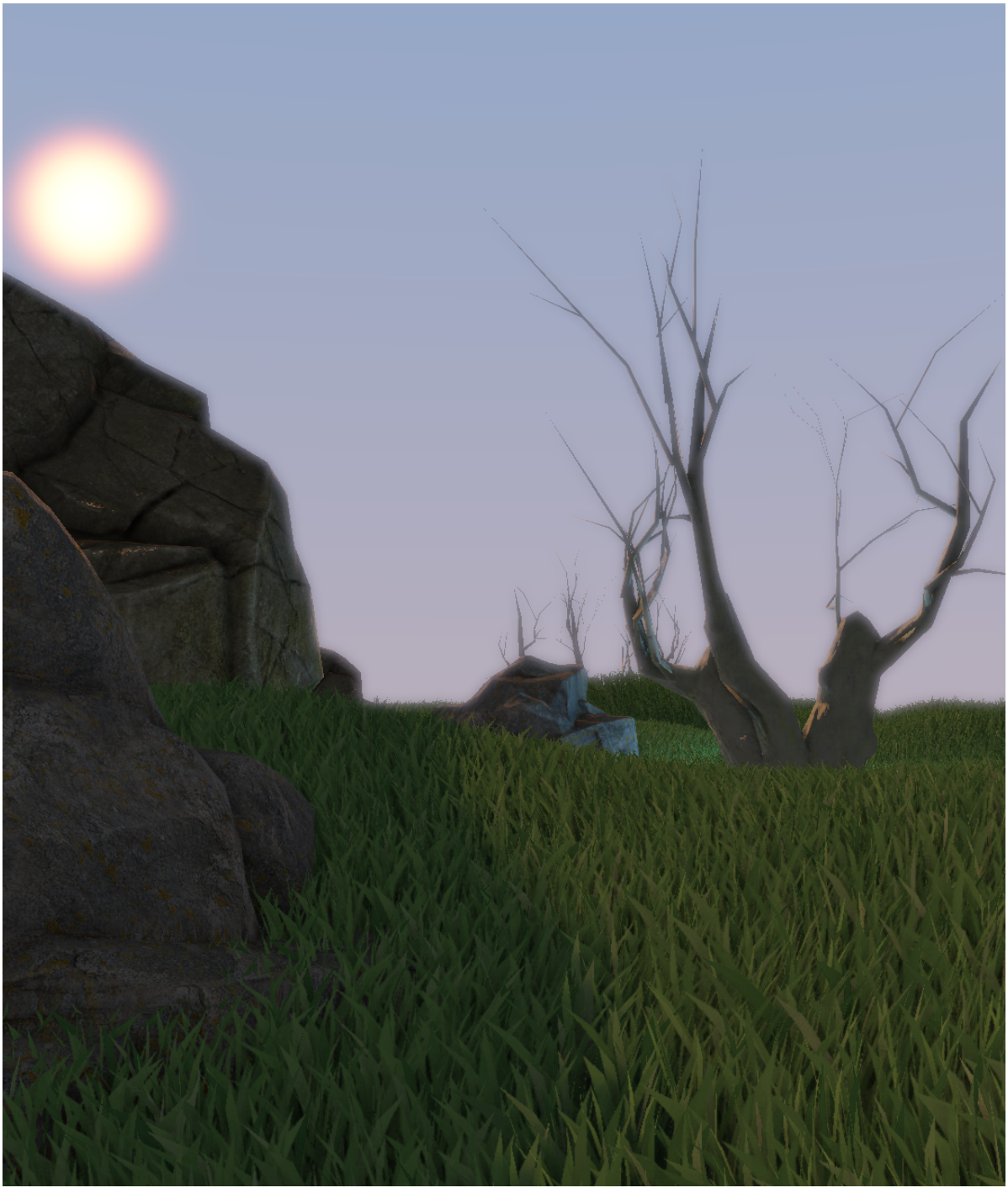
TSBK03 - Project

Hans-Filip Elo (hanel742), Lage Ragnarsson (lagra033) and Isak Wiberg (isawi527)

December 2016



1 Introduction

We have continued to work on a 3D graphics engine that we call "Emerald Engine" (Emerald). Emerald was initially created during the course TSBK07, and we've continued our work from there. One can follow the work done throughout the project by looking at the different sprints that were defined, and the corresponding issues related to them, on Github. From these issues one can look at the related pull request (PR) that closed the issue in order to see the specific code for that feature.

