.
- ***PEEP: Perceptually Enhanced Exploration of Pictures***. *M. Agus, A. Jaspe Villanueva, G. Pintore, E. Gobbetti. International Workshop on Vision, Modeling and Visualization (VMV) full paper (2016)* – This work shows an interesting approach to generate an illusion of depth from a single picturer, supporting the illusion of 3D exploration. I co-invented the method and participated to its implementation and evaluation.

- ***CRS4 Visual Computing***. *E. Gobbetti, M. Agus, F. Bettio, A. Jaspe Villanueva, F. Marton, R. Pintus, and A. Zorcolo. Lab presentations STAG'16 (2016)*
  – This is an overview of the work done at CRS4/Visual Computing.
- ***Artworks in the Spotlight: Characterization with a Multispectral Dome***. *I. Ciortan, T. Dulecha, A. Giachetti, R. Pintus, A. Jaspe Villanueva, and E. Gobbetti. Materials Science and Engineering Journal (2018)*
  – This work focuses on acquisition of materials, instead of the large models treated in this thesis;

Bibliography

[1] Marie Curie Actions. *Data Intensive Visualization and Analaysis ITN*. 2016. URL: `http://www.diva-itn.eu` (cit. on p. xi).

[2] "The American Heritage Dictionary of the English Language". In: *Houghton Mifflin Company* (2007) (cit. on p. 3).

[3] Sung-eui Yoon, Enrico Gobbetti, David Kasik, and Dinesh Manocha. *Real-time Massive Model Rendering*. Vol. 2. Synthesis Lectures on Computer Graphics and Animation 1. Morgan and Claypool, 2008 (cit. on pp. 3–5, 136, 137).

[4] Yangdong Deng, Yufei Ni, Zonghui Li, Shuai Mu, and Wenjun Zhang. "Toward Real-Time Ray Tracing: A Survey on Hardware Acceleration and Microarchitecture Techniques". In: 50 (Aug. 2017), pp. 1–41 (cit. on pp. 5, 6, 137, 138).

[5] Oliver Mattausch, Jiri Bittner, Alberto Jaspe Villanueva, Enrico Gobbetti, Michael Wimmer, and Renato Pajarola. "CHC+RT: Coherent Hierarchical Culling for Ray Tracing". In: *Computer Graphics Forum* 34.2 (2015). Proc. Eurographics 2015, pp. 537–548 (cit. on pp. 6, 9, 54, 82, 116, 138, 147).

[6] Alberto Jaspe Villanueva, Fabio Marton, and Enrico Gobbetti. "SSVDAGs: Symmetry-aware Sparse Voxel DAGs". In: *Proc. ACM i3D*. Feb. 2016, pp. 7–14 (cit. on pp. 6, 10, 87, 91, 94, 114, 117, 138, 147).

[7] Alberto Jaspe Villanueva, Fabio Marton, and Enrico Gobbetti. "Symmetry-aware Sparse Voxel DAGs (SSVDAGs) for compression-domain tracing of high-resolution geometric scenes". In: *Journal of Computer Graphics Techniques* 6.2 (2017), pp. 1–30. ISSN: 2331-7418 (cit. on pp. 6, 10, 87, 114, 117, 138, 147).

[8] Ulf Assarsson, Markus Billeter, Dan Dolonius, Elmar Eisemann, Alberto Jaspe Villanueva, Leonardo Scandolo, and Erik Sintor. "Voxel DAGs and Multiresolution Hierarchies: From Large-Scale Scenes to Pre-computed Shadows". In: *Proc. EUROGRAPHICS Tutorials*. Ed. by Tobias Ritschel and Alexandru Telea. To appear. Apr. 2018 (cit. on pp. 6, 114, 138).

[9] Alberto Jaspe Villanueva, Omar A. Mures, Emilio J. Padrón, and Juan R. Rabuñal. *A Multiresolution System for Managing Massive Point Cloud Data Sets*. Tech. rep. University of A Coruña, 2014 (cit. on pp. 8, 22, 29, 51, 116, 146).

[10] Omar A. Mures, Alberto Jaspe Villanueva, Emilio J. Padrón, and Juan R. Rabuñal. "Point Cloud Manager: Applications of a Middleware for Managing Huge Point Clouds". In: *Effective Big Data Management and Opportunities for Implementation*. Ed. by Manoj Kumar Singh and Dileep Kumar G. IGI Global, June 2016. Chap. 13. ISBN: 9781522501824 (cit. on pp. 8, 51, 116, 146).

[11] Omar A. Mures, Alberto Jaspe Villanueva, Emilio J. Padrón, and Juan R. Rabuñal. "Virtual Reality and Point-based Rendering in Architecture and Heritage". In: *Handbook of Research on Visual Computing and Emerging Geometrical Design Tools*. Ed. by Giuseppe Amoruso. IGI Global, Apr. 2016. Chap. 4. ISBN: 9781522500292 (cit. on pp. 8, 51, 116, 146).

[12] Fabio Bettio, Alberto Jaspe Villanueva, Emilio Merella, Fabio Marton, Enrico Gobbetti, and Ruggero Pintus. "Mont'e Scan: Effective Shape and Color Digitization of Cluttered 3D Artworks". In: *ACM Journal on Computing and Cultural Heritage* 8.1 (2015), 4:1–4:23 (cit. on pp. 9, 22, 51, 116).

[13] Oliver Mattausch, Jiří Bittner, and Michael Wimmer. "CHC++: Coherent Hierarchical Culling Revisited". In: *Computer Graphics Forum* 27.3 (2008), pp. 221–230 (cit. on pp. 17, 57, 63, 64).

[14] Viktor Kämpe, Erik Sintorn, and Ulf Assarsson. "High Resolution Sparse Voxel DAGs". In: *ACM Trans. Graph.* 32.4 (2013), 101:1–101:13 (cit. on pp. 17, 85, 87, 88, 97, 99, 100, 102, 104–107, 113, 118, 144, 149).

[15] Alberto Jaspe Villanueva and Supervisors: Emilio J. Padrón and Javier Taibo Pena. "Point Cloud Manager: A multiresolution system for managing massive point cloud datasets". MA thesis. University of A Coruña, 2013 (cit. on pp. 22, 51).

[16] Omar A. Mures and Supervisors: Alberto Jaspe villanueva and Emilio J.Padrón. "Real-time management tool for massive 3D point clouds". MA thesis. University of A Coruña, 2014 (cit. on pp. 22, 51).

[17] Marc Levoy and Turner Whitted. *The Use of Points as Display Primitives*. Tech. rep. TR 85-022. Department of Computer Science, University of North Carolina at Chapel Hill, 1985 (cit. on p. 22).

[18] J.P. Grossman and William J. Dally. "Point Sample Rendering". In: *Proceedings Eurographics Workshop on Rendering*. Eurographics. Eurographics, 1998, pp. 181–192 (cit. on p. 22).

[19] Hanspeter Pfister and Markus Gross. "Point-Based Computer Graphics". In: *IEEE Computer Graphics and Applications* 24.4 (July 2004), pp. 22–23 (cit. on p. 22).

[20] Markus H. Gross. "Getting to the Point...?" In: *IEEE Computer Graphics and Applications* 26.5 (Sept. 2006), pp. 96–99 (cit. on p. 22).

[21] Markus H. Gross and Hanspeter Pfister, eds. *Point-Based Graphics*. Series in Computer Graphics. Morgan Kaufmann Publishers, 2007 (cit. on p. 22).

[22] Miguel Sainz and Renato Pajarola. "Point-Based Rendering Techniques". In: *Computers & Graphics* 28.6 (2004), pp. 869–879 (cit. on p. 22).

[23] Leif Kobbelt and Mario Botsch. "A Survey of Point-Based Techniques in Computer Graphics". In: *Computers & Graphics* 28.6 (2004), pp. 801–814 (cit. on p. 22).

[24] Szymon Rusinkiewicz and Marc Levoy. "QSplat: A Multiresolution Point Rendering System for Large Meshes". In: *Proceedings ACM SIGGRAPH*. Siggraph, 2000, pp. 343–352 (cit. on pp. 22, 30).

[25] S. Grottel, G. Reina, C. Dachsbacher, and T. Ertl. "Coherent Culling and Shading for Large Molecular Dynamics Visualization". In: *Computer Graphics Forum (Proceedings of EUROVIS 2010)*. Vol. 29. 2010, pp. 953–962 (cit. on p. 23).

[26] Carsten Dachsbacher, Christian Vogelgsang, and Marc Stamminger. “Sequential Point Trees”. In: *ACM Transactions on Graphics* 22.3 (2003), pp. 657–662 (cit. on p. 23).

[27] Renato Pajarola, Miguel Sainz, and Roberto Lario. “XSplat: External Memory Multiresolution Point Visualization”. In: *Proceedings IASTED International Conference on Visualization, Imaging and Image Processing*. VIIP, 2005, pp. 628–633 (cit. on p. 23).

[28] Michael Wimmer and Claus Scheiblauer. “Instant points: Fast rendering of unprocessed point clouds”. In: *Proc. SPBG*. 2006, pp. 129–137 (cit. on p. 23).

[29] Enrico Gobbetti and Fabio Marton. “Layered Point Clouds”. In: *Proceedings Eurographics/IEEE VGTC Symposium on Point-Based Graphics*. Eurographics, 2004, pp. 113–120 (cit. on pp. 23, 27, 28).

[30] Michael Wand, Alexander Berner, Martin Bokeloh, Arno Fleck, Mark Hoffmann, Philipp Jenke, Benjamin Maier, Dirk Staneker, and Andreas Schilling. “Interactive Editing of Large Point Clouds”. In: *Proceedings Eurographics/IEEE VGTC Symposium on Point-Based Graphics*. Eurographics, 2007, pp. 37–46 (cit. on p. 23).

[31] Fabio Bettio, Enrico Gobbetti, Fabio Martio, Alex Tinti, Emilio Merella, and Roberto Combet. “A Point-based System for Local and Remote Exploration of Dense 3D Scanned Models”. In: *Proceedings Eurographics Symposium on Virtual Reality, Archaeology and Cultural Heritage*. Eurographics, 2009, pp. 25–32 (cit. on p. 23).

[32] Mark Pauly, Markus Gross, and Leif P. Kobbelt. “Efficient Simplification of Point-Sampled Surfaces”. In: *Proceedings IEEE Visualization*. Computer Society Press, 2002, pp. 163–170 (cit. on p. 23).

[33] Prashant Goswami, Yanci Zhang, Renato Pajarola, and Enrico Gobbetti. “High Quality Interactive Rendering of Massive Point Models using Multi-way kd-Trees”. In: *18th Pacific Conference on Computer Graphics and Applications (PG)*. 2010, pp. 93–100. ISBN: 978-1-4244-8288-7 (cit. on p. 23).

[34] Paul Debevec, Tim Hawkins, Chris Tchou, Haarm-Pieter Duiker, Westley Sarokin, and Mark Sagar. “Acquiring the reflectance field of a human face”. In: *Proc. SIGGRAPH*. 2000, pp. 145–156 (cit. on p. 24).

[35] Hendrik P. A. Lensch, Jan Kautz, Michael Goesele, Wolfgang Heidrich, and Hans-Peter Seidel. “Image-based reconstruction of spatial appearance and geometric detail”. In: *ACM TOG* 22.2 (2003), pp. 234–257 (cit. on p. 24).

