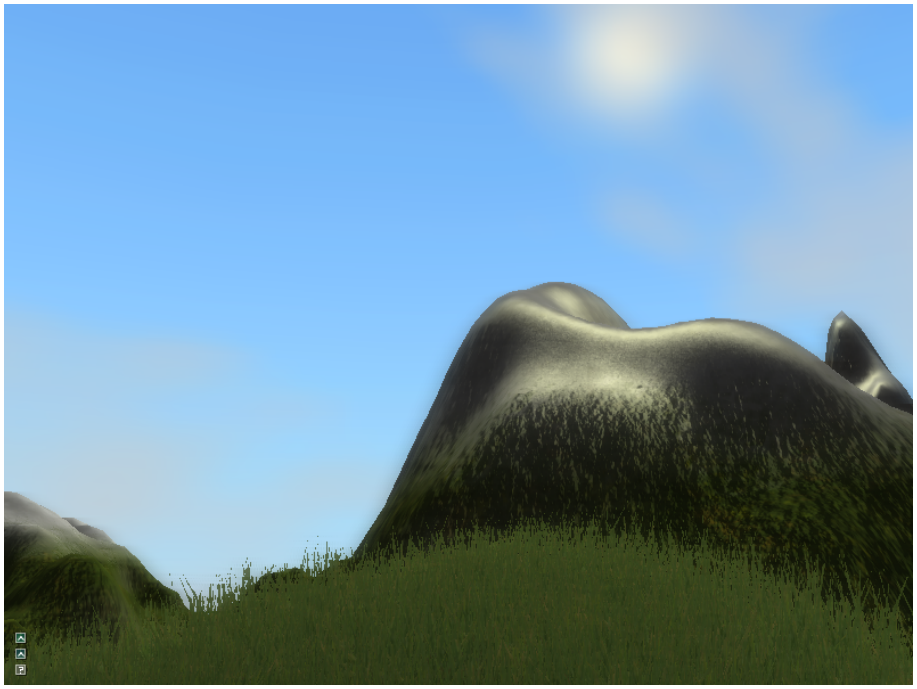
Rendering natural looking outdoor environments

Couse: Introduction to Computer Graphics, spring 2010



Asger Dam Hoedt
asgerhoedt@gmail.com
20051770

Christian P. V. Christoffersen
cpvc@cs.au.dk
20050879

Department of computer science
Aarhus Universitet

Contents

Chapter 1

Introduction

This report is the product of the second part of the course: Introduction to Computer Graphics, followed by the authors during the spring of 2010 at the department of computer science at Aarhus University. The focus of this second part of the course is OpenGL's programmable pipeline and OpenGL extensions.

To utilize the power of these advanced rendering technologies we have chosen to try and make a realistic looking outdoor environment. The project is based on a previous OpenEngine project, but most of it has been rewritten as part of this project.

1.1 Setting the scene

When creating a scene in computer graphics the director, as when making a movie or directing a play, writes a script. This script includes descriptions of the mood and the setting where the story unfolds. This report describes techniques for rendering a scenery where the authors had the following setting in mind. The scene consists of an island surrounded by a calm ocean as far as the eye can tell. The island has sandy beaches, fields of grass waving gently in the breeze, and tall mountains with pure white snow on the top. Clouds are drifting on the breeze and in the horizon the sun is setting, casting it's last golden rays toward the landscape.

1.2 Previous work

This project is build on top of a previous heightmap project in the OpenEngine framework. While that old project's only purpose was to explorer heightmap techniques, this project has been about making the environment more aesthetically pleasing. A lot of effort have therefore gone into the shaders, both with respect to creating a more realistic result and optimizing them, this means that none of the old shader code remains. Most of the code behind the heightmap scene node have also been completely rewritten to better facilitate extensions made in this project and for future extensions.

The water used in this project was made by the authors as part of the old heightmap extension, but contains most of it's original code, apart from using the new `FrameBuffer` abstraction.

Apart from the terrain, we have extended the scene with an animated sky dome, waving grass and post process effects.

1.3 OpenEngine

The work presented in this report is build on top of the OpenEngine framework. This framework is an open source project that was started at Aarhus University in the spring of 2007, and was original build to teach the course Computer Game Development, CGD, the same year. The authors of this report are both deeply involved in the ongoing development of OpenEngine, and have been so since its beginning.