The original specification of the project can be found at Github. The project specification were divided into two parts, one "will do" part and one "might do" part.

The reader of this document is expected to have some knowledge of the OpenGL rendering pipeline[1].

1.1 Will do features

The features we wanted to prioritize were:

- Terrain generation from a heightmap
- Procedurally generated grass with geometry shader (GS)
- Animation of generated grass
- Directional light source (sun) which casts shadows with shadow mapping
- Performance optimizations.

The main performance enhancements done to Emerald are listed on the first sprint of this project, see that Github milestone for further reference.[2]

1.2 Might do features

The original "might do" features were:

- Shadow maps for point lights
- Grass animation based on nearby objects / collision detection
- Different level of detail (LOD) algorithms

Of these features, collision detection for grass and different LODs for grass were implemented.

We also decided to create a multi-mesh terrain. The purpose of the multi-mesh terrain is to work with the frustum culling which has been expanded to now cull terrain meshes and individual meshes of models as well. It was also necessary for the way we implemented grass LOD.

[1] OpenGL rendering pipeline, https://www.opengl.org/wiki/Rendering_Pipeline_Overview

[2] Sprint 1: https://github.com/lragnarsson/Emerald-Engine/milestone/2?closed=1

2 Background

The background for the technologies implemented in the project are described in this section.

2.1 Grass Generation

There are many ways to create grass in a 3D scene. You can use billboards or predefined patches of grass as models. Another way is to generate the geometry during runtime in the geometry shader stage. Generating grass this way is not the best in terms of performance but it's very flexible and great for experimentation.

The inspiration for testing this came from a blog post[3] describing a GS approach to grass rendering in the 3D engine Outerra.

2.1.1 Grass Animation

In order to make the grass come to life we wanted it to move in a believable fashion. Swaying in the wind was the first part of this. We wanted to get both high and low frequency noise affecting each individual blade of grass. But also to have areas of the field moving somewhat together to mimic gushes of wind.

Having the grass react in response to nearby objects can become a difficult and expensive task if the goal is correct collision detection. In our case it was enough to create a repelling force from the center of each model which affects all nearby blades of grass.

2.1.2 Barycentric Coordinates

For grass distribution within a triangle, barycentric coordinates are used. A point P inside the triangle defined by the points $\{P_1, P_2, P_3\}$ is given by

$$P = P_1 + u\ \overline{P_1P_2} + v\ \overline{P_1P_3}, \tag{1}$$

where $0 \leq u, v \leq 1$ and $u + v \leq 1$.

2.1.3 Grass LOD

To allow for large amounts of grass it is desirable to lower the detail level on grass that is further away. Our approach to grass generation gives us many possibilities in this respect, and we have chosen to create three levels of detail (LODs) with decreasing number of grass blades per triangle, and decreasing detail level on each blade of grass.

[3] Procedural grass rendering http://outerra.blogspot.se/2012/05/procedural-grass-rendering.html

2.2 Shadow Mapping

Figure 1: A shadow map rendered to screen in Emerald.

Shadow mapping is a technology where you first render the scene with depth buffer only from the light source point of view. Each fragment within this depth buffer is then, during the regular render pass, transformed into view space of the camera or world space depending on implementation. If the depth of a fragment within the camera view space is the same as the shadow buffers depth, then the fragment is lit. If the fragment depth differs between the shadow buffer and the camera view space, then the fragment is within shadow. The resulting image from this comparison is called a shadow map. In Figure 1 a shadow map can be seen. For more information on Shadow mapping, please see the course material[4].