[36] Paul Debevec. “Rendering synthetic objects into real scenes: bridging traditional and image-based graphics with global illumination and high dynamic range photography”. In: *Proc. SIGGRAPH*. 1998, pp. 189–198 (cit. on p. 24).

[37] Massimiliano Corsini, Marco Callieri, and Paolo Cignoni. “Stereo Light Probe”. In: *Computer Graphics Forum* 27.2 (2008), pp. 291–300 (cit. on p. 24).

[38] Matteo Dellepiane, Marco Callieri, Massimiliano Corsini, Paolo Cignoni, and Roberto Scopigno. “Flash Lighting Space Sampling”. In: *Computer Vision/Computer Graphics Collaboration Techniques*. 2009, pp. 217–229 (cit. on pp. 24, 36).

[39] Matteo Dellepiane, Marco Callieri, Massimiliano Corsini, Paolo Cignoni, and Roberto Scopigno. “Improved color acquisition and mapping on 3D models via flash-based photography”. In: *ACM JOCCH* 2.4 (2010), Article 9 (cit. on pp. 24, 36).

[40] Seon Joo Kim, Hai Ting Lin, Zheng Lu, Sabine Suesstrunk, S. Lin, and M. S. Brown. "A New In-Camera Imaging Model for Color Computer Vision and Its Application". In: *IEEE Trans. PAMI* 34.12 (2012), pp. 2289–2302 (cit. on pp. 24, 36, 37).

[41] Marc Levoy, Kari Pulli, Brian Curless, Szymon Rusinkiewicz, David Koller, Lucas Pereira, Matt Ginzton, Sean Anderson, James Davis, Jeremy Ginsberg, et al. "The digital Michelangelo project: 3D scanning of large statues". In: *Proc. SIGGRAPH*. 2000, pp. 131–144 (cit. on pp. 24, 31, 36, 41).

[42] Marco Callieri, Paolo Cignoni, Massimiliano Corsini, and Roberto Scopigno. "Masked photo blending: Mapping dense photographic data set on high-resolution sampled 3D models". In: *Computers & Graphics* 32.4 (2008), pp. 464–473 (cit. on pp. 24, 31, 35).

[43] Ruggero Pintus, Enrico Gobbetti, and Marco Callieri. "Fast Low-Memory Seamless Photo Blending on Massive Point Clouds using a Streaming Framework". In: *ACM JOCCH* 4.2 (2011), Article 6 (cit. on pp. 24, 31, 35, 40, 49).

[44] Fabio Bettio, Enrico Gobbetti, Emilio Merella, and Ruggero Pintus. "Improving the digitization of shape and color of 3D artworks in a cluttered environment". In: *Proc. Digital Heritage*. Oct. 2013, pp. 23–30 (cit. on p. 24).

[45] Michael Kazhdan and Hugues Hoppe. "Screened Poisson surface reconstruction". In: *ACM Trans. Graph* 32.1 (2013) (cit. on pp. 24, 40, 43).

[46] Chunlin Wu, Jiansong Deng, and Falai Chen. "Diffusion equations over arbitrary triangulated surfaces for filtering and texture applications". In: *Visualization and Computer Graphics, IEEE Transactions on* 14.3 (2008), pp. 666–679 (cit. on pp. 24, 35, 43).

[47] Marco Callieri, Matteo Dellepiane, Paolo Cignoni, and Roberto Scopigno. "Digital Imaging for Cultural Heritage Preservation: Analysis, Restoration, and Reconstruction of Ancient Artworks". In: CRC Press, 2011. Chap. Processing sampled 3D data: reconstruction and visualization technologies, pp. 69–99 (cit. on p. 25).

[48] Antonio Adan and Daniel Huber. "3D reconstruction of interior wall surfaces under occlusion and clutter". In: *Proc. 3DIMPVT*. 2011, pp. 275–281 (cit. on p. 25).

[49] Florent Lafarge and Clément Mallet. "Creating large-scale city models from 3D point clouds: a robust approach with hybrid representation". In: *IJCV* 99.1 (2012), pp. 69–85 (cit. on p. 25).

[50] Mohamed Farouk, Ibrahim El-Rifai, Shady El-Tayar, Hisham El-Shishiny, Mohamed Hosny, Mohamed El-Rayes, Jose Gomes, Frank Giordano, Holly E Rushmeier, Fausto Bernardini, et al. "Scanning and Processing 3D Objects for Web Display". In: *3DIM*. 2003, pp. 310–317 (cit. on pp. 25, 34).

[51] Hui Zhang, Jason E Fritts, and Sally A Goldman. "Image segmentation evaluation: A survey of unsupervised methods". In: *Computer Vision and Image Understanding* 110.2 (2008), pp. 260–280 (cit. on p. 25).

[52] Kevin McGuinness and Noel E O'Connor. "A comparative evaluation of interactive segmentation algorithms". In: *Pattern Recognition* 43.2 (2010), pp. 434–444 (cit. on p. 25).

[53] Adobe Systems Inc. *Adobe Photoshop User Guide*. 2002 (cit. on p. 25).

[54] Eric N Mortensen and William A Barrett. "Toboggan-based intelligent scissors with a four-parameter edge model". In: *Proc. CVPR*. Vol. 2. 1999 (cit. on p. 25).

[55] Michael Kass, Andrew Witkin, and Demetri Terzopoulos. "Snakes: Active contour models". In: *IJCV* 1.4 (1988), pp. 321–331 (cit. on p. 25).

[56] Yung-Yu Chuang, Brian Curless, David H Salesin, and Richard Szeliski. "A Bayesian approach to digital matting". In: *Proc. CVPR*. 2001, pp. 264–271 (cit. on pp. 25, 39).

[57] Yuri Y Boykov and M-P Jolly. "Interactive graph cuts for optimal boundary & region segmentation of objects in N–D images". In: *Proc. ICCV*. Vol. 1. 2001, pp. 105–112 (cit. on p. 25).

[58] Yun Zeng, Dimitris Samaras, Wei Chen, and Qunsheng Peng. "Topology cuts: A novel min-cut/max-flow algorithm for topology preserving segmentation in N–D images". In: *Computer vision and image understanding* 112.1 (2008), pp. 81–90 (cit. on p. 25).

[59] Ning Xu, Narendra Ahuja, and Ravi Bansal. "Object segmentation using graph cuts based active contours". In: *Computer Vision and Image Understanding* 107.3 (2007), pp. 210–224 (cit. on p. 25).

[60] Carsten Rother, Vladimir Kolmogorov, and Andrew Blake. "Grabcut: Interactive foreground extraction using iterated graph cuts". In: *ACM TOG*. Vol. 23. 3. 2004, pp. 309–314 (cit. on pp. 25, 37, 39).

[61] Enrico Gobbetti and Fabio Marton. "Layered Point Clouds: A Simple and Efficient Multiresolution Structure for Distributing and Rendering Gigantic Point-Sampled Models". In: *Computers & Graphics* 28.1 (Feb. 2004), pp. 815–826 (cit. on p. 27).

[62] Fausto Bernardini and Holly Rushmeier. "The 3D model acquisition pipeline". In: *Computer Graphics Forum*. Vol. 21. 2. 2002, pp. 149–172 (cit. on p. 31).

[63] Paolo Pingi, Andrea Fasano, Paolo Cignoni, Claudio Montani, and Roberto Scopigno. "Exploiting the scanning sequence for automatic registration of large sets of range maps". In: *Computer Graphics Forum* 24.3 (2005), pp. 517–526 (cit. on p. 31).

[64] Michael Kazhdan, Matthew Bolitho, and Hugues Hoppe. "Poisson surface reconstruction". In: *Proc. SGP*. 2006, pp. 61–70 (cit. on pp. 31, 34, 40).

[65] Josiah Manson, Guergana Petrova, and Scott Schaefer. "Streaming surface reconstruction using wavelets". In: *Computer Graphics Forum*. Vol. 27. 5. 2008, pp. 1411–1420 (cit. on pp. 31, 40).

[66] Gianmauro Cuccuru, Enrico Gobbetti, Fabio Marton, Renato Pajarola, and Ruggero Pintus. "Fast low-memory streaming MLS reconstruction of point-sampled surfaces". In: *Graphics Interface*. May 2009, pp. 15–22 (cit. on pp. 31, 40).

[67] Fatih Calakli and Gabriel Taubin. "SSD: Smooth signed distance surface reconstruction". In: *Computer Graphics Forum*. Vol. 30. 7. 2011, pp. 1993–2002 (cit. on p. 31).

[68] Ruggero Pintus, Enrico Gobbetti, and Roberto Combet. "Fast and Robust Semi-Automatic Registration of Photographs to 3D Geometry". In: *Proc. VAST*. 2011, pp. 9–16 (cit. on pp. 31, 40).

[69] M Corsini, M Dellepiane, F Ganovelli, R Gherardi, A Fusiello, and R Scopigno. "Fully Automatic Registration of Image Sets on Approximate Geometry". In: *IJCV* (2012), pp. 1–21 (cit. on p. 31).

[70] Ruggero Pintus, Enrico Gobbetti, and Marco Callieri. “A Streaming Framework for Seamless Detailed Photo Blending on Massive Point Clouds”. In: *Proc. Eurographics Area Papers*. 2011, pp. 25–32 (cit. on p. 31).

[71] Fabio Remondino. “Heritage recording and 3D modeling with photogrammetry and 3D scanning”. In: *Remote Sensing* 3.6 (2011), pp. 1104–1138 (cit. on pp. 31, 32).

[72] Anestis Koutsoudis, Blaž Vidmar, George Ioannakis, Fotis Arnaoutoglou, George Pavlidis, and Christodoulos Chamzas. “Multi-image 3D reconstruction data evaluation”. In: *Journal of Cultural Heritage* 15.1 (2014), pp. 73 –79. ISSN: 1296-2074. DOI: `http://dx.doi.org/10.1016/j.culher.2012.12.003` (cit. on p. 32).

[73] James Davis, Stephen R Marschner, Matt Garr, and Marc Levoy. “Filling holes in complex surfaces using volumetric diffusion”. In: *Proc. 3DPVT*. 2002, pp. 428–441 (cit. on p. 35).

[74] Ruggero Pintus and Enrico Gobbetti. “A Fast and Robust Framework for Semi-Automatic and Automatic Registration of Photographs to 3D Geometry”. In: *ACM Journal on Computing and Cultural Heritage* (2014). To appear (cit. on p. 35).

[75] David Friedrich, Johannes Brauers, André A Bell, and Til Aach. “Towards fully automated precise measurement of camera transfer functions”. In: *IEEE Southwest Symposium on Image Analysis & Interpretation*. IEEE. 2010, pp. 149–152 (cit. on p. 36).

[76] Mark A Ruzon and Carlo Tomasi. “Alpha estimation in natural images”. In: *Proc. CVPR*. 2000, pp. 18–25 (cit. on p. 39).

[77] Michael Oren and Shree K Nayar. “Generalization of Lambert’s reflectance model”. In: *Proc. SIGGRAPH*. ACM. 1994, pp. 239–246 (cit. on p. 42).

[78] Noah Snavely, Steven M. Seitz, and Richard Szeliski. “Modeling the World from Internet Photo Collections”. In: *IJCV* 80.2 (2008) (cit. on p. 44).

