SCÈNE 3D - RAPPORT

INF443 - Projet Scène 3D OpenGL

June 6, 2020

ALBUQUERQUE SILVA Igor, GALVÃO LOPES Aloysio
X2018

Contents

1 Abstract

This project consists in the implementation of an interactive winter 3D scene in C++ using OpenGL. The scene contains a mountain terrain, a skydome, a forest, a lake, an animated bear, animated birds and simulated snow. This document will present the ideas behind each element and give an overview of how each of them were implemented.

2 Elements

In this section we will introduce the different elements that are present in the scene. The details seen in the course will not be emphasized and the main focus will be to describe the particularities of the implemented objects, as well as the difficulties we faced. Figure 1 shows an overview of the scene and some of the elements contained in it.

Figure 1: Left: panorama of the scene. Right: models of the bird, bear, rock and trees, respectively.

2.1 Terrain

Basically we utilize an approach similar to the one seen on course: a combination of different frequencies of a Perlin noise. To add some more realism, based on [1], we were able to highlight the mountain and also add altitude erosion. To add the mountain a 2D Gaussian is summed to the lowest frequency noise and amplified by a function described in [1]. The lake is just a 2D Gaussian added at the end. The texture was adapted in Photoshop to fit specifically for this terrain, to facilitate the texture edition, a height map was exported using a simple Python script.

2.2 Skydome

The skydome's mesh was created in Autodesk Maya and exported to the scene as an OBJ file. We used this approach to correctly set the texture coordinates for each vertex in the dome, and make sure the texture would be displayed without any distortions or edges. The face's normal vectors point to the inside of the dome, to be able to look at the sun's reflections from the inside.

2.3 Forest

The forest has two different types of objects: trees and rocks. They're both generated procedurally, so that the scene contains high diversity and looks more realistic.

The tree model was based on Hewitt's work [2], which is a Blender add-on to generate trees of different species. The idea is to make the trunk and branches by defining some points and interpolate them using Bezier splines with a radius value, which is also interpolated linearly. With this, is it possible to make curved branches defining the key points.

The tree generation starts at the bottom of the trunk using a relative cursor, the "turtle". It will generate the branches recursively following the equations described in [2] to determine length, radius, curves, splits, etc. Each tree species has different parameters, so the generator has a wide range of possibilities. The bushes are also variations of the tree generator. The snow aspect was added in branches and leaves that are directed upwards, forming a small angle with the z axis.

To optimize the number of rendered faces, the leaves are made of triangles, and branches have the shape of a triangular prism. In total the four species we used, Pine, Black Tupelo, first and second bushes have around 15000, 5000, 700 and 400 vertices total, respectively.

Rocks follow a much simpler approach. They consist of semi-ellipsoids with Perlin noise. Snow was added in the faces which the normal vector makes a small angle with the z axis.

To populate the forest we divide the terrain in square tiles of fixed size. Each of these tiles have a density probability, which is read from a greyscale image file, configured for the terrain's relief. Trees and rocks are added by shuffling the tiles randomly, and iterating through them adding objects in empty tiles with a probability equal to the density of the tile, at a random position in the tile. Trees are added in a first moment and then rocks, we assure that there are no two of those elements in the same tile. Bushes are added afterwards using the same algorithm, but they can be added in occupied tiles indefinitely.

In total there are 400 trees, 800 bushes and 50 rocks in the forest. For memory and time optimization, there is a limit in the number of unique trees and bushes, which is 25. The trees are therefore translations of these unique instances. The total time for the forest generation is less than 2 seconds.

2.4 Trajectories

To make the trajectories, we created a virtual object called companion object. It is represented by a dipole, this way it generates a field that naturally leads the objects that follow this field to the companion and arriving pointing to the dipole moment vector. This way we can generate smooth trajectories to a given point pointing to a given direction.

2.5 Bear

The bear model was imported from Blender as an FBX file using the Assimp library. With this, it was possible to model its animation and fine-tune its mesh with better precision. It uses a skeleton-based animation with four bone weights for each vertex in the skin.

It was loaded from a preexisting polar bear blender model in which we drastically simplified the number of vertices, changed the texture and remade the animation.

The bear moves with a constant speed vector and the bear's normal stays aligned to the terrain's normal except around the bear's speed vector axis (this way we avoid a weird lateral turning movement).

To give a trajectory to the bear we make a companion object that was described in the trajectories section and make this companion tied to closed loop cardinal spline. This way we have a dipole tied to the spline that points in the direction tangent to the spline. The dipole is oriented towards the spline positive orientation (the cyclic spline implemented defines a positive loop direction).