[4]Ragnemalm I, So how can we make them scream, page 43-49, http://www.computer-graphics.se/TSBK03-files/SHCWMTS-2016.pdf, fetched 2016-12-27

Figure 2: Results from shadow mapping before smoothing.

2.2.1 Shadow Smoothing

When using shadow mapping one usually gets jagged edges of the shadows. This effect will go away with a high enough rendering resolution for the shadow map, but that quickly gets expensive to compute. In order to address this issue one can smoothen the shadows by using a low-pass filter kernel to smooth the addition of shadow in each fragment. This results in smooth shadows at lower rendering resolutions. This can be seen in Figure 3.

Figure 3: Results from shadow mapping after smoothing.

3 Implementation

For implementation details on Emeralds original features, see the report from the course TSBK07.[5]

3.1 Grass Generation

One of the largest parts of this project was to generate a large field of animated grass. The steps we took to do this is described in this section.

Grass generation is almost entirely done on the GPU in shader code, and only set up by the CPU. The parts of the CPU code that does something with grass generation are:

Class	Purpose
Mesh	Emeralds smallest geometry component. Subpart of a model or terrain.
Terrain	Container class for all meshes of the terrain. Grass are generated on the terrain meshes.
Renderer	Handles rendering pipeline and calls grass shader.

3.1.1 Grass Distribution

Our approach to grass distribution was to set a specific amount off grass blades on each triangle in the terrain mesh. This means that the grass density depends heavily on the size of the triangles in the terrain mesh but since the triangles are evenly distributed in a regular pattern and with similar size it works quite well.

Another way could have been to choose the number of grass blades dynamically based on the area of the triangle but since the max amount of output vertices from the geometry shader must be specified we opted for a fix amount of grass blades per triangle. It was also found that setting the max amount of output vertices higher than what was actually output had a negative impact on performance, so therefore the choice seemed to fit well with how geometry shaders were optimized.

Within each triangle the grass blades are placed using barycentric coordinates defined in Section 2.1.2. The coordinates was predetermined in a regular pattern which can be seen in Figure 4. Figure 5 shows the grass field with grass distributed in this way. There are clear patterns in the grass field, so therefore noise to the positions and rotations was added.

[5] Elo H-F, Ragnarsson L, Wiberg I, TSBK07 Project report, May 2016.

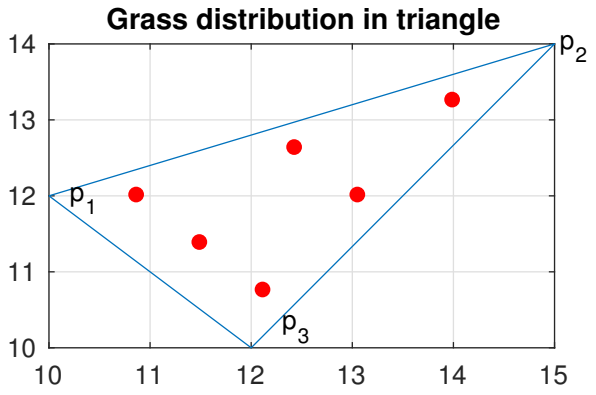


Figure 4: Grass distribution inside triangle. p_1, p_2 and p_3 are the three vertices that forms the triangle.

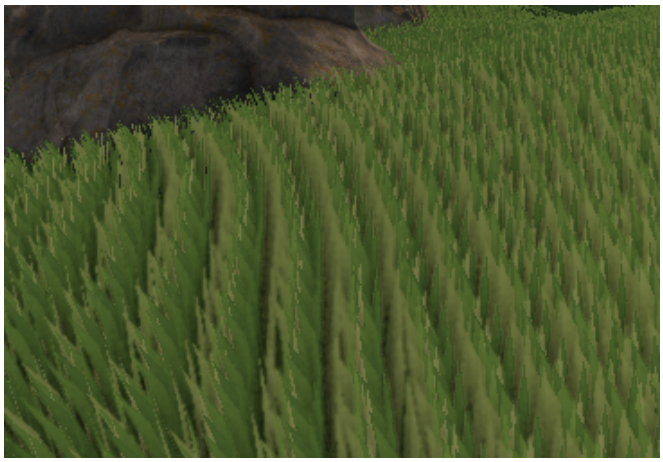


Figure 5: Screenshot showing the grass field without noise added to the grass blade positions and rotations. Note the regular pattern and that all grass blades are oriented in the same plane.