[79] Javier Rey Neira and Supervisors: Alberto Jaspe villanueva and Emilio J.Padrón. “Sistema cliente-servidor para la visualización de nubes de puntos con WebGL”. MA thesis. University of A Coruña, 2013 (cit. on p. 51).

[80] Ingo Wald, William R Mark, Johannes Günther, Solomon Boulos, Thiago Ize, Warren Hunt, Steven G Parker, and Peter Shirley. “State of the Art in Ray Tracing Animated Scenes”. In: *Computer Graphics Forum* 28.6 (2009), pp. 1691–1722 (cit. on p. 54).

[81] M. Hapala and V. Havran. “Review: Kd-tree Traversal Algorithms for Ray Tracing”. In: *Computer Graphics Forum* 30.1 (2011), pp. 199–213 (cit. on p. 54).

[82] Jeffrey Goldsmith and John Salmon. “Automatic Creation of Object Hierarchies for Ray Tracing”. In: *IEEE Computer Graphics and Applications* 7.5 (1987), pp. 14–20 (cit. on pp. 54, 62).

[83] John Amanatides. “Ray Tracing with Cones”. In: *SIGGRAPH Computer Graphics* 18.3 (1984), pp. 129–135 (cit. on p. 55).

[84] Paul S. Heckbert and Pat Hanrahan. “Beam Tracing Polygonal Objects”. In: *SIGGRAPH Computer Graphics* 18.3 (1984), pp. 119–127. ISSN: 0097-8930 (cit. on p. 55).

[85] Samuli Laine, Samuel Siltanen, Tapio Lokki, and Lauri Savioja. “Accelerated Beam Tracing Algorithm”. In: *Applied Acoustics* 70.1 (2009), pp. 172–181 (cit. on p. 55).

[86] Karthik Ramani, Christiaan P. Gribble, and Al Davis. “StreamRay: a stream filtering architecture for coherent ray tracing”. In: *ACM SIGPLAN Notices* 44.3 (2009), pp. 325–336 (cit. on p. 55).

[87] Benjamin Mora. “Naive ray-tracing: A divide-and-conquer approach”. In: *Transactions on Graphics* 30.5 (2011), p. 117 (cit. on p. 55).

[88] Solomon Boulos, Ingo Wald, and Carsten Benthin. “Adaptive ray packet reordering”. In: *IEEE Symposium on Interactive Ray Tracing*. 2008, pp. 131–138 (cit. on p. 55).

[89] Timo Aila and Samuli Laine. “Understanding the Efficiency of Ray Traversal on GPUs”. In: *Proc. High-Performance Graphics*. 2009, pp. 145–149 (cit. on pp. 56, 67).

[90] Timo Aila and Tero Karras. “Architecture Considerations for Tracing Incoherent Rays”. In: *Proc. High-Performance Graphics*. 2010, pp. 113–122 (cit. on p. 56).

[91] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. “OptiX: a general purpose ray tracing engine”. In: *Transactions on Graphics* 29 (4 2010), 66:1–66:13. ISSN: 0730-0301 (cit. on p. 56).

[92] Timo Aila, Samuli Laine, and Tero Karras. *Understanding the Efficiency of Ray Traversal on GPUs – Kepler and Fermi Addendum*. Tech. rep. NVR-2012-02. NVIDIA, June 2012 (cit. on pp. 56, 76, 77).

[93] Nathan A. Carr, Jesse D. Hall, and John C. Hart. “The Ray Engine”. In: *Proc. ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. Saarbrucken, Germany, 2002, pp. 37–46. ISBN: 1-58113-580-7 (cit. on pp. 56, 61, 67).

[94] David Roger, Ulf Assarsson, and Nicolas Holzschuch. “Whitted Ray-Tracing for Dynamic Scenes using a Ray-Space Hierarchy on the GPU”. In: *Proc. Eurographics Symposium on Rendering*. 2007, pp. 99–110 (cit. on p. 56).

[95] Laszlo Szirmay-Kalos, Barnabas Aszodi, Istvan Lazanyi, and Matyas Premecz. “Approximate Ray-Tracing on the GPU with Distance Impostors”. In: *Computer Graphics Forum* 24.3 (2005), pp. 695–704 (cit. on p. 56).

[96] David Roger and Nicolas Holzschuch. “Accurate Specular Reflections in Real-Time”. In: *Computer Graphics Forum* 25.3 (2006), pp. 293–302 (cit. on p. 56).

[97] Jan Novák and Carsten Dachsbacher. “Rasterized Bounding Volume Hierarchies”. In: *Computer Graphics Forum* 31.2 (2012), pp. 403–412 (cit. on p. 56).

[98] Tomás Davidovic, Thomas Engelhardt, Iliyan Georgiev, Philipp Slusallek, and Carsten Dachsbacher. “3D rasterization: a bridge between rasterization and ray casting”. In: *Proc. Graphics Interface*. 2012, pp. 201–208 (cit. on p. 56).

[99] Tobias Zirr, Hauke Rehfeld, and Carsten Dachsbacher. “Object-order ray tracing for fully dynamic scenes”. In: *GPU Pro 5*. A K Peters/CRC Press, 2014 (cit. on p. 56).

[100] Wei Hu, Yangyu Huang, Fan Zhang, Guodong Yuan, and Wei Li. “Ray Tracing via GPU Rasterization”. In: *Visual Computer* 30.6-8 (June 2014), pp. 697–706. ISSN: 0178-2789 (cit. on p. 56).

[101] Enrico Gobbetti, Dave Kasik, and Sung-eui Yoon. “Technical strategies for massive model visualization”. In: *Proc. ACM Symposium on Solid and Physical Modeling*. 2008, pp. 405–415 (cit. on p. 57).

[102] Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan. “Rendering Complex Scenes with Memory-Coherent Ray Tracing”. In: *Proc. SIGGRAPH*. 1997, pp. 101–108 (cit. on p. 57).

[103] Bochang Moon, Yongyoung Byun, Tae-Joon Kim, Pio Claudio, Hye-Sun Kim, Yun-Ji Ban, Seung Woo Nam, and Sung-Eui Yoon. “Cache-oblivious ray reordering”. In: *Transactions on Graphics* 29.3 (2010), pp. 1–10 (cit. on pp. 57, 73).

[104] Per H. Christensen, David M. Laur, Julia Fong, Wayne L. Wooten, and Dana Batali. “Ray Differentials and Multiresolution Geometry Caching for Distribution Ray Tracing in Complex Scenes”. In: *Computer Graphics Forum* 22.3 (2003), pp. 543–543 (cit. on p. 57).

[105] Christian Lauterbach, Sung-eui Yoon, Ming Tang, and Dinesh Manocha. “ReduceM: Interactive and Memory Efficient Ray Tracing of Large Models”. In: *Computer Graphics Forum* 27.4 (2008), pp. 1313–1321 (cit. on p. 57).

[106] Attila T. Áfra. “Interactive Ray Tracing of Large Models Using Voxel Hierarchies”. In: *Computer Graphics Forum* 31.1 (2012), pp. 75–88 (cit. on p. 57).

[107] Brian Budge, Tony Bernardin, Jeff A. Stuart, Shubhabrata Sengupta, Kenneth I. Joy, and John D. Owens. “Out-of-Core Data Management for Path Tracing on Hybrid Resources”. In: *Computer Graphics Forum* 28.2 (2009), pp. 385–396 (cit. on p. 57).

[108] Tae-Joon Kim, Xin Sun, and Sung-Eui Yoon. “T-ReX: Interactive Global Illumination of Massive Models on Heterogeneous Computing Resources”. In: *Transactions on Visualization and Computer Graphics* 20.3 (2014), pp. 481–494 (cit. on p. 57).

[109] Jacopo Pantaleoni, Luca Fascione, Martin Hill, and Timo Aila. “PantaRay: Fast Ray-Traced Occlusion Caching of Massive Scenes”. In: *Transactions on Graphics* 29 (4 2010), 37:1–37:10. ISSN: 0730-0301 (cit. on p. 57).

[110] Kirill Garanzha, Alexander Bely, Simon Premoze, and Vladimir Galaktionov. “Out-Of-Core GPU Ray Tracing of Complex Scenes”. In: *SIGGRAPH Talks*. 2011, 21:1–21:1 (cit. on p. 57).

[111] Paolo Cignoni, Fabio Ganovelli, Enrico Gobbetti, Fabio Marton, Federico Ponchio, and Roberto Scopigno. “Adaptive tetrapuzzles: efficient out-of-core construction and visualization of gigantic multiresolution polygonal models”. In: *Transactions on Graphics* 23.3 (Aug. 2004), pp. 796–803 (cit. on p. 57).

[112] Louis Borgeat, Guy Godin, François Blais, Philippe Massicotte, and Christian Lahanier. “GoLD: interactive display of huge colored and textured models”. In: *Transactions on Graphics* 24.3 (2005), pp. 869–877 (cit. on p. 57).

[113] Enrico Gobbetti and Fabio Marton. “Far Voxels – A Multiresolution Framework for Interactive Rendering of Huge Complex 3D Models on Commodity Graphics Platforms”. In: *ACM Trans. Graph.* 24.3 (2005), pp. 878–885 (cit. on p. 57).

[114] Dirk Staneker, Dirk Bartz, and Michael Meissner. “Improving Occlusion Query Efficiency with Occupancy Maps”. In: *Proc. Symposium on Parallel and Large-Data Visualization and Graphics*. 2003, pp. 15– (cit. on p. 57).

[115] Jiří Bittner, Michael Wimmer, Harald Piringer, and Werner Purgathofer. “Coherent Hierarchical Culling: Hardware Occlusion Queries Made Useful”. In: *Computer Graphics Forum* 23.3 (2004), pp. 615–624. ISSN: 0167-7055 (cit. on pp. 57, 60, 63).

[116] M. Guthe, Á. Balázs, and R. Klein. “Near Optimal Hierarchical Culling: Performance Driven Use of Hardware Occlusion Queries”. In: *Proc. Eurographics Symposium on Rendering*. 2006 (cit. on p. 57).

[117] Jiří Bittner, Oliver Mattausch, Ari Silvennoinen, and Michael Wimmer. “Shadow Caster Culling for Efficient Shadow Mapping”. In: *Proc. Symposium on Interactive 3D Graphics and Games*. San Francisco, Feb. 2011, pp. 81–88 (cit. on p. 57).

[118] Michael Deering, Stephanie Winner, Bic Schediwy, Chris Duffy, and Neil Hunt. “The Triangle Processor and Normal Vector Shader: A VLSI System for High Performance Graphics”. In: *SIGGRAPH ’88 Proceedings* 22.4 (Aug. 1988), pp. 21–30 (cit. on p. 57).

[119] Takafumi Saito and Tokiichiro Takahashi. “Comprehensible Rendering of 3-D Shapes”. In: *Computer Graphics* 24.4 (1990), pp. 197–206 (cit. on p. 57).

[120] Ola Olsson, Markus Billeter, and Ulf Assarsson. “Clustered Deferred and Forward Shading”. In: *Proc. Conference on High Performance Graphics*. Paris, France, 2012 (cit. on p. 57).

[121] Martin Mittring. “Finding next gen: Cryengine 2”. In: *SIGGRAPH Courses*. ACM. 2007, pp. 97–121 (cit. on p. 69).

[122] Kirill Garanzha and Charles Loop. “Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing”. In: *Computer Graphics Forum* 29.2 (2010), pp. 289–298 (cit. on p. 73).

