CEIG - Spanish Computer Graphics Conference (2025)
O. Argudo and O. Iparraguirre (Editors)

Multidisciplinary Development of an Educational 3D Simulation Game for Bee Biology using Advanced Graphics Techniques

Iago Caldentey [1], Francisco J. Perales [1] and Mar Leza [2]

[1]Universitat de les Illes Balears
Departamento de Ciencias Matemáticas e Informática
[2] Universitat de les Illes Balears
Departamento de Biología

Abstract

Be a Bee is an 3D open-world simulation video game that applies advanced computer graphics techniques to achieve a high-performance, visually rich, and interactive environment. The project focuses on procedural terrain generation, optimized rendering, and efficient entity simulation within Unity's ***DOTS!*** *(****DOTS!****) and* ***ECS!*** *(****ECS!****).*

CCS Concepts

• ***Computing methodologies*** → *Procedural generation; Educational simulation; Virtual ecology;* • ***Hardware*** → *GPU optimization;* • ***Software and its engineering*** → *Data-oriented design;*

1. Introduction

Be a Bee is a 3D simulation game designed to raise awareness about bees and their ecological role through an immersive, educational experience. Leveraging advancements in computer graphics, the game uses procedural terrain, mesh instancing, LODs, and GPU-driven optimizations to simulate a realistic ecosystem efficiently. Built with Unity's DOTS and ECS, it incorporates custom memory management and interactive features like a pollen particle system. The project demonstrates how combining biology and real-time graphics can create effective educational tools that balance scientific accuracy, performance, and engagement.

2. Related Work

Educational simulation games have been shown to be highly effective teaching tools [VM17], with games like Eco and Foldit demonstrating how interactive systems can intuitively engage users with complex scientific concepts. These examples underscore the value of gamification in promoting environmental and scientific literacy.

Procedural generation has been widely applied to create vast, dynamic virtual environments, supporting realism and scalability in ecological simulations [Par14]. Rendering optimizations—including **LOD!** (**LOD!**) [HD04], mesh instancing [BB22], vertex compression, and GPU-driven techniques—are essential for maintaining performance in large-scale, real-time systems.

Although prior studies have modeled bee behavior and pollination in ecological contexts [CGF*24], few have combined scientific accuracy with real-time graphics and interactive gameplay. Be a Bee bridges this gap by integrating scientific modeling with engaging, high-performance educational simulation. Additionally, to reduce computational overhead, it replaces Unity's built-in physics-based ray casting with a custom ray-cast algorithm.

3. Procedural terrain generation

To reduce CPU and GPU load and improve efficiency, the world is divided into chunks, enabling optimized rendering and spatial data management for object scanning during simulation. The environment is procedurally generated using OpenSimplex2S noise combined with fractal Brownian motion, which layers noise with varying parameters to create realistic and immersive terrain.

4. Vertex compression

The world, to be rendered, requires generating a mesh. This mesh is composed of data which represents different vertices. A vertex, in this case, is composed by a position, normal vector and its texture coordinates, which are required in the later stages inside the render graphics pipeline.

Here we can take advantage of how our world behaves and some vertex compression techniques can be used. The compression done for each vertex attribute is discussed separately:

- Position:
  - X & Z: Since the world is a grid, the X and Z coordinates will have a discrete value within the range [0,32]. Therefore, we can avoid sending the X and Z values to the GPU and

DOI: 10.2312/ceig.20251111

calculate them in the Vertex Shader using the vertex index. The X and Z coordinates can be calculated using the index in the following way:

$$x = \frac{\text{Vertex}_{\text{ID}}}{\text{VPR}}$$

$$z = \text{Vertex}_{\text{ID}} \mod \text{VPR}$$

The VPR value refers to the number of vertices per row. This will depend on the mesh's LOD, a concept that will be developed further. For now, we need to know that there are 3 levels of detail, so the Vertex Shader will need to receive, in the vertex information, 2 bits indicating which one it is.

– Y value: The height at which the vertex is located falls within the continuous range [0,100]. Thus, we need to store both the decimal part (2 digits) and the integer part (3 digits) in the vertex. In this way, the Y value will only occupy 10 bits instead of 32.

Therefore, the position can be stored in 12 bits instead of 96.

- Normal vector: The normal vector has 3 components that oscillate within the continuous range [-1, 1]. This value is stored in a float, but we do not require such high precision. Therefore, we can reduce each component from 32 bits to only 4 bits for the absolute value. Finally, the signs of the X and Z components must be added to the vertex information, with each occupying 1 bit. For the Y axis, the sign can be omitted because we know that, for the specific case of the project, the Y component will oscillate within the range [0,1]. Thus, the 96 bits previously occupied by the vector are packed into 14 bits.
- UV Coordinates: Since it is a texture that repeats across the entire mesh, we can calculate the values of U and V, avoiding storing them in memory, in the following way:

$$U = \text{Vertex}_{\text{ID}} \mod 2$$

$$V = \frac{\text{Vertex}_{\text{ID}}}{\text{VPR}+1} \mod 2$$

Therefore, for each vertex, we only need 26 bits instead of 256, and it is formatted as shown below:

$S_Z S_X$ $N_Z N_Z N_Z N_Z N_Y N_Y N_Y N_Y$ $N_X N_X N_X N_X$ YYYYYYYYYY DD

Normal vector | Height | **LOD!**

5. Entities

To simulate a large number of entities, each with its own behavior and state, a data-oriented approach is required, which is achieved through the use of an Entity-Component-System (**ECS!**) architecture. Basically, **ECS!** distinguishes between the following core elements:

- Entity: discrete representation which has its own set of data. In Unity, it is simply a unique identifier.
- Component: data container for an entity.
- System: provides logic that transforms data inside components.