Zero-mean noise is added to both the u and v coordinate of each grass position which essentially gives a random vector in the triangle plane like

$$\bar{x} = e_1 \overline{P_1 P_2} + e_2 \overline{P_1 P_3}, \tag{2}$$

where e_1 and e_2 are the random numbers with zero mean. The grass blade position is then moved in the direction of this random vector. The random vector also defines the rotation of the grass blade.

An important part of our grass distribution approach is the random number generation. The randomness of the grass field should not change over time and it should only change with the world coordinates. We chose to use the texture coordinates in the terrain mesh as seeds to a pseudo-random function[6], and it seems to work very well.

[6] GLSL rand http://byteblacksmith.com/improvements-to-the-canonical-one-liner-glsl-rand-for-opengl-es-2-0/, fetched 2016-12-31

3.1.2 Grass Shape

A grass blade is constructed from a base vector in the triangle plane, a start position inside the triangle, the world space up vector and the up vector of the terrain mesh triangle. The base shape of the grass blades is two-dimensional and is made up by a set of vertices that define the grass blade. We use two different kinds of grass blades, one long and thin and a shorter and thicker double grass blade. These can be seen in Figures 6 and 8. To reduce the amount of geometry that is created we have simplified the grass somewhat for grass that is further away. This was done by removing vertices while preserving the overall shape of the grass blades as much as possible. The simplified grass blades can be seen in Figures 7 and 9.

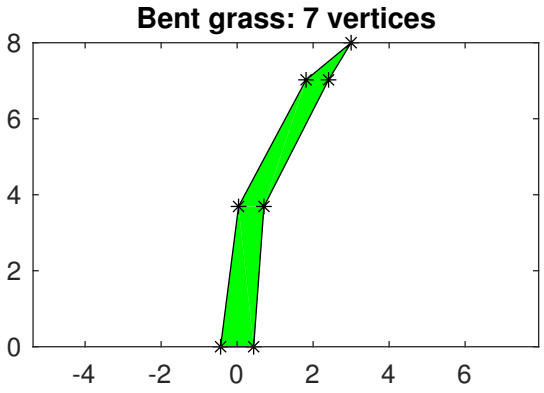


Figure 6: Bent grass with 7 vertices. Used in highest detailed LOD.

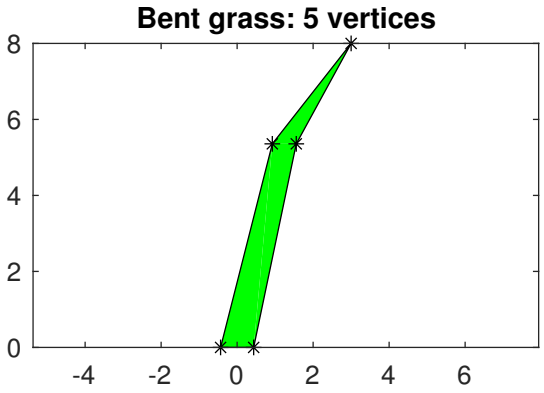


Figure 7: Bent grass with 5 vertices. Used in less detailed LODs.