[123] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. “Procedural Modeling of Buildings”. In: *Transactions on Graphics* 25.3 (July 2006), pp. 614–623 (cit. on p. 74).

[124] Arcot J Preetham, Peter Shirley, and Brian Smits. “A practical analytic model for daylight”. In: *Proc. Conference on Computer graphics and Interactive Techniques*. 1999, pp. 91–100 (cit. on p. 74).

[125] Jirí Bittner, Michal Hapala, and Vlastimil Havran. “Fast Insertion-Based Optimization of Bounding Volume Hierarchies”. In: *Computer Graphics Forum* 32.1 (2013), pp. 85–100 (cit. on p. 77).

[126] Ingo Wald, Aaron Knoll, Gregory P Johnson, Will Usher, Valerio Pascucci, and Michael E Papka. “CPU ray tracing large particle data with balanced Pkd trees”. In: *2015 IEEE Scientific Visualization Conference (SciVis)*. IEEE. 2015, pp. 57–64 (cit. on p. 83).

[127] Kostas Vardis, Andreas-Alexandros Vasilakis, and Georgios Papaioannou. “DIRT: deferred image-based ray tracing”. In: *High Performance Graphics*. 2016, pp. 63–73 (cit. on p. 83).

[128] Rasmus Barringer, Magnus Andersson, and Tomas Akenine-Möller. “Ray Accelerator: Efficient and Flexible Ray Tracing on a Heterogeneous Architecture”. In: *Computer Graphics Forum* 36.8 (2017), pp. 166–177 (cit. on p. 83).

[129] Timothy R Kol, Pablo Bauszat, Sungkil Lee, and Elmar Eisemann. “MegaViews: Scalable Many-View Rendering With Concurrent Scene-View Hierarchy Traversal”. In: *Computer Graphics Forum* (2018) (cit. on pp. 83, 117, 148).

[130] Warren Hunt, Michael Mara, and Alex Nankervis. “Hierarchical Visibility for Virtual Reality”. In: *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 1.1 (2018), p. 8 (cit. on p. 83).

[131] Samuli Laine and Tero Karras. “Efficient sparse voxel octrees”. In: *IEEE Trans. Vis. Comput. Graph* 17.8 (2011), pp. 1048–1059 (cit. on pp. 84, 87, 89, 99, 101, 106, 110, 144).

[132] Marcos Balsa Rodriguez, Enrico Gobbetti, José Antonio Iglesias Guitián, Maxim Makhinya, Fabio Marton, Renato Pajarola, and Susanne Suter. “State-of-the-art in Compressed GPU-Based Direct Volume Rendering”. In: *Computer Graphics Forum* 33.6 (2014), pp. 77–100 (cit. on pp. 85, 87, 144).

[133] Erik Sintorn, Viktor Kämpe, Ola Olsson, and Ulf Assarsson. “Compact Precomputed Voxelized Shadows”. In: *ACM Trans. Graph.* 33.4 (July 2014), 150:1–150:8 (cit. on pp. 87, 88, 113, 118, 149).

[134] Viktor Kämpe, Erik Sintorn, and Ulf Assarsson. “Fast, memory-efficient construction of voxelized shadows”. In: *Proc. ACM I3D*. 2015, pp. 25–30 (cit. on pp. 87, 88, 113, 118, 149).

[135] Bas Dado, Timothy R. Kol, Pablo Bauszat, Jean-Marc Thiery, and Elmar Eisemann. “Geometry and Attribute Compression for Voxel Scenes”. In: *Computer Graphics Forum (Proc. Eurographics)* 35.2 (May 2016), pp. 397–407 (cit. on pp. 87–89, 102, 113, 114, 118, 149).

[136] Enrico Gobbetti, Fabio Marton, and José Antonio Iglesias Guitián. “A single-pass GPU ray casting framework for interactive out-of-core rendering of massive volumetric datasets”. In: *The Visual Computer* 24.7-9 (2008), pp. 797–806 (cit. on p. 87).

[137] Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre, and Elmar Eisemann. “Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering”. In: *Proc. ACM I3D*. 2009, pp. 15–22 (cit. on pp. 87, 88, 98).

[138] Marco Agus, Enrico Gobbetti, José Antonio Iglesias Guitián, and Fabio Marton. “Split-Voxel: A Simple Discontinuity-Preserving Voxel Representation for Volume Rendering”. In: *Proc. Volume Graphics*. 2010, pp. 21–28 (cit. on p. 88).

[139] Cyril Crassin, Fabrice Neyret, Miguel Sainz, Simon Green, and Elmar Eisemann. “Interactive indirect illumination using voxel cone tracing”. In: *Computer Graphics Forum*. Vol. 30. 7. 2011, pp. 1921–1930 (cit. on p. 88).

[140] Robert E Webber and Michael B Dillencourt. “Compressing quadtrees via common subtree merging”. In: *Pattern recognition letters* 9.3 (1989), pp. 193–200 (cit. on p. 88).

[141] Eric Parker and Tushar Udeshi. “Exploiting Self-similarity in Geometry for Voxel Based Solid Modeling”. In: *Proc. ACM Solid Modeling*. 2003, pp. 157–166 (cit. on p. 88).

[142] Rama Karl Hoetzlein. “GVDB: Raytracing Sparse Voxel Database Structures on the GPU”. In: *Eurographics/ ACM SIGGRAPH Symposium on High Performance Graphics*. Ed. by Ulf Assarsson and Warren Hunt. The Eurographics Association, 2016. ISBN: 978-3-03868-008-6. DOI: 10.2312/hpg.20161197 (cit. on pp. 88, 114).

[143] Viktor Kämpe, Sverker Rasmuson, Markus Billeter, Erik Sintorn, and Ulf Assarsson. “Exploiting Coherence in Time-varying Voxel Data”. In: *Proc. ACM I3D*. 2016, pp. 15–21. DOI: 10.1145/2856400.2856413 (cit. on p. 88).

[144] Erik Hubo, Tom Mertens, Tom Haber, and Philippe Bekaert. “Self-similarity Based Compression of Point Set Surfaces with Application to Ray Tracing”. In: *Comput. Graph.* 32.2 (Apr. 2008), pp. 221–234 (cit. on p. 88).

[145] Ruwen Schnabel and Reinhard Klein. “Octree-based Point-Cloud Compression.” In: *Proc. SPBG*. 2006, pp. 111–120 (cit. on pp. 89, 106).

[146] Sylvain Lefebvre and Hugues Hoppe. “Compressed random-access trees for spatially coherent data”. In: *Proc. EGSR*. 2007, pp. 339–349 (cit. on p. 89).

[147] Johanna Beyer, Markus Hadwiger, and Hanspeter Pfister. “State-of-the-Art in GPU-Based Large-Scale Volume Visualization”. In: *Computer Graphics Forum*. In press. 2015 (cit. on pp. 89, 113).

[148] Cyril Crassin and Simon Green. “Octree-based sparse voxelization using the GPU hardware rasterizer”. In: *OpenGL Insights* (2012), pp. 303–318 (cit. on pp. 89, 105).

[149] Tim Foley and Jeremy Sugerman. “KD-tree Acceleration Structures for a GPU Raytracer”. In: *Proc. ACM Graphics Hardware*. 2005, pp. 15–22 (cit. on p. 98).

[150] Daniel Reiter Horn, Jeremy Sugerman, Mike Houston, and Pat Hanrahan. “Interactive k-D tree GPU raytracing”. In: *Proc. ACM I3D*. 2007, pp. 167–174 (cit. on p. 99).

[151] Martin Pätzold and Andreas Kolb. “Grid-free out-of-core voxelization to sparse voxel octrees on GPU”. In: *Proc. High-Performance Graphics*. 2015, pp. 95–103 (cit. on p. 105).

[152] Dan Dolonius, Erik Sintorn, Viktor Kampe, and Ulf Assarsson. “Compressing Color Data for Voxelized Surface Geometry.” In: *IEEE transactions on visualization and computer graphics* (2017) (cit. on pp. 114, 118, 149).

[153] Weiwei Duan, Jianxin Luo, Guiqiang Ni, Bin Tang, Qi Hu, and Yi Gao. “Exclusive grouped spatial hashing”. In: *Computers & Graphics* 70 (2018), pp. 71–79 (cit. on p. 114).

A Sinopsis (thesis summary in Spanish)

La disponibilidad y uso de datos tridimensionales crece a un ritmo enormemente acelerado gracias a la rápida evolución de las técnicas de adquisición y creación de modelos 3D, que a su vez son cada vez más detallados. Al mismo tiempo, muchos dominios no sólo se limitan a estudiar estos datos de manera *off-line*, sino que requieren sistemas interactivos de exploración y visualización de estos modelos con un cierto realismo. Esto plantea nuevos retos para conseguir transformar eficientemente cantidades masivas de datos 3D en tipos de datos renderizables, y representarlos eficazmente de una manera visualmente convincente, a altas frecuencias de *frames* por segundo. Esta tesis presenta nuevos métodos escalables para mejorar creación de modelos masivos capturados de la realidad y técnicas novedosas que mejoren la escalabilidad de renderizado avanzado, mediante procesamiento por lotes adaptable y adecuado para las GPUs, o mediante compresión de representaciones voxelizadas de las escenas. En este anexo se ofrece una sinopsis de la tesis en lengua española, centrándose en las motivaciones, objetivos, y logros conseguidos, junto a las descripciones resumidas de los diversos métodos propuestos.

A.1 Contexto, motivación e hipótesis

HOY en día se generan continuamente una inmensa cantidad datos tridimensionales de enorme escala y densidad, provenientes de diversos tipos de sensores, sistemas de escaneado, modelado 3D o simulaciones numéricas en una gran variedad de disciplinas y sus diversos campos de aplicación. Estas tecnologías, así como la creciente digitalización de las metodologías de trabajo en prácticamente cualquier dominio, producen modelos 3D cada vez más complejos y con tamaños siempre mayores. Una de los tipos de dato más utilizados son los *modelos de superficie 3D*, que describen la forma y color de objetos diseñados, simulados o reales. La figura Fig. 1.2 en el capítulo introductorio de la tesis muestra ejemplos de este tipo de datos utilizados en diversas disciplinas. Estos modelos han sido generados desde distintas fuentes o como resultado de procesos computacionales, dependiendo de su función. Una clasificación informal de los tipos de modelos más utilizados podría ser de esta manera:

- **Modelos CAD**. Las técnicas de Diseño Asistido por Computador o CAD son utilizadas comúnmente por ingenieros, arquitectos, diseñadores, etc. Estas escenas a menudo representan proyectos de gran tamaño, y como suelen tener un objetivo técnico, presentan gran cantidad de detalles de enorme precisión. La manera en que normalmente se visualizan estos datos es utilizando modelos de superficie, compuestos por mallas de triángulos derivadas de los bordes externos de los objetos CAD, ya sean estos sólidos o paramétricos. Estos modelos de superficie llegan a ser verdaderamente grandes; por ejemplo el Boeing 777 de la figura Fig. 1.1 contiene entorno a 350 millones de triángulos.
- **Modelos adquiridos**. El uso de scanners como LIDAR u otras técnicas como la fotogrametría se han extendido ampliamente en estos últimos años. Éstas permiten obtener geometría de alta precisión y información colorimetrétrica del mundo real. Al mismo tiempo que el costo de estas técnicas se reduce drásticamente, aumenta su grado de resolución. Estas tecnologías se usan en una gran variedad de sectores y a diversas escalas, desde ingeniería con, por ejemplo, escaneos aéreos de vastas extensiones de terreno o entornos urbanos, hasta la conservación cultural, donde estas metodologías se han vuelto de uso común para la digitalización o captura de objetos artísticos o arqueológicos como pinturas o esculturas. Normalmente la resolución de los objetos adquiridos incrementa cuando se aumenta el número de muestras, lo que genera enormes bases de datos, difíciles de gestionar, procesar, o peor aún, explorar visualmente. La representación más básica de esos modelos es la nube de puntos, con conjuntos de datos que pueden contener decenas, centenares o miles de millones de puntos. Por ejemplo, el modelo del David de Michelangelo de la figura Fig. 1.2 contiene medio millar de millones de puntos.
- **Modelos resultado de computaciones**, Muchos modelos 3D se obtienen como resultado de procesos numéricos, fundamentalmente en sectores científicos como la química, biología o astrofísica. A veces son usados como soporte para cómputos secundarios, como para realizar iluminación global, sombras, o el cálculo de colisiones, donde algoritmos similares a los utilizados normalmente para visualización se aprovechan para efectos o comportamientos específicos. Estos pueden ser de diversos tipos, como por ejemplo campos volumétricos escalares; pero también en este ámbito los grandes modelos de superficies son muy utilizados.
- **Modelos de malla diseñados o construidos**, ampliamente usados en simulaciones, diseño de interiores, entornos de realidad virtual, o la industria de entretenimiento, como películas o videojuegos. Este tipo de modelos los gen-

eran normalmente artistas o diseñadores con herramientas de autor. Pueden tener todo tipo de tamaños, de muy pequeños a enormes, dependiendo de su utilidad. Pero es común encontrar modelos con grandes cantidades de detalles, como los ejemplos de la Fig. 1.2.

Una gran variedad de campos, como se ha mostrado, producen lo que podríamos denominar como *modelos masivos*. Como se señala en la referencia estándar sobre renderizado de este tipo de modelos [3], éstos se cumplen tres preceptos: por una parte, la representación de sus superficies requiere millones o incluso miles de millones de primitivas geométricas, que se traducen en cantidades ingentes de memoria que llegan fácilmente a los centenares de gigabytes o incluso terabytes. Por otra parte, estos datos digitales que representan los modelos, describen altísimos niveles de detalle que normalmente no son percibidos por el ojo humano hasta que son magnificados, mientras que la forma del modelo completa sólo es percibida cuando se aleja mucho el punto de vista. Por último, la gestión de estos datos sobrepasa la capacidad normal de las técnicas y hardware convencionales de procesado.

Aunque una variedad de análisis diversos pueden ser efectuados *off-line* en tales modelos, muchos de sus usos requieren su inspección interactiva por operadores humanos. La visualización 3D en tiempo real de estos conjuntos de datos es, sin embargo, particularmente desafiante, dada la necesidad inherente de generar imágenes visualmente ricas en altas frecuencias y con baja latencia, en respuesta al movimiento del punto de vista efectuada por el usuario. De hecho, para que una aplicación de visualización sea interactiva debe, por un lado, generar imágenes a una velocidad lo suficientemente alta como para proporcionar la ilusión de animación al sistema de percepción humana. Esto típicamente significa mantener al menos un ritmo de actualización de unos 10Hz [3]. Además, la aplicación debe responder con una latencia lo suficientemente baja como para proporcionar la impresión de una retroalimentación instantánea, que es necesaria para el control interactivo. Este típicamente significa usar sólo unos pocos milisegundos para responder a una acción del usuario, como un click de ratón o un cambio en la dirección de movimiento. Las imágenes generadas a alta frecuencia y con baja latencia deben ser de una calidad lo suficientemente alta como para ofrecer una información visual convincente, lo que significa, para muchas aplicaciones, la necesidad de calcular sombras e iluminación no local/global.

A pesar del continuo aumento de potencia de procesado de los ordenadores y unidades gráficas (GPUs), es evidente que no se puede simplemente esperar que el uso de técnicas de fuerza bruta en hardware más potente sea suficiente para alcanzar el objetivo de la inspección interactiva de datos masivos. Esto no es sólo debido a que las mejoras de hardware también conducen a la generación de conjuntos de datos cada vez más complejos, sino también porque el ancho de banda de la memoria y la velocidad de acceso a los datos crecen a un ritmo mucho más lento que la potencia de procesamiento, convirtiéndose así en los principales cuellos de botella para manejar estas grandes cantidades de datos (ver Fig. 1.3), especialmente en el caso de iluminaciones complejas no locales, que deben combinar, por píxel, la contribución de muchas partes de la escena que afectan a sombras y/o interreflexiones, y que hacen aumentar dramáticamente los requisitos de ancho de banda.

Por estas razones fundamentales, muchos esfuerzos de investigación se han centrado en la problema de idear métodos inteligentes para renderizar modelos masivos en hardware gráfico (véase la investigación clásica de Yoon et al. [3], y la más reciente sobre soluciones de ray-tracing por Deng et al. [4]). En general, las principales técnicas empleadas en todas las soluciones se esfuerzan por reducir la cantidad de datos que necesitan ser almacenados o procesados en un momento dado. Pueden ser clasificadas como sigue:

- **Técnicas de filtrado de datos**. Dado que los modelos masivos son demasiado grandes para ser procesados y requieren demasiada computación, muchos métodos intentan dividirlo muy eficientemente en *datasets* reducidos en los que realizar el cálculo de renderizado sea lo suficientemente rápido como para cumplir con las restricciones de tiempo y no reducir la calidad. Este objetivo se logra empleando estructuras de datos y algoritmos apropiados para la visibilidad o la selección de detalles, que eliminan rápidamente porciones de la escena que no contribuyen a la imagen final (véase el estado del arte de la sección Sec. 4.2).
- **Técnicas *out-of-core* adaptativas**. Dado que los modelos masivos no caben totalmente en la memoria gráfica, y a menudo incluso en la memoria principal, sus métodos de renderizado están diseñados para trabajar en estructuras fuera del núcleo o *out-of-core*, cargando datos bajo demanda. Dados los altos costes de entrada/salida, normalmente se emplean métodos adaptativos que tratan de hacer un acceso coherente, con el objetivo de reducir el número de fallos de caché y, por tanto, el tiempo de acceso a los datos (véase también los estados del arte en Sec. 3.2 y Sec. 4.2).
- **Técnicas de compresión de datos**. Dado que la cantidad limitada de memoria impone límites de tamaño en el modelo más grande que se puede gestionar den-

tro del procesador (*in-core*) y, al mismo tiempo, el acceso a grandes cantidades de datos también es muy costoso en términos de tiempo, muchos métodos reducen los requisitos de tamaño de los datos con técnicas de compresión. Dado que muchos algoritmos complejos, como aquellos de trazado de rayos, requieren acceso aleatorio a las estructuras de datos espaciales, así como a los datos de escena, el formato comprimido está diseñado para soportar una descompresión aleatoria muy eficiente y transitoria (véase el estado del arte en Sec. 5.2).

Se han propuesto muchas soluciones que combinan estos ingredientes en complejos y potentes sistemas de render. Sin embargo, el problema general del renderizado de modelos masivos está lejos de ser resuelto, y muchos aspectos necesitan ser investigados con más profundidad [5, 6, 7].

En particular, muchas de las técnicas de aceleración anteriores se han diseñado e implementado especialmente para métodos de rasterización acelerados en la GPU que utilizan una iluminación local sencilla. El cálculo de efectos no locales, como sombras e interreflexiones, requiere la implementación de métodos aproximados de pasadas múltiples, no triviales de realizar en el contexto de un renderizador *out-of-core* en tiempo real, debido a la necesidad de programar cuidadosamente el acceso a datos y el procesamiento de pases de render basados en complejas dependencias entre partes de escenas desarticuladas. Esto ha limitado en gran medida la calidad de las imágenes en los recorridos en tiempo real basados en soluciones de rasterización [5]. Por el contrario, se han propuesto sistemas de renderizado de alta calidad que soportan la iluminación avanzada basados en el trazado de rayos en tiempo real [4], pero sólo se han realizado soluciones completamente *out-of-core* mediante aceleración en la CPU. El complejo patrón de acceso del trazado de rayos se beneficiaría de la compresión, por ejemplo, para ajustar completamente los datos en la memoria de la GPU para obtener una renderización de baja latencia, pero las soluciones más avanzadas para la compresión de las estructuras de datos espaciales totalmente renderizables y de los datos de la escena asociados a éstos reducen demasiado el tiempo de acceso como para soportar el rendimiento requerido para el tiempo real, o bien no están comprimiendo los datos lo suficiente como para soportar modelos de gran tamaño [6, 8].

El trabajo presentado en esta tesis está motivado principalmente por la necesidad de eliminar estas limitaciones.

A.2 Objetivos

Esta tesis se ha realizado con el objetivo de contribuir a la mejora del estado del arte en las áreas de exploración de modelos 3D masivos, investigando el potencial de tecnologías nóveles que traspasen los límites de complejidad en los modelos y calidad de render en entornos interactivos, utilizando hardware de uso común. En particular, se fijan los siguientes objetivos:

- **Mejorar la creación masiva de modelos extendiendo los procesos de fusión de datos a estructuras escalables**. Aún mientras que el enfoque principal de la tesis es el diseño y desarrollo de técnicas para explorar interactivamente modelos masivos, se propone como primer objetivo abordar el problema del manejo eficiente y la creación de modelos 3D a partir de cantidades masivas de datos adquiridos. Ahondar en este tema permitirá comenzar a trabajar no sólo en modelos ya creados, sino también a partir de los datos brutos utilizados para construirlos. Dado que las actuales técnicas topográficas para capturar escenas reales, como la fotografía digital, la fotogrametría y el escaneado láser, están haciendo posible adquirir rápidamente representaciones de forma y color muy densas de objetos y ambientes, me propongo como objetivo la creación de métodos y técnicas escalables para gestionar una representación de grandes bases de datos en bruto y fusionarlos para producir formas de color limpias, renderizables y detalladas, que puedan ser utilizadas en aplicaciones de renderizado interactivo.
- **Mejorar la exploración masiva de modelos mediante una estrategia de división en lotes de trabajo *out-of-core***. Aunque las GPUs actuales soportan modelos generales de programación y permiten ejecutar algoritmos complejos sobre estructuras de datos de aceleración, la gestión eficiente de la memoria y la programación de los cálculos para el trazado de rayos es mucho más difícil que para la rasterización, lo que provoca problemas de rendimiento y/o ancho de banda, cuando se intenta integrar la rasterización y el trazado de rayos en la misma aplicación, por ejemplo, para calcular iluminación global compleja. En esta tesis se estudia cómo utilizar la visibilidad con una planificación inteligente de trabajo por lotes, para poder procesar y renderizar directamente modelos 3D de gran tamaño *out-of-core*, desde dentro de un núcleo de renderizado flexible que soporte naturalmente iluminación compleja.
- **Mejorar la exploración masiva de modelos mediante una estrategia de compresión *in-core***. Los métodos adaptativos *out-of-core* no tienen límites en

el tamaño de los modelos que pueden gestionar, debido al hecho de que trabajan en lotes de tamaño limitado, pero introducen de forma inherente cierta latencia mientras se van cargando datos en la GPU para actualizar el conjunto de trabajo renderizable. En varias aplicaciones, esta latencia, aunque sea mínima, es un factor limitante. Por lo tanto, hacen falta también técnicas permitan cargar la mayor cantidad de datos en la memoria del núcleo como sea posible, y en un formato totalmente renderizable. El voxelizado de superficies para la representación de escenas complejas en 3D ha sido ampliamente utilizadas recientemente para este propósito, ya que ofrecen una codificación de los datos muy fácil de interpretar. Sin embargo, en la actualidad, estas representaciones necesitan grandes cantidades de memoria. para soportar el renderizado de modelos masivos, en esta tesis se propone como objetivo mejorar el rendimiento de compresión de este tipo de representación basadas en vóxeles, minimizando su tamaño en memoria y manteniendo tiempos de cómputo similares a los actuales, y, por lo tanto, permitiendo aplicar el trazado de rayos en la GPU a modelos masivos.