Then we move the dipole at each frame to the closest distance it can get to the bear (tied to the spline) and we apply an offset in the dipole's direction. The bear moves in the direction of the field generated by the dipole and this way tries to align itself to the dipole in the XY plane. This way, when the bear gets near the dipole the dipole moves forward in the spline and thus the bear never reaches the dipole and moves in a given cyclic trajectory.

The reason to use this rather complex approach is that we can add, in addition to the dipole, the fields of positive repulsive charges in the positions of the obstacles (trees, rocks etc) and then the bear follows a spline in the XY plane but also avoid obstacles when getting too close to them. It suffices to have the dipole's charge considerably higher than the obstacle's charge to get a nice trajectory following with obstacle avoidance. In the figure 2 we show an image of a graphical representation of a companion as well as the bear following it.

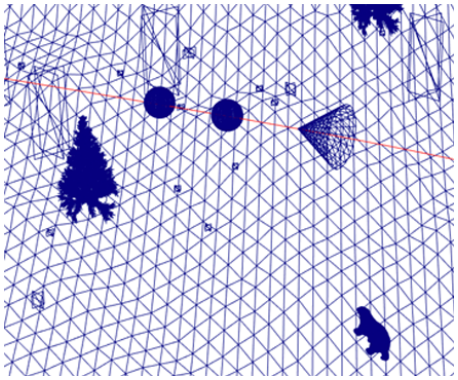

Figure 2: Bear following graphical representation of a companion (the spheres are the charges and the arrow is the orientation of the dipole).

2.6 Birds

The birds behave like a Boid and their basic implementation was based on [3]. The birds follow three basic principles: their speeds increment in the direction of the center of mass of the birds and proportionally to the distance to the center of mass, their speeds increment in the opposite direction near birds and proportionally to their distance and they try to match their speed with the other bird proportionally to their difference to the average speed.

Each one of those rules has a range of action. In addition to this, the birds max speed is limited. To assure that they stay in a certain volume their speed is increased in the opposite direction if they pass some limit. The way this counter action is made is different from limit to limit (Z positive, Z negative and XY) to assure that they won't fly too high and also that their movement is smooth.

Finally the birds model was made from scratch in Maya based on a video tutorial. The animation was made from scratch as well in the same way as the bear's. The birds texture was made from a galaxy dataset image (INF442) that surprisingly was a very good looking texture when applied with the right UV map. Similarly to the bear the bird's model was imported as FBX. The birds also change their rotation around their speed vector axis based on the radius of their XY trajectory. The birds animation vary on speed based on the angle of their speed vector with the Z axis and the radius of their XY trajectory. Most of the bird's movements are made using angular accelerations or linear accelerations to make their movement smooth.

2.7 Lake

The lake is a simple model of water with a texture applied on it and transparency. To model the water behavior the water particles are fixed in a mesh and at each step they update their position as the average of their neighbors. This way waves can propagate in the lake. To generate the waves, prior to the average, the lake

particles suffer an electrical force and move in the Z axis making their position update based in this force. This attraction is made towards particles that oscillate up and down in the lake.

Its important to say that the movement of the oscillators create energy, so we add a damping factor in the water average step, this way the waves don't grow indefinitely.

2.8 Snow

We simulated snow using particles with a simple physics model, which includes gravity and an air friction force equal to $-kv$, where k is the damping constant and v is the current particle's velocity. We introduce a uniform noise in the acceleration which is restarted every second.

These particles are contained in a small box around the camera and every time a particle leaves this box its position is randomized again inside the box, while keeping its previous velocity and acceleration. With this behavior, the snow is able to follow the camera and when the camera does not change position drastically, the snowflakes do not restart their position either.

However, we still introduced a factor of translation for the snowflakes, so that if the camera translates a certain distance, the snow will also translate a factor of this distance. We found that without this aspect the snowflakes were resetting too frequently, which was undesired.

2.9 Shadows

The shadows system uses cascading depth maps to be able to have high resolution shadows, and still use low-definition depth maps. We followed Bejarano's tutorial [4] for the implementation.

The camera frustum is split in three regions, and each has an associated depth map. Figure 3 shows a scheme of the region that is covered by the depth map on the left, and the frustum splits on the right.