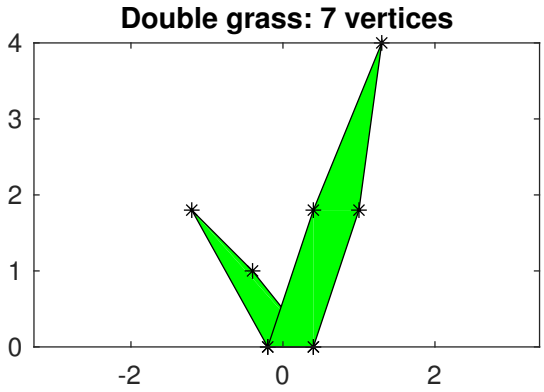


Figure 8: Double grass with 7 vertices. Used in highest detailed LOD.

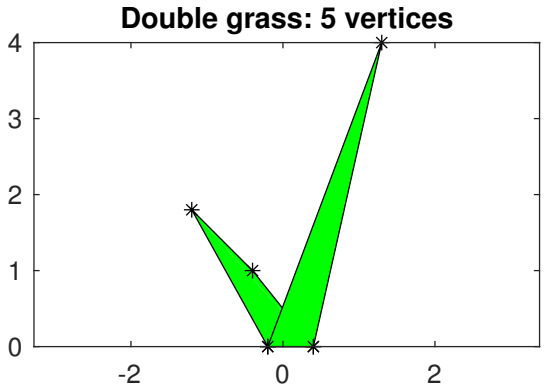


Figure 9: Double grass with 5 vertices. Used in less detailed LODs.

The number of grass blades created on each terrain mesh triangle is also reduced in LODs further away from the camera. The detail level of the three different LODs are presented in the table below. LOD1 is closest to the camera and LOD3 is farthest away.

Detail level	**Grass blades per triangle**	**Vertices per grass blade**
LOD 1	6	7
LOD 2	4	5
LOD 3	1	5

The choice between the different LODs is taken based on the distance between the camera and the center of every specific terrain mesh. The distances for the three LODs can be changed during runtime. Figure 10 shows the three different LODs with grass blades colored differently in the three LODs.

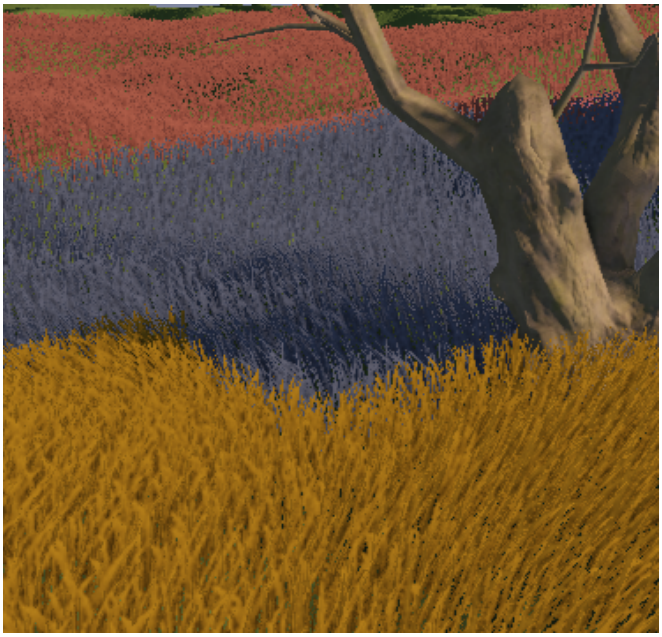


Figure 10: Differently colored grass in the three LODs. The terrain mesh size can be seen clearly in the border between the yellow and blue area.

3.1.3 Grass Animation

When a blade of grass has been placed in the terrain its shape is modified based on wind and nearby objects. Both of these effects are represented by translation vectors affecting each vertex. They are calculated independently, summed together and then projected into the plane of the terrain triangle the grass is on. This is done to prevent the grass from stretching along its height. When animating a blade of grass like this the bottom two vertices should always remain stationary to prevent it from sliding around on the surface. In addition we wanted the topmost vertices to be more affected to make the grass bend and not just sway. To achieve this we interpolate the translation vector $\bar{t}$ based on the vertex height. If one blade of grass contains n vertices then the translation vector, $\bar{t_i}$,for vertex i can be calculated using its height h_i:

$$\bar{t_i} = \left(\frac{h_i}{h_n}\right)^{1.5} \bar{t} \tag{3}$$

We raise the fraction to the power of 1.5 to increase the effect of the bending near the tip of the grass.