- **Validar las diferentes estrategias propuestas con casos de uso de escenas masivas del mundo real**. Con el fin de verificar en la práctica todos estos enfoques, uno de los objetivos será realizar implementaciones de prototipos realmente viables capaces de proporcionar un rendimiento sin precedentes en datos masivos del mundo real. Por lo tanto, cada uno de los métodos deberá compararse con un gran número de datos masivos y con otras soluciones existentes.

A.3 Métodos escalables para la gestión *out-of-core* de nubes de puntos y fusión de datos

La proliferación de la fotografía digital y los dispositivos de digitalización 3D están haciendo posible adquirir, a un coste razonable, muestras muy densas y precisas de las propiedades geométricas y ópticas de las superficies de objetos reales. Una gran variedad de aplicaciones en el campo del patrimonio cultural se benefician especialmente de esta evolución tecnológica. De hecho, este progreso en la tecnología está haciendo factible la construcción de réplicas digitales a color, no sólo de objetos digitales individuales sino a gran escala. Las reconstrucciones digitales de precisión generadas a partir de medidas objetivas tienen muchas aplicaciones, que van desde la restauración virtual hasta la comunicación visual.

Las nubes de puntos son uno de los tipos de datos más utilizados para representar tales modelos en campos como la ingeniería, las ciencias ambientales o el patrimonio cultural. En manera natural son escalables, ya que cuantas más muestras tenga el conjunto de datos, más fina será la representación del objeto o escena real. Sin embargo, los actuales conjuntos de datos de nubes de puntos pueden llegar a ser imposibles de renderizar en el hardware actual, dado que pueden superar fácilmente los miles de millones de muestras. La gestión de conjuntos de datos tan grandes requiere técnicas escalables. En este capítulo, considero el caso común en el que una nube de puntos muy grande debe ser optimizada para permitir rápidamente la exploración, análisis y coloración de múltiples resoluciones. Ejemplos comunes de aplicaciones de estas estructuras son:

- fusión de nubes de puntos con datos fotográficos para, por ejemplo, la creación de modelos fotorrealistas, realizados con láser para la forma, y cámaras para el color;
- extracción de características geométricas como planos, cilindros, etc. con propósito de análisis en ingeniería;
- exploración en tiempo real de modelos de nubes de puntos masivos en una variedad de ordenadores, adaptando la complejidad del render a las capacidades de cada plataforma.

Todos estos casos de uso requieren técnicas capaces de optimizar estáticamente una nube de puntos para transformarla en una estructura multirresolución *out-of-core*, de la que extraer, en tiempo de ejecución, los detalles de las distintas partes para las operaciones requeridas. En esta tesis se presentan una implementación de una arquitectura basada en un refinamiento de la técnica "*Layered Point Clouds*", así como una aplicación particularmente desafiante de la fusión de datos de puntos e imágenes en el ámbito del patrimonio cultural: la digitalización eficiente de la forma y el color de obras de arte.

Contribuciones. Las principales contribuciones de esta tesis en este primer ámbito son:

- un diseño general de un sistema escalable capaz de crear, colorear, analizar y explorar nubes masivas de puntos completamente *out-of-core*;
- un protocolo de adquisición sencillo basado en escaneo láser y fotografía con flash para generar nubes masivas de puntos con color;

- un método semiautomático de eliminación de soportes y oclusiones y masquerado de fotografías para generar nubes de puntos libres de ruido con mínima intervención manual;
- una implementación escalable de una completa *pipeline* de masquerado, edición, relleno de huecos, corrección de color y combinación de fotografías, que funciona completamente *out-of-core* sin límites en el tamaño del modelo o número de fotos.
- la evaluación del método y las herramientas, en una aplicación real de gran escala.

A.4 Exploración escalable mediante planificación adaptativa y algoritmos *out-of-core*

La exploración interactiva de grandes modelos, incluidos aquellos capturados del mundo real y de alta densidad generados con los métodos descritos en la sección anterior, requieren técnicas eficientes de rendering para poder superar las limitaciones de tiempo de cálculo. Los métodos de *culling* por oclusión o *view-frustrum*, además de la multirresolución, se utilizan comúnmente para cargar y procesar sólo la parte visible de la escena y, por lo tanto, para hacer un rendering *output-sensitive*. Estas técnicas son especialmente eficaces en el caso de la rasterización usando la *pipeline* de la GPU estándar, adaptada para hacer *streaming* ordenado por objetos de lotes de primitivas geométricas (en particular, triángulos). En el capítulo 4 de la tesis, se amplía este enfoque a un entorno de trazado de rayos más flexible, proponiendo un novedoso método que generaliza el método *CHC++* de oclusión jerárquica. Este nuevo enfoque explota la *pipeline* de rasterización y las consultas de oclusión de hardware para crear lotes coherentes de trabajo que serán procesados por núcleos de *ray-tracing* codificados en *shaders* locales. Mediante la combinación de dos jerarquías diferentes, una en el espacio de rayos, y otra en el espacio de los objetos, el método es capaz de compartir los resultados intermedios de los algoritmos de *traversal* entre múltiples rayos. Además, se explota la coherencia temporal entre conjuntos de rayos similares entre *frames* recientes en el tiempo y el actual. Una gestión adecuada del estado de visibilidad permite beneficiarse del *culling* por oclusión para tipos de rayos menos coherentes, como las reflexiones difusas. Dado que las escenas de gran tamaño siguen siendo un reto para los *ray-tracers* modernos en la GPUs, el método presentado es muy útil para escenas de complejidad media a alta, especialmente porque es intrínsecamente compatible con el trazado de rayos de modelos muy complejas que no caben en la memoria de la GPU (*out-of-core*). Para escenas *in-core*, nuestro método es comparable al trazado de rayos en CUDA y funciona hasta seis veces mejor que otros *ray-trafers* basados en *shaders*.

Contribuciones. Abordamos estos problemas proponiendo una técnica de trazado de rayos que está diseñada para ser integrada en la *pipeline* de rasterización por *streaming*. La idea central del método es aprovechar la *pipeline* de rasterización estándar del hardware gráfico, junto con las *occlusion queries*, para crear lotes de trabajo coherentes de trazado de rayos en la GPU. Mediante la combinación de las jerarquías antes mencionadas (espacio objetos y espacio rayos), y haciendo uso de la coherencia temporal, se minimiza la sobrecarga en el algoritmo de *traversal* por la estructura, y el método puede centrarse en el cálculo de las intersecciones rayo - objeto para conjuntos de rayos y objetos significativamente reducidos. Este enfoque de computación por lotes y gestión de memoria permite utilizar los mismos esquemas de *streaming* pero para el trazado de rayos, en vez de la rasterización. De esta manera, se abre la puerta a una integración flexible de la rasterización y el *ray-tracing*, tanto para escenas dinámicas como *out-of-core*. Se muestra la eficacia del método para varios tipos de rayos, como aquellos usados para calcular sombras suaves, o las interreflexiones difusas. Las principales contribuciones de este trabajo son:

- Método de descarte por oclusiones para *ray-tracing* usando la *pipeline* de rasterización, que es hasta $6\times$ más rápida que un trazador de rayos estándar basado en OpenGL.
- Un medio para planificar las intersecciones rayo - triángulo de partes visibles de la jerarquía de la escena en la GPU, que permite una extensión simple y natural al trazado de rayos *out-of-core*.

Ademas, se muestra una implementación eficiente del método usando el estándar OpenGL, con una serie de beneficios:

- un algoritmo novel para identificar automáticamente grupos de rayos que pueda intersecar con triángulos;
- puede ser fácilmente adaptado para escenas dinámicas (basta una jerarquía espacial aproximada);
- consigue una paralelización eficiente mediante el hardware de rasterización estándar (así que la gestión de los datos por streaming y su asignación a los diferentes cores es automáticamente llevada por el hardware);

- permite utilizar el método en equipos más antiguos, ya que usa una *pipeline* estándar.

A.5 Exploración escalable mediante compresión y algoritmos *in-core*

Con el aumento rendimiento y la capacidad de programación de las unidades de procesamiento gráfico, el *ray-casting* en la GPU se está convirtiendo en una solución eficaz para muchos problemas de renderizado en tiempo real. Con el fin de manejar escenas detalladas de gran tamaño, es fundamental crear una representación de escena compacta y eficiente para acelerar las consultas de intersección de geometría con los rayos, y en este sentido se han propuesto se han propuesto múltiples soluciones (véase Sec. 5.2). Entre ellos, los *sparse voxels octrees* (SVO) [131] han proporcionado resultados muy eficientes, ya que se pueden crear a partir de distintos tipos de representaciones de escenas, y la codifican descartando eficientemente el espacio vacío. Esto aporta grandes beneficios sea en el rendimiento del trazado de rayos, como en la ocupación de memoria , e implícitamente proporciona un mecanismo de niveles de detalle (LOD, por sus siglas en inglés). Pero su costo de memoria es aún relativamente alto, y por lo tanto requiere también un ancho de banda, así que estos enfoques voxelizados se han visto limitados a tamaños y resoluciones de escena moderados, o a efectos que no requieren detalles geométricos precisos (como por ejemplo, sombras suaves).

Aunque se han propuesto muchas representaciones extremadamente compactas para modelos volumétricos de alta resolución, especialmente en el área de renderizado de volumenes [132], el gran aumento en los *ratios* de compresión de estas soluciones se equilibra con el aumento de los costos de descompresión y *traversal*, lo que las hace difícilmente utilizables en entornos generales. Esto ha desencadenado una búsqueda de representaciones más simples que pueden proporcionar *datasets* más compactos, in necesidad de excesivas costes de descompresión. Kämpe et al. [14] han demostrado recientemente que, para escenas típicas de simulaciones y videojuegos, una maya binaria de vóxeles puede ser representada órdenes de magnitud más eficientemente que usando un SVO, simplemente fusionando subárboles idénticos, generalizando el árbol de vóxeles disperso a un gráfico acíclico dirigido (SVDAG). Esta representación es enormemente compacta, ya que los nodos pueden compartir punteros con subárboles idénticos, y sigue siendo tan rápida como las SVOs y los *octrees* simples, ya que la rutina de trazado es esencialmente la misma.