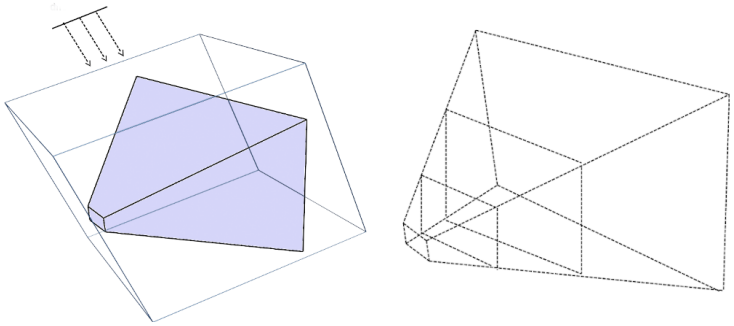

Figure 3: Left: directional light source covering the camera frustum. Right: camera frustum with 3 cascade splits.

Therefore, each light source has an associated matrix that converts a point in the global coordinate to a point inside the box covered by the depth map, in which the z coordinate represents the depth. With this, the scene is rendered to this depth map before the actual drawing, and the closest depths are saved in the depth map using a specific shader. In the actual drawing, the shader checks if this element's depth is higher than the depth map to place shadows by changing the diffuse and specular light components.

The advantages of this method is that further away objects have shadows of lower resolution, which is not noticeable because they are far away. The processing time is also not affected so drastically, because only one depth map is rendered at each frame, not all three of them (so the shadows are actually at a third of the scene's frames per second).

2.10 Billboards

For optimization purposes, each tree and rock has a billboard associated with it. These billboards are displayed in the place of these objects when their size in the screen dimensions is small (we used a threshold of 10% of the screen dimension). This provides an indispensable sparing for the gpu, because we are able to avoid rendering most of the objects in the scene.

To generate these billboards, a virtual camera is used to take two snapshots of each object in perpendicular angles, at a certain distance, and save these snapshots in textures. Figure 4 shows the result of this algorithm.

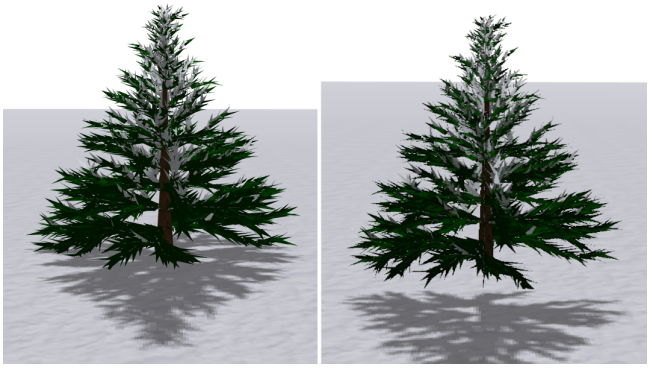

Figure 4: A Pine tree and its associated billboard.

We have also used bounding spheres to detect which objects are inside the camera frustum at every frame and avoid drawing those that are not.

2.11 Camera movement

The camera follows different companions (charged dipoles as explained in the trajectories section). We defined three types of companion for the camera trajectory: one follows a cardinal spline trajectory and aligns itself tangent to the trajectory, one hovers around the bear pointing towards it and the last one follows the bird pointing in the direction of the bird's speed vector.

The camera selects the current companion to follow and gives the companion a signal to begin its trajectory following. Then the camera follows the field lines generated by the companion to reach its position aligned to it. When the camera gets too close to the companion its speed becomes proportional to its distance to the companion, this way, the camera never gets to the negative charge and we avoid unpredictable behavior.

We used this approach to make sure that the transitions between trajectories are smooth. We highlight also that speed changes in the camera are made using accelerations to assure smooth movements. Finally we made specific tweaks on some parts: at the bear hovering we disabled the field based approach when too close to the bear and in the bird following we added a new rotation to follow the birds rotation around its speed axis.

3 Conclusion

We were able to create a complex scene that looks realistic. It runs at 15-60 frames per second in a MacBook Pro 2018 13', depending on how many objects are on the screen. Many implementations came from extensive research of different methods, and a lot of new modeling techniques were learned in this project.

References

[1] Gabriel Costa Backes. "Real-Time Massive Terrain Generation using Procedural Erosion on the GPU". In: (2018), p. 4.

[2] Charlie Hewitt. "Procedural generation of tree models for use in computer graphics". Trinity Hall, May 19, 2017. URL: https://chewitt.me/Papers/CTH-Dissertation-2017.pdf.

[3] *beneater/boids*. URL: https://github.com/beneater/boids (visited on 06/05/2020).

[4] Antonio Hernández Bejarano. *Cascaded Shadow Maps*. Jan. 2020. URL: https://ahbejarano.gitbook.io/lwjglgamedev/chapter26.