The translation vector from the wind should for a fixed point in the world only vary based on time. We use a normal map texture commonly used for water to represent this translation after it has been translated and scaled to represent an offset such that each vector component satisfies $t_x, t_y, t_z \epsilon [-1, 1]$. See figure 11. The source of the normal map used is unknown but it circulates on several Internet forum as an example texture for water shading. The lower frequency noise in the texture will make each blade of grass vary slightly compared to its neighbours and the lower frequency noise will nicely emulate the effect of gushes of wind affecting a larger area.

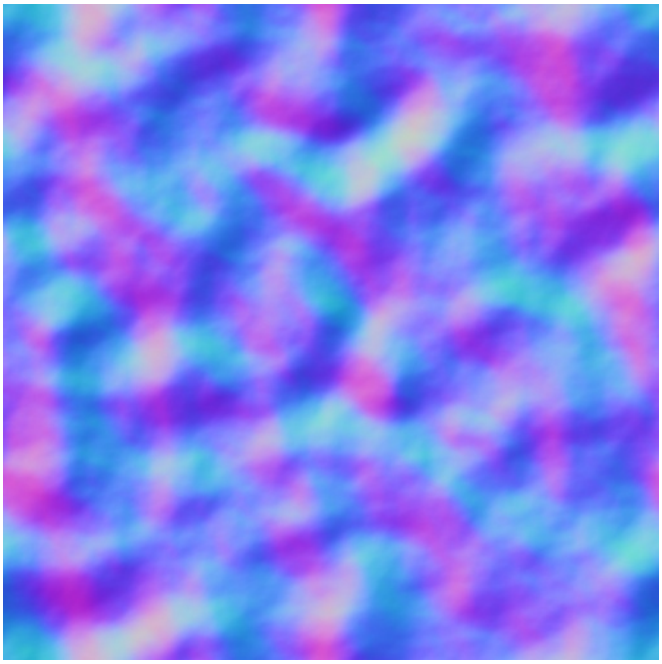


Figure 11: Normal map used for wind animation.

The normal texture is indexed with the texture coordinate of the terrain where the grass is growing from. Then a time based offset, which is the same for all grass, is added. This is what gives it the time varying wind animation.

To make the grass interact more with the rest of the world we uploaded bounding spheres for each model to the grass shaders using a UBO. These are the same bounding spheres used for frustum culling so we just had to upload an array of the bounding spheres currently in the viewing frustum. In the geometry shader for the two closest LODs we calculate a translation vector for each triangle in the terrain in order to avoid doing it for every single blade of grass. The repelling "force" affects all triangles which has their grass collision point within the sphere and the translation vector calculated is proportional to the distance in such a way that it's zero at the boundary and maximum at the center of the sphere. The grass collision point for each triangle is at grass height above the center of the triangle to approximate the effect of colliding with the tip of each blade of grass. See figure 12 for the resulting effect. Note that we artificially increased the bounding sphere radius of our glowing orbs to exaggerate the collision effect for demonstration purposes.