This architecture guarantees an efficient memory management taking benefit of the locality principle. Additionally, it allows batch and parallel processing of large chunks of entities that fall into the same archetype. For further clarification, refer to the diagram in the Figure 1.

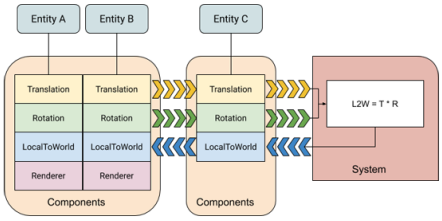

Figure 1: *ECS example usage diagram*

6. Mesh instancing

Rendering many entities puts a heavy load on the CPU due to numerous draw calls. Mesh instancing reduces this by combining multiple draw calls into one, using a uniform buffer to pass entity-specific data like position to the vertex shader. For efficient processing, entities must be grouped by shared mesh type. While the ECS architecture helps maintain data locality, decorative world objects—outside the ECS—must also be organized by type for performance.

7. Animations

To provide immersion and realism to the world we are creating, entities and elements of the world must be animated. Usually, animations cause a heavy load for the CPU as it must compute all vertices new positions for each mesh every frame. Currently, the CPU is already the bottleneck due to handling the simulation of a large number of entities. Therefore, it is crucial to reduce this animations computational costs from the CPU taking advantage of the GPU's main goal, which is parallel computing.

7.1. Wind in decorative objects

Adding the visual wind effect on trees and plants helps immensely to realism of a virtual world. This animation will be a procedural animation computed directly in the vertex shader, that is an animation that will be calculated in real-time per vertex. The new vertex position will be calculated following this formula:
$P' = (sin((P + Time) * WIND\_SPEED) * WIND\_STRENGTH) + P$
Also, we must ensure that only the vertices that are not directly connected to the ground are animated. To do so, only the positions of the vertices where $V = 1$ will be modified without any branching by modifying the previous formula to the following:
$P' = (sin((P + Time) * WIND\_SPEED) * WIND\_STRENGTH) * V + P$

7.2. Animation baking into texture

Another powerful technique to remove completely all computational costs needed for animations is baking all the vertex data per frame of an animation and inserting it into a texture that will be later passed and read from the shader.

The texture in the Figure 2 stores the *flying* animation of the bees. The vertices are stored across the X-axis and each row represents a key-frame of the animation.

Figure 2: *Flying animation baked into texture*

8. Levels of detail

To improve performance in a large simulated world, **LOD!** techniques are used to reduce unnecessary processing and rendering. This involves creating different variations of the same object, selected based on distance to the camera. **LOD!**s help lower both graphical and computational costs and can be applied to various resources like textures, models, animations, and behaviors. In this project, four distance-based LOD levels are defined per object, with rendering skipped entirely at the farthest range.

8.1. Meshes

Meshes are a primary focus for applying this technique, as reducing the total number of triangles—and thus the total number of vertices—to be processed has a significant impact on GPU performance. See as examples both Figure 3 and 4.

Figure 3: *Bee model in different **LOD!**s*

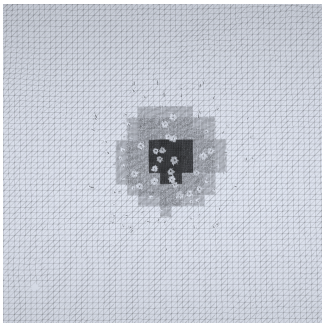

Figure 4: *Terrain rendered with different **LOD!**s depending on distance to camera*

8.2. Textures

Texture's **LOD!**s are useful to avoid the additional cost caused by the *minification filtering*† process that is applied when multiple *texels* coexist on the same screen pixel by reducing the resolution of the texture. Moreover, using **LOD!**s reduce the effects caused by *aliasing*. See Figure 5 as an example.

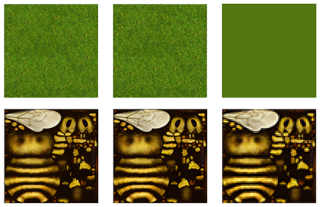

Figure 5: *Bee and terrain textures in different **LOD!**s*

8.3. Shaders

To avoid unnecessary GPU calculation costs during the different rendering stages, using different shaders for different **LOD!**s is a requirement. Different shaders help to reduce the number of light and shadow calculations. These are done per-pixel in the fragment shader for the highly detailed shaders, but done per-vertex in the vertex shader with a later linear interpolation between these values for the less detailed shaders.