OpenEngine is a framework for rendering 3D scenes using a scene graph. For our project we have mostly implemented new scene graph nodes, such as `HeightmapNode`, `MeshNode` or `PostProcessNode`, where the two latter were developed during the process of creating this project and have made their way

into the core framework. We use the two existing nodes; `TransformationNode`, for translating the sky dome, and `BlendNode`, to make the sky dome transparent.

OpenEngine is composed of a core framework, extensions, and projects. In this project we have made use of some already available extensions:

- `SDL`, to create a window and get an OpenGL context.
- `GenericHandlers` for keyboard and mouse event handling.
- `CairoResource` and `HUD` to draw and render the FPS counter on top of the rendered image.
- `AntTweakBar`, `Inspection`, and `InspectionBar` to add graphical components, allowing us to change values of variables at run-time.

The following extensions are also used in our project and have either been created or extended during the implementation of our project:

- `OpenGLRenderer` which handles all interaction with OpenGL inside the OpenEngine framework. This includes the allocation of *textures*, *vertex buffer object* and *frame buffer objects* on the graphics card and a `RenderingView` that traverses the scene graph and renders it. During this course the OpenEngine core and this extension have been extended with buffer object support, three dimensional textures, new GLSL shader implementation and our `PostProcessNode`.
- `FreeImage` was wrapped in an extension and enables us to load and save `PNG`, `TGA` and `EXR` images,
- `MeshUtils` which was created as part of the new geometry structure to handle buffer objects. It creates the geodesic spheres used for the sky domes.
- `TexUtils` which includes our noise generation algorithm described in chapter 5,
- `OpenGLPostProcessingEffects` which holds shaders for some of the effects described in chapter 6 and a lot of other effects.
- `Heightmap` which implements a heightmap node and a grass node. This extension also includes an expansion to the `RenderingView` in `OpenGLRenderer`, which can render the heightmap and grass.

All of these extensions are used by our `Terrain` project, which sets up the scene and holds modules for handling logic, like passing time of day or sun direction to shaders.

To utilise OpenGL's programmable pipeline we chose to use the OpenGL Shader Language, GLSL, instead of NVIDIA's CG. This was partly due to our previous experience with GLSL and partly because a lot of work had gone into optimizing and extending the OpenEngine GLSL shader implementation in the first quarter of the course.

1.4 OpenGL version and extensions

Since every new version of OpenGL is composed of a set of extension, OpenGL now consists of quite a large number of these extensions. Therefore instead of listing every extension used in this project, OpenGL 2.1 is assumed and only extensions not in this version will be listed. The extensions used in our project are described in appendix A

Chapter 2

Terrain

The terrain itself is generated from a heightmap and is a continuation of previous work, as already stated in the introduction. How a heightmap is loaded from a texture and converted into geometry is pretty straight forward, and thus won't be the focus in this chapter. Instead we will focus on the lighting model and how it was derived from the general model presented in [AMHH02, page 83]. We will talk about how we give the illusion of more detailed geometry by implementing *bump maps*, which requires us to transform a bumped normal from *tangent space* and into *world space*, where the lighting calculations are done. Finally we will explain how *texture arrays* where used to cut back on *texture unit* usage and texture lookups when rendering.

However before we start, we must briefly touch upon the subject of *geomorphing* to understand part of what is going on in the vertex shader.

The heightmap uses cluster-based Continuous Level of Detail to minimize vertex processing overhead. An example can be seen in figure 2.1 where a mound is rendered with three different levels of detail, LOD. When changing level of detail, the terrain will appear to suddenly deform, this is termed *Vertex Popping*. Geomorphing, as presented by [Wag03], can be used to remove this popping by gently morphing in the new vertices. The technique is rather simple. Instead of letting a vertex pop into it's actual position when changing LOD, the vertex is created on the line between it's two visible neighbours and then morphed into its actual position. Because our terrain is a heightmap, geomorphing only needs to be done along the y-axis.

Vertex popping has now been removed, but all other vertex attributes must be morphed as well to remove texture popping and normal popping. Our heightmap texture coordinates are proportional to the vertex xz-positions and will therefore not pop when changing LOD. But the adding or removing of normals will cause popping in the lighting. We could fix this by morphing the normals, but instead we've used a *normalmap* to store the normals in texture and look them up in the fragment shader. This has the added benefit that we will retain our light shading, even when the geometric detail is reduced. The downside is that it forces us to use per pixel lighting, since the normal is not available until the fragment shader.