En esta parte de la tesis (capítulo 5) se muestra cómo una compresión eficiente de la geometría sin pérdidas puede combinarse con un buen rendimiento de trazado de rayos, mediante la fusión no sólo de subárboles completamente idénticos, sino también de aquellos similares después de aplicarles una transformación de similitud, a distintos niveles del grafo (nodos internos y hojas), y compactando los punteros de los nodos de acuerdo a su frecuencia de ocurrencia. La estructura resultante, que bautizamos como *Symmetry-aware Sparse Voxel DAG* (SSVDAG) puede ser construida eficientemente mediante un algoritmo *out-of-core* ascendente que reduce un SVO a un SSVDAG mínimo alternando diferentes fases en cada nivel. Primero, todos los nodos que representan subárboles similares son agrupados y reemplazados por un solo representante. Luego, los punteros a esos nodos en el nivel inmediatamente superior son reemplazados por punteros etiquetados al único representante, donde la etiqueta codifica la transformación que necesita ser aplicada para recuperar el subárbol original del representante. Finalmente, los representantes son ordenados por el número de referencias que les apuntan, lo que permite una eficiente codificación de los punteros con *bitrate* variable. Demostramos que, seleccionando reflexiones en los planos axiales a lo largo de las direcciones principales de la cuadrícula como transformación de simetría, se puede lograr un buen rendimiento de construcción y trazado.

Contribuciones. En el ámbito de la compresión de modelos masivos, las contribuciones de esta tesis son:

- Una nueva estructura tridimensional compacta llamada *Symmetry-aware Sparse Voxel DAG* (SSVDAGs) que puede representar sin pérdida geometría voxelizada de escenas y datos reales con una enorme compresión, y eficientemente renderizable.
- Un algoritmo *out-of-core* basado en un método simple de multipases para construir esta representación desde un SVO o un SVDAG;
- Una modificación sencilla de los algoritmos estándares en GPU para poder realizar el *traverse* y render de la estructura con muy poca carga extra. Se describe, en particular, los detalles de un método basado en el algoritmo multirresolución del *Digital Differential Analyzer* (DDA), implmentando con una pila completa (*full-stack*).

Nuestra técnica de reducción se basa en el supuesto de que las representaciones de la escena original son geométricamente redundantes, en el sentido de que contienen una gran cantidad de subárboles que son similares con respecto a una transformación reflexiva. Nuestros resultados, ver Sec. 5.6, demuestran que esta suposición es válida para escenas habituales del mundo real, de tipos y características muy diferentes, que van desde modelos CAD de gran tamaño, a escaneados en 3D, hasta modelos de juegos típicos. Esto permite representar escenas muy grandes a alta resolución en las GPUs, y soportar renderizados geométricos precisos, así como fenómenos de alta frecuencia, como sombras nítidas, con una sobrecarga en el cómputo de trazado de rayos menor al 15%.

A.6 Logros y conclusiones

El trabajo de investigación llevado a cabo durante esta tesis ha dado lugar a los siguientes logros y publicaciones revisadas por pares:

- La introducción de un diseño multiresolución general para un sistema escalable, capaz de crear, colorear, analizar y explorar nubes de puntos masivas totalmente *out-of-core*. Una implementación acelerada en la GPU capaz de procesar y renderizar un conjunto de datos de miles de millones de puntos [9], así como su aplicación en campos como el patrimonio cultural o la ingeniería[10, 11].

  "**Point Cloud Manager: Applications of a Middleware for Managing Huge Point Clouds**". O. A. Mures,A. Jaspe Villanueva, E.J-Padrón, J.R. Rabuñal. Chapter 13 of “Effective Big Data Management and Opportunities for Implementation” book. Pub. IGI Global (2016)

  "**Virtual Reality and Point-based Rendering in Architecture and Heritage**". O. A. Mures, A. Jaspe Villanueva, E.J- Padrón, J.R. Rabuñal.Chapter 4 of “Handbook of Research on Visual Computing and Emerging Geometrical Design Tools” book. Pub. IGI Global(2016)

- Un protocolo de adquisición fácil de aplicar, basado en el escaneo láser y la fotografía con flash para generar nubes de puntos coloreadas, que introduce un nuevo método semiautomático para la eliminación de las oclusiones y enmascaramiento fotográfico para generar *datasets* de nubes de puntos limpios y estructurados, con una mínima intervención manual. El diseño multirresolución previamente introducido permite que todo el enmascaramiento, edición, relleno, corrección de color y mezcla de color funcionen completamente *out-of-core* sin límites en el tamaño del modelo y el número de fotos.

"**Mont'e Scan: effective shape and color digitalization of cluttered 3D artworks**". F. Bettio, A. Jaspe Villanueva, E. Merella, F. Marton, E. Gobbetti, R. Pintus. ACM Journal on Computing and Cultural Heritage, Vol 8, Num 1 (2015)

- Un novedoso enfoque para aprovechar el *pipeline* de rasterización y las *occlusion quesries* por hardware, con el fin de crear lotes coherentes de trabajo para núcleos de *ray-tracing* basados en *shaders* locales [5]. Mediante la combinación de jerarquías tanto en el espacio de rayos como en el espacio de objetos, el método es capaz de compartir los resultados intermedios de *traversal* entre múltiples rayos. Entonces, la coherencia temporal se explota entre *sets* de rayos de los *frames* cercanos. Esta arquitectura de programación permite de manera natural el trazado de rayos *out-of-core*, con la posibilidad de renderizar escenas potencialmente ilimitadas.

"**CHC+RT: Coherent Hierarchical Culling for Ray Tracing**". O. Mattausch, J. Bittner, A. Jaspe Villanueva, E. Gobbetti, M. Wimmer, and R. Pajarola. Computer Graphics Forum Journal Vol 32, Num 2. Presented at Eurographics'15 (2015)

- Un nuevo método de compresión llamado SSVDAG (*Symmetry-aware Sparse Voxel DAG*) [6, 7], que puede representar sin pérdidas geometrías 3D voxelizada, así como un algoritmo *out-of-core* para construir dicha representación a partir de un SVO o un SVDAG, y una modificación sencilla del algoritmo de *ray-casting* en GPU estándar para atravesar y renderizar esta representación con una baja sobrecarga computacional. Esta técnica ha demostrado que es capaz de comprimir un *grid* de vóxeles de hasta $1M^3$ en un *dataset* que cabe completamente en lq memoria de la GPU y renderizarlo en tiempo real.

"**SSVDAGs: Symmetry-aware Sparse Voxel DAGs**". A. Jaspe Villanueva, F. Marton, and E. Gobbetti- ACM SIGGRAPH i3D full paper (2016).

"**Symmetry-aware Sparse Voxel DAGs (SSVDAGs) for compression-domain tracing of high-resolution geometric scenes**". A. Jaspe Villanueva, F. Marton, E. Gobbetti. Journal of Computer Graphics Techniques Vol 2 Num 6 (2017).

"**Voxel DAGs and Multiresolution Hierarchies: From Large-Scale Scenes to**

Pre-computed Shadows". U. Assarsson, M. Billeter, D. Dolonius, E. Eisemann, A. Jaspe Villanueva, L. Scandolo, E. Sintorn. Eurographics Tutorials (2018).

- La evaluación de todos los métodos anteriormente descritos con datos de gran escala del mundo real. En particular, como se describe en los secciones 3.4.7, 5.6, y 4.7, todos las técnicas se han probado con modloes que sobrepasan los cientos de millones de primitivas geométricas.

Además, durante el curso de mis estudios de doctorado y siempre en relación con esta tesis, he contribuido también en una serie de publicaciones que no han sido incluidas directamente en este trabajo, y que se listan en la sección dedicada a los resultado bibliográficos (Sec. 6.3).

Conclusiones y futuro. La tesis ha abordado tanto el problema de la creación como el de la exploración de modelos masivos utilizando técnicas escalables. Mientras que las contribuciones presentadas en el ámbito de las nubes de puntos son, en su mayor parte, a nivel práctico, mostrando una novedosa implementación completa de las mejores prácticas que resuelve problemas muy prácticos, los enfoques estudiados para mejorar la exploración en tiempo real son potencialmente más interesantes en términos de potencial para el trabajo futuro.

La técnica CHC+RT basada en lotes que se ha presentado representa un novedoso uso de la oclusión jerárquica para acelerar el trazado de rayos basado en OpenGL. Este trabajo encaja perfectamente en el área de renderizado de modelos masivos, ya que permite explotar la técnica de carga adaptativa basada en la visibilidad típica de los rasterizadores *out-of-core* actuales en un entorno de *ray-tracing* más flexible, lo que abre la puerta a un renderizado *out-of-core* acelerado en la GPU utilizando sombras e iluminación global. Esto se logra en un contexto muy simple, utilizando una técnica que puede integrarse fácilmente con los renderizadores actuales. Hemos demostrado efectos de iluminación no-locales en grandes modelos a través de sombras e iluminación difusa de primer rebote. Es interesante mencionar, como se comenta en la Sec. 4.9, que la idea de la aceleración basada en la visibilidad usando una jerarquía dual está siendo explotada actualmente para trabajos de iluminación global muy avanzados, como en la técnica de *Megaviews* de Kol et al. [129].

Aunque esta técnica permite renderizar escenas muy grandes, que superan la memoria de la GPU, mediante carga adaptativa, las contribuciones siguientes de la tesis, que usan técnicas de compresión, abordan el problema compactando los datos de forma agresiva para que quepan, en un formato totalmente renderizable, en la memoria de la GPU. Considero esta parte como mi principal contribución en mi trabajo doctoral. De hecho, este enfoque encaja perfectamente con la tendencia actual de aumento de memoria las GPUs. Tal aumento, de varios GBs, es todavía demasiado bajo para permitir la renderización de modelos grandes sin comprimir, aunque si al mismo tiempo comienzan a no ser no despreciables. Con este trabajo, se demuestra que, aprovechando las transformaciones de similitud y la codificación inteligente, los *sparse voxel octrees*, unas representaciones muy fáciles de usar en la GPU pero que consumen mucha memoria, pueden transformarse en DAGs de vóxeles dispersos muy compactos para una gran variedad de tipos de modelos, incluidos aquellos provenientes de escaneados láser, modelos CAD y los modelos de videojuegos. Estas representaciones tan compactas son cientos de veces más pequeñas que los datos originales y, por lo tanto, requieren poco almacenamiento sea *offline* que en la GPU, así como poco ancho de banda para su transmisión hacia el hardware gráfico. Al mismo tiempo, la codificación compatible con la GPU permite usar las técnicas avanzadas de trazado de rayos sin pérdida de rendimiento.

Como demuestra las secciones finales de los distintas técnicas presentadas (Sec. 3.4.7, Sec. 4.7 and Sec. 5.6), los resultados de los métodos desarrollados durante esta tesis obtienen el rendimiento esperado, tal y como se describe en la Sec. 2.4 de verificación de las hipótesis, para todos los *datasets* probados, y hacen avanzar el estado del arte en el campo de la exploración de modelos masivos.

En el ámbito de la compresión, de forma similar a otros trabajos sobre DAGs [14, 133, 134], en este tesis se ha trabajado principalmente en la compresión de los datos geométricos. Otros autores han mostrado cómo asociar a la geometría otros atributos como colores o normales [135, 152]. Además de evaluar cómo pueden mejorarse estas técnicas, se han identificado diversas áreas de interés para el trabajo futuro.