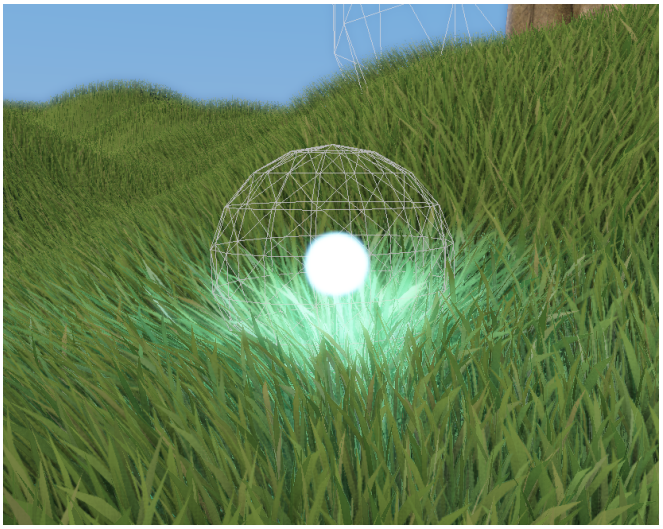


Figure 12: A glowing orb with its bounding sphere visualized interacting with the grass.

3.1.4 Terrain Generation

Emerald classes involved in terrain generation:

Class	Purpose
Mesh	Emeralds smallest geometry component. Sub-part of a model or terrain.
Terrain	Container class for all meshes of the terrain. Loads a PNG from disk and generates sub-meshes.
Loader	Loads an emerald scene file which specifies PNG and scales to use.

The terrain generation is based on an implementation by Ingemar Ragnemalm from the course TSBK07[7].

When generating terrain we read a PNG-image from the filepath given in a Emerald scene file. Within the scene file, height scale, denoted as S_h, and planar scale, denoted as S_p, are also defined. The length of the sides of a sub-mesh, denoted as l_m, is also defined here.

From the PNG-image each pixel in the image, seen as the xz-plane, maps to a vertex within the Terrain model, in model coordinates, according to:

$$x = pos_{width} \tag{4}$$

$$y = \frac{\sum_{i=1}^{n} channelvalue_i}{n} \tag{5}$$

$$z = pos_{height} \tag{6}$$

Where *pos* consists of the pixel position in 2D image coordinates and y is the mean of the pixel channel values for that position.

[7]Ragnemalm I, Lab 4 Interactive terrain, http://www.computer-graphics.se/TSBK07-files/lab2015/lab4.html, fetched 2016-12-23

The terrain is, as earlier mentioned, generated into sub-meshes with the shape of squares. Each side within a sub-mesh is $l_m + 1$ in order to get a terrain without gaps in it. The overlapping vertices leads to exponentially larger memory use when shrinking the sub-mesh size - so this should be done with care.

3.2 Skydome and Sun

In order to demonstrate our dynamic shadows we needed a moving sun. Instead of using a predefined skybox with a sun on it we created a simple dynamic skydome with a sun that moves across it. The skydome is a unit sphere in the origin of view space. The inside of it is drawn at the end on all fragments not yet drawn which means we change the cull face to the front side when drawing the sky. The result can be seen in figure 13.

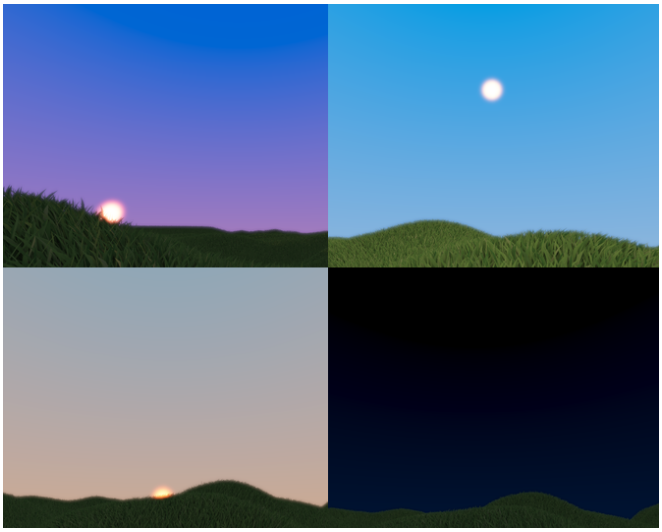


Figure 13: Skydome at different times during the day. From the top left we have dawn, mid day, dusk and midnight.