We assume that the terrain is not rotated, scaled or translated, so all our vertices and normals are given directly in world/model space. This is done to simplify some of our calculations and save a normal transformation in the fragment shader, but the theory can easily be extended to transformed geometry.

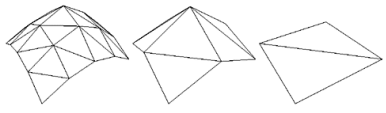


Figure 2.1: A wireframe mound rendered with three different levels of detail. The new vertices introduced in higher levels of detail are the ones causing vertex popping.

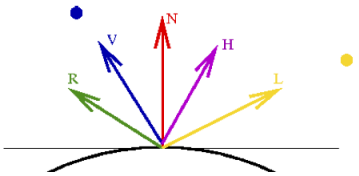


Figure 2.2: The vectors used in lighting calculations

2.1 Lighting

Because we are using shaders for rendering, we need to implement a light model. The general lighting equation for one light source[1] states

$$\mathbf{i}_{tot} = \mathbf{a}_{glob} \otimes \mathbf{m}_{amb} + \mathbf{m}_{emi} + c_{spot}(\mathbf{i}_{amb} + d(\mathbf{i}_{diff} + \mathbf{i}_{spec}))$$

Since the scene is set outside in a natural environment the only light source is the sun, which is a directional light. This allows several aspects of the lighting equation to be ignored. First of all since there is only one light source in the scene, the global ambiance can be taken into account in the light source's ambiance, removing that expression from the equation.

$$\mathbf{i}_{tot} = \mathbf{m}_{emi} + c_{spot}(\mathbf{i}_{amb} + d(\mathbf{i}_{diff} + \mathbf{i}_{spec}))$$

Also because the light source is a directional light, the spotlight factor, c_{spot}, and attenuation factor, d, can safely be removed from the expression.

$$\mathbf{i}_{tot} = \mathbf{m}_{emi} + \mathbf{i}_{amb} + \mathbf{i}_{diff} + \mathbf{i}_{spec}$$

Lastly the emission factor is 0, since none of our surfaces emit light, so we end up with

$$\begin{aligned} \mathbf{i}_{tot} &= \mathbf{i}_{amb} + \mathbf{i}_{diff} + \mathbf{i}_{spec} \\ &= \mathbf{m}_{amb} \otimes \mathbf{s}_{amb} + (\mathbf{n} \cdot \mathbf{l})\mathbf{m}_{diff} \otimes \mathbf{s}_{diff} + (\mathbf{v} \cdot \mathbf{r})^{m_{shi}}\mathbf{m}_{spec} \otimes \mathbf{s}_{spec} \end{aligned}$$

where the following three formulas for ambient, diffuse and specular lighting where used.

$$\begin{aligned} \mathbf{i}_{amb} &= \mathbf{m}_{amb} \otimes \mathbf{s}_{amb} \\ \mathbf{i}_{diff} &= (\mathbf{n} \cdot \mathbf{l})\mathbf{m}_{diff} \otimes \mathbf{s}_{diff} \\ \mathbf{i}_{spec} &= (\mathbf{v} \cdot \mathbf{r})^{m_{shi}}\mathbf{m}_{spec} \otimes \mathbf{s}_{spec} \end{aligned}$$

where $\mathbf{n}$ is the surface normal, $\mathbf{l}$ is the direction of the light, $\mathbf{v}$ is the vector pointing to the camera from the surface point and $\mathbf{r}$ is the reflection of the light around the normal. All the vectors are assumed to be normalized and can be seen on figure 2.2.

The contribution of each factor to the final image can be seen in figure 2.3.