8.4. Animations

In further objects from the camera, animations can be less detailed by:

- Reducing number of *keyframes* and removing interpolation.
- Avoid simulation for further away instances as it will not be noticeable for the player.
- Reducing the number of bones decreases the computational load per mesh by minimizing the total number of influences per vertex.

Also, by using mesh **LOD!**s, the animations are less expensive because less vertices must be processed.

9. Ray-cast algorithm

Unity's built-in ray-cast becomes increasingly costly with more entities, as it relies on frequent physics world updates, negatively impacting performance. To address this, a custom, faster ray-cast algorithm was implemented, eliminating the need for a physics world and its associated overhead.
The custom ray-cast algorithm achieves speed by eliminating unnecessary calculations. It first discards distant entities, then uses a dot product check against a squared distance-based threshold to filter candidates. Only if these checks pass are more expensive operations, like distance calculation, performed to find the closest entity to the ray.

† Process that samples the texels that share screen pixel to compute the final color to output.

Algorithm 1 RayCast Function

```
function RayCast(src : float, direction : vec3, range : float) Entity
    closestDot ← 0
    closestEntity ← null
    for all entity ∈ Entities do
        tag ← entity.SelectableTag
        transform ← entity.LocalTransform
        distsq ← sqdist(src, transform.Position)
        if distsq ≤ range then
            dirToBee ← transform.Position − src
            dotProduct ← dot(norm(dirToBee), norm(direction))
            threshold ← lerp(0.9875, 1.0, distsq/range)
            if dotProduct ≥ threshold then
                pNBee ← (normalize(direction) × √distsq) + src
                distToBeeRay ← sqdist(pNBee, transform.Position)
                if distToBeeRay ≤ CollisionRadius then
                    if dotProduct > closestDot then
                        closestEntity ← entity
                        closestDot ← dotProduct
                    end if
                end if
            end if
        end if
    end for
    return closestEntity
end function
```

10. Results

This section presents benchmarks and tests to demonstrate the impact of various advanced techniques and optimizations implemented.

Hardware used: CPU: Intel Core i7-9700F @ 3.00GHz, GPU: NVIDIA GeForce GTX 1660 SUPER, RAM: 16 GB.

Entities	100	10.000	100.000
% CPU	28%	45%	80%
% GPU	0.2%	2%	3.8%
RAM (MB)	1004	1070	2036
VRAM (MB)	200	200	200

Table 1: *Results of usage benchmarks*

Entities	100	10.000	100.000
FPS Range (CPU)	[150-220]	[70-110]	[9-11]
FPS Range (GPU)	[180-300]	[130-230]	[40-80]

Table 2: *FPS with CPU-driven and GPU-driven animations*

Entities	100	10,000	100,000
Own Ray-cast (ms)	0.0001	0.001	0.001
Unity's Ray-cast (ms)	0.0015	0.02	0.1

Table 3: *Time comparison (ms) of ray-cast algorithms*

The new ray-cast algorithm improves the execution time of Unity's built-in ray-cast, while also eliminating the need for periodic updates to the physics world, see Table 4, thereby reducing drastic performance fluctuations.

Entities	100	10,000	100,000
Time per frame (ms)	[1-3.2]	[7-12]	[25-27]

Table 4: *Update time (ms) of physics world*

11. Conclusions and future work

The simulation achieves high performance in managing numerous entities across a large world through combined computational and rendering optimizations. Benchmarks reveal the CPU as the main bottleneck, highlighting the need for further improvements. Proposed solutions include simulation-based LODs—reducing update frequency for distant entities—along with optimizing distance and ray-casting calculations using approximations and lookup tables. RAM usage could be reduced by dynamically loading and unloading world data. Graphically, baking terrain meshes and textures into a single static mesh may improve efficiency. Additional systems like sound, particles, and quests also offer future optimization opportunities.

12. Acknowledgements

Grant PID2023-149487OA-I00funded by MICIU/AEI/ 10.13039/501100011033 and by ERDF/EU.

This work is part of the Project PID2022-136779OB-C32 (PLEISAR) funded by MICIU/ AEI /10.13039/501100011033/ and FEDER, EU

References

[BB22] Burger M., Bischof C.: Using instancing to efficiently render carbon nanotubes. 1

[CGF*24] Chen Q., Guo W., Fang Y., Tong Y., Lu T., Jin X., Deng Z.: A bio-inspired model for bee simulations. *IEEE Transactions on Visualization and Computer Graphics* (2024). 1

[HD04] Heok T. K., Daman D.: A review on level of detail. In *Proceedings. International Conference on Computer Graphics, Imaging and Visualization, 2004. CGIV 2004.* (2004), IEEE, pp. 70–75. 1

[Par14] Parberry I.: Designer worlds: Procedural generation of infinite terrain from real-world elevation data. *Journal of Computer Graphics Techniques 3*, 1 (2014). 1

[VM17] Vlachopoulos D., Makri A.: The effect of games and simulations on higher education: a systematic literature review. *International Journal of Educational Technology in Higher Education 14* (2017), 1–33. 1