The appearance of the sun and sky is controlled by three different color values: the sky horizon color, sky zenith color and the sun color. The sun color is simply the color of the sun. The sky zenith color is the color of the sky straight up and the sky horizon color is the color of the sky at the horizon. When drawing a fragment on the inside of the skydome the color of that fragment is decided by interpolating between the zenith and horizon colors based on the height over the horizon. We use a non-linear interpolation to make the horizon color more local to the horizon. The sun is drawn in a similar way as specular highlights in the Phong shading model. The current sun position is uploaded as a uniform and the dot product between it and the fragment position is raised to an arbetrary power to get a nice falloff effect.

To get shifting colors throughout the day for both the sun and the sky we have specified four variants of the three different colors resulting in 12 colors in total. These variants are for dawn, mid day, dusk and midnight. The actual sun,

sky zenith and sky horizon colors that are uploaded as uniforms are linearly interpolated between these four variants based on the time of day. This means the sun and sky will slowly shift colors during the day-night cycle. The color used for the sun light source is the same one used to draw the sun in the sky which means the color of the scene will also vary based on time of day.

3.3 Shadow Mapping

Emeralds shadow mapping is loosely based on an implementation from Learn OpenGL[8] but is somewhat changed in order to work with the deferred rendering pipeline.

Emerald classes involved in shadow mapping:

Class	Purpose
Skydome	The Emerald skydome. Contains the directional light (sun).
Renderer	Controls the Emerald render flow.

In shadow mapping the depth buffer rendered from the sun is created first. This is done by attaching a frame buffer object (FBO) containing only a depth buffer to a texture. We then use this FBO as our frame buffer for the shader that renders the shadow map. The shadow map is rendered with an orthogonal frustum instead of a normal viewing frustum. This is done since the light source is a sun, and thereby a directional light source. For a point-light source a regular viewing frustum should be used.

The shadow buffer is then attached to a texture unit in the deferred stage. The matrix which transforms from the light source's view space to the cameras view space is uploaded as a uniform to the deferred stage. These are then used to calculate the depth (z-component) in each fragment, which is then compared to the depth in current view.

The depth comparison is then done for a square of 3x3 fragments, with the current fragment in the middle, where each fragment contributes to the shadow in the current fragment an equal amount. This gives the smooth shadow seen in Figure 3.

4 Problems

A brief description of some issues encountered during the project.

4.1 UBOs on macOS

Using a Uniform Buffer Object (UBO) for bounding spheres of objects within the world used to interact with the grass does not work on macOS with Intel graphics (NVIDIA and AMD not tested). It is unclear why this happens because we use a similar method for representing our light sources but we get strange artifacts in the grass when using this feature on macOS. It could be due to something we're doing which is not in the OpenGL specification but still supported in some implementations. Intel and NVIDIA GPUs on Linux does not show this problem.

4.2 Performance

During this project the performance requirements of the engine has increased. In order to get decent performance on integrated Intel chips the viewport resolution and scene complexity has to be lowered. We have done some minor profiling and the slowest parts seem to be the geometry pass and the SSAO pass which is most likely due to a high triangle count. The situation could be improved by using terrain and model LODs and smarter culling.

5 Conclusions and Future Work

We are happy with the results of this project and we enjoy seeing our graphics engine grow in terms of features. As we have been working with this engine since we started learning about computer graphics there are many parts of it we know can be improved greatly. However, adding many new features and experimenting with them has been our main priority and as a result we have had the chance to learn a lot of different things.

Some of the things considered for development that *might* be considered for implementation in the future are:

- Motion Blur
- Depth of field effects
- Screen spece reflections
- Expand the deferred rendering to tile based
- Some form of scripting support
- Integrate some form of CPU based physics engine

All of these are listed as issues at the Emerald Github page[9].

[8] Learn OpenGL, Shadow mapping, https://learnopengl.com/#!Advanced-Lighting/Shadows/Shadow-Mapping, fetched 2016-12-27

[9] Emerald issues, https://github.com/lragnarsson/Emerald-Engine/issues, fetched 2016-12-30