En primer lugar, el enfoque usado hasta ahora ha sido la compresión sin pérdidas de la representación voxelizada. Parece muy prometedor, en cambio, considerar también los enfoques con pérdidas (o *lossy*), en las mismas configuraciones. Dado que las representaciones voxelizadas son ya una discretización de otra geometría (por ejemplo, triángulos, nubes de puntos, superficies implícitas, parches de orden superior, etc.), parece razonable considerar que ésta podría ser ligeramente variada para mejorar las posibilidades de encontrar similitudes. Esto podría implementarse utilizando un enfoque de prefiltrado, que reduzca la variabilidad a nivel de hoja (conservando al mismo tiempo algunos errores) para aumentar el número de subárboles similares.

Este enfoque, que promete una compresión mucho mayor, no se ha intentado hasta ahora.

Una segunda mejora muy práctica, podría considerar un estudio a nivel de sistema de las cargas incrementales, utilizando la propia estructura comprimida. Hasta ahora, la compresión agresiva mediante una representación DAG siempre se ha utilizado para estructuras monolíticas mantenidas en la GPU. Esta es una situación muy favorable, ya que la residencia completa de la GPU soporta un renderizado muy eficiente. Sería interesante estudiar, especialmente en el contexto del renderizado en remoto (usando una red de datos, por ejemplo), cómo podría utilizarse esta representación tan compacta en un contexto de carga incremental. Esto requeriría la implementación de un sistema de paginado *ad-hoc* y, a nivel de compresión, una mejora en la ordenación de los datos para apoyar la localización de la geometría a nivel espacial, de modo que sea más probable que los datos cercanos entre ellos se almacenen en la misma página. Actualmente, esta localidad espacial queda completamente anulada por la reordenación basada en la similitud que se realiza a nivel de compresión. Parece muy interesante poder encontrar compromisos para optimizar al mismo tiempo los datos y la similitud espacial. Tal implementación permitiría, por ejemplo, conseguir renderizadores basados en *ray-tracing* muy eficientes en la Web (y eventualmente, incluso en dispositivos móviles), reduciendo enormemente la latencia de carga de datos.

B Curriculum Vitae

Alberto Jaspe Villanueva is an expert researcher of the Visual Computing group (ViC) specialized in the field of Computer Graphics. He also teaches Virtual and Augmented Reality courses at the European Institute of Design. He has been previously awarded with a Mary Curie Early Stage Researcher Fellowship from the DIVA Initial Training Network. He also is a PhD student at University of A Coruña (Spain), where he got his Bachelor and M.Sc. degrees with honors in Computer Science. Before joining CRS4, he worked as a Computer Graphics developer and researcher for VideaLAB and RNASA groups at the same university, where he contributed to projects in the fields of Virtual Reality, Architecture Visualization, Terrain and Point Clouds Rendering, and Natural Interaction. He also has experience in the industry, as he started and managed for two years the R&D department of CEGA Audiovisuals, a company focused on interactive audio and video installations.

Contact Information

Name	Alberto Jaspe Villanueva
Address	CRS4, Ex-Distilleria Pirri. Via Ampere 2, 09134 - Caglari, Italy
E-Mail	ajaspe@gmail.com

Online	`http://albertojaspe.net`
	`https://www.linkedin.com/in/albertojaspe`
	@albertojaspe

Personal Details

Date of Birth	July 10th, 1981
Place of Birth	A Coruña, Spain
Languages	Spanish (native), Galician (native), English (fluent), Italian (fluent)

Education

Since 2013	**Ph.D. candidate on Computer Science.** University of A Coruña (UDC), Spain. Doctoral program in Information and Communications Technologies.
2012 - 2013	**Bachelor on Computer Science**. UDC, Spain
2011 - 2012	**Master on High Performance Computing**. UDC, Spain
2005 - 2011	**Technical Engineering in Computer Science**. UDC, Spain

Employment History

Since 2016	**Expert Researcher** at the Visual Computing Group of the Center for Advanced Studies, Research and Development in Sardinia (CRS4), Italy.
2013 - 2015	**Marie Curie Early Stage Researcher** at the Visual Computing Group (CRS4), Italy.
2011 - 2013	**Research & Development Manager** at CEGA Audiovisuals S.L., Spain
2003 - 2010	**Developer & Researcher** at the Visualization for Engineering Architecture and Urban Design Group (VideaLAB) at UDC, Spain.
2001 - 2003	**Undergraduate intern** at the Artificial Neural Networks and Adaptive Systems lab (RNASA) at UDC, Spain.

Other Professional Activities

Since 2018	**Adjunct lecturer** at European Institute of Design (IED), teaching Virtual and Augmented Reality courses.
Since 2018	**Program Committee member** at Eurographics Smart Tools and Applications in Graphics (STAG).
Since 2013	**Invited reviewer** for various IEEE and ACM journals.
2006 - 2012	**Invited lecturer** for seminars on Computer Graphics and Interaction (UDC).
2006	**Adjunct lecturer** at Master in Digital Creativity and Communication (UDC), teaching Audio & Video Editing course.

Selected Publications

Journal Articles

- **Artworks in the Spotlight: Characterization with a Multispectral Dome**.
  I. Ciortan, T. Dulecha, A. Giachetti, R. Pintus, A. Jaspe, and E. Gobbetti.
  Materials Science and Engineering Journal (2018).
- **Symmetry-aware Sparse Voxel DAGs (SSVDAGs) for compression-domain tracing of high-resolution geometric scenes**. A. Jaspe, F. Marton, E. Gobbetti.
  Journal of Computer Graphics Techniques (2017).
- **CHC+RT: Coherent Hierarchical Culling for Ray Tracing**.
  O. Mattausch, J. Bittner, A. Jaspe, E. Gobbetti, M. Wimmer, and R. Pajarola.
  Computer Graphics Forum Journal Vol 32, Num 2. Presented at Eurographics'15 (2015)
- **Mont'e Scan: effective shape and color digitalization of cluttered 3D artworks**.
  F. Bettio, A. Jaspe, E. Merella, F. Marton, E. Gobbetti, R. Pintus.
  ACM Journal on Computing and Cultural Heritage, Vol 8, Num 1 (2015)
- **Automatic room detection and reconstruction in cluttered indoor environments with complex room layouts**. C. Mura, O. Mattausch, A. Jaspe, E. Gobbetti, R. Pajarola.
  Computer & Graphics Journal Num 44 (2014)
- **IsoCam: interactive visual exploration of massive cultural heritage models on large projection setups**.
  F. Marton, M. Balsa, F. Bettio, M. Agus, A. Jaspe, and E. Gobbetti.
  ACM Journal on Computing and Cultural Heritage, Vol 7, Num 2 (2014)
- **ExploreMaps: Efficient Construction and Ubiquitous Exploration of Panoramic View Graphs of Complex 3D Environments**.
  M. Di Benedetto, F. Ganovelli, M. Balsa, A. Jaspe, R. Scopigno, and E. Gobbetti.
  Computer Graphics Forum Journal, Vol 33, Num 2. Presented at EuroGraphics'14 (2014)
- **Space perception in architectonic visualization using immersive virtual reality**.
  L. A. Hernández, J. Taibo, A. Seoane, A. Jaspe.
  Architectonic Graphic Expression Journal, Num 18 (2011)
- **Physically Walking in Digital Spaces: A Virtual Reality Installation for Exploration of Historical Heritage**.
  L. A. Hernández, J. Taibo, D. Blanco, J. A. Iglesias, A. Seoane, A. Jaspe y R. López.
  International Journal of Architectural Computing, Vol 5, Num 13 (2007)

Conference Papers

- **Objective and Subjective Evaluation of Virtual Relighting from Reflectance Transformation Imaging Data**. R.Pintus, T. Dulecha, A. Jaspe, A. Giachetti, I. Ciortan, E. Gobbetti. Eurographics Workshop on Graphics and Cultural Heritage (2018).
- **Voxel DAGs and Multiresolution Hierarchies: From Large-Scale Scenes to Pre-computed Shadows**. U. Assarsson, M. Billeter, D. Dolonius, E. Eisemann, A. Jaspe, L. Scandolo, E. Sintor. Eurographics Tutorials (2018).
- **PEEP: Perceptually Enhanced Exploration of Pictures**. M. Agus, A. Jaspe, G. Pintore, E. Gobbetti. International Workshop on Vision, Modeling and Visualization (VMV) full paper (2016).
- **SSVDAGs: Symmetry-aware Sparse Voxel DAGs**.
  A. Jaspe, F. Marton, and E. Gobbetti.
  ACM SIGGRAPH i3D full paper (2016).
- **SOAR: Stochastic Optimization for Affine global point set Registration**. M. Agus, E. Gobbetti, A. Jaspe, C. Mura, and R. Pajarola.
  International Symposium on Vision, Modeling, and Visualization VMV'14 full paper (2014)
- **Practical line rasterization for multi-resolution textures**. J. Taibo, A. Jaspe, A. Seoane, Marco Agus, and L. A. Hernandez.
  STAG'14 full paper (2014)
- **Robust Reconstruction of Interior Building Structures with Multiple Rooms under Clutter and Occlusions**.
  C. Mura, O. Mattausch, A. Jaspe, E. Gobbetti, and R. Pajarola.
  CAD/Graphics'13 full paper (2013)
- **Automatic Geometric Calibration of Projector-based Light-field Displays**. M. Agus, E. Gobbetti, A. Jaspe, G. Pintore, and R. Pintus.
  EuroVIS'13 short paper (2013)
- **Interactive installations and virtual reality in the museum. The Galicia Dixital experience**. L. Hernández, A. Seoane, R. López, A. Jaspe.
  ICT'07, Conference in Historical Heritage full paper (2008)
- **Hardware-Independent Clipmapping**. A. Seoane, J Taibo, L. Hernández, R. López and A. Jaspe.
  WSCG'07 full paper. The 15th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (2007)
- **Real-time visualization of geospatial features through integration of GID with a realistic 3D terrain dynamic visualization system**. L. A. Hernandez, J. Taibo, A. Seoane, R. López, A. Jaspe, A. Varela.
  ICC'05 full paper. XXII International Cartographic Conference (2005)
- **The Creativity Space: An Immersive VR Framework for 3D Creation**. L.A. Hernández, J. Taibo, A. Jaspe, R. López, D. Blanco, R. López, A. Seoane.
  Digital Engineering Workshop, CAD/CAM Workshop full paper (2005)

Book chapters

- **Point Cloud Manager: Applications of a Middleware for Managing Huge Point Clouds**. O. A. Mures, A. Jaspe, E.J-Padrón, J.R. Rabuñal.
  Chapter 13 of "Effective Big Data Management and Opportunities for Implementation" book. Pub. IGI Global. ISBN: 9781522501824 (2016)
- **Virtual Reality and Point-based Rendering in Architecture and Heritage**. O. A. Mures, A. Jaspe, E.J- Padrón, J.R. Rabuñal.
  Chapter 4 of "Handbook of Research on Visual Computing and Emerging Geometrical Design Tools" book. Pub. IGI Global. ISBN: 9781522500292 (2016)
- **Acceleration of AI algorithms using GPUs**. A. Seoane and A. Jaspe. Chapter of "Encyclopedia of Artificial Intelligence" book. Pub. IGI Global. ISBN: 978-1-59904-849-9 (2008)
- **Mapping Large Textures for Outdoor Terrain Rendering**. A. Seoane, J. Taibo, L. Hernández, A. Jaspe.
  Chapter of "Game Programming Gems 7" book. Pub. Charles River Media. ISBN: 978-1584505273 (2008)