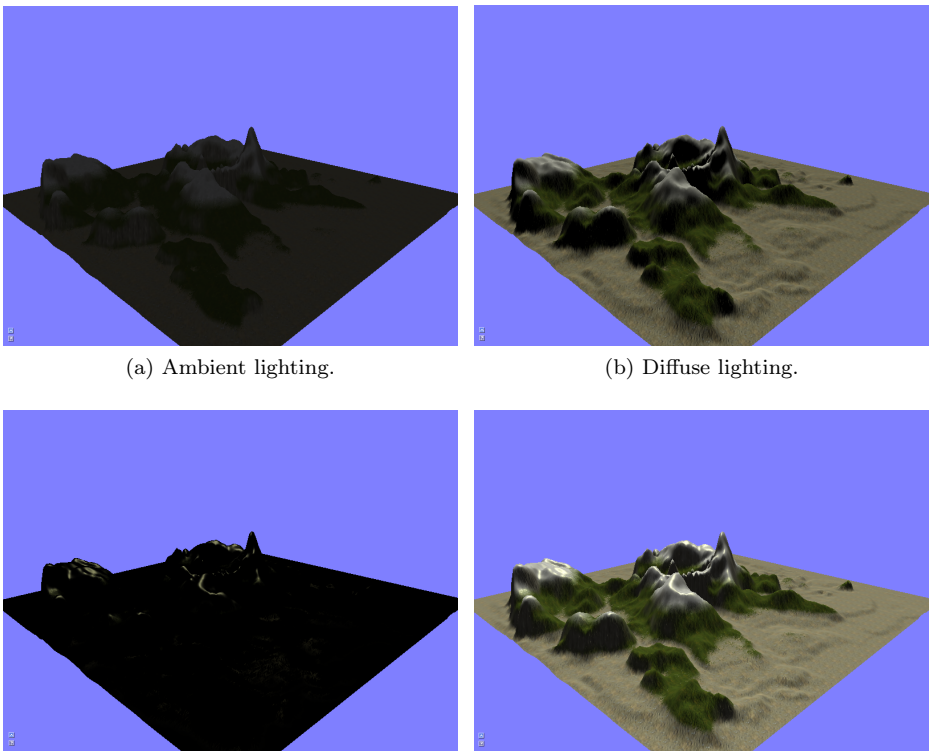
Now if we restrict our material to only specify one color and a specular intensity instead of a separate color for ambient, diffuse and specular, we can reduce the equation to

$$\begin{aligned} \mathbf{i}_{tot} &= \mathbf{m}_{color} \otimes \mathbf{s}_{amb} + (\mathbf{n} \cdot \mathbf{l})\mathbf{m}_{color} \otimes \mathbf{s}_{diff} + (\mathbf{v} \cdot \mathbf{r})^{m_{shi}}\mathbf{m}_{color} \otimes \mathbf{s}_{spec} \\ &= \mathbf{m}_{color} \otimes (\mathbf{s}_{amb} + (\mathbf{n} \cdot \mathbf{l})\mathbf{s}_{diff} + m_{intensity}(\mathbf{v} \cdot \mathbf{r})^{m_{shi}}\mathbf{s}_{spec}) \end{aligned}$$

This restriction is more physically correct, in the sense that only the color of the material is now used with respect to the different light components, but is also more restrictive to artists.

In the above formulas the specular lighting has been given by the *Phong lighting equation*[2], $\mathbf{i}_{spec} = (\mathbf{v} \cdot \mathbf{r})^{m_{shi}}\mathbf{m}_{spec} \otimes \mathbf{s}_{spec}$, where $\mathbf{m}_{spec} = \mathbf{m}_{color}$ and $\mathbf{r} = 2(\mathbf{n} \cdot \ \mathbf{l}) - \mathbf{l}$.

[1] [AMHH02, page 83]
[2] [AMHH02, page 76]



(a) Ambient lighting. (b) Diffuse lighting.

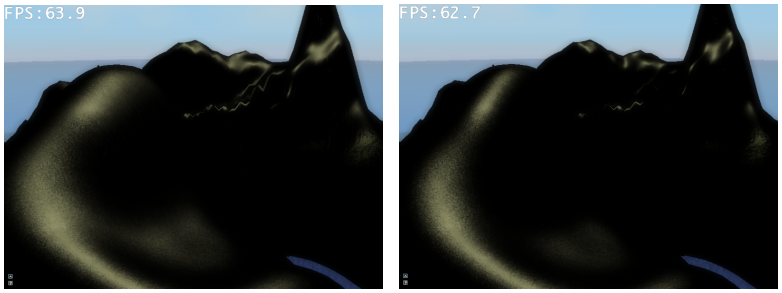
(c) Specular lighting. (d) Combined lighting.

Figure 2.3: Images of the different light components making up the terrain lighting.

A faster approximation was proposed by Blinn[3]. Instead of basing the specular highlight on the angle between the view direction and reflection vector, he proposed to base it on the angle between the normal and the half-vector, a vector halfway between the view- and light direction given by $\mathbf{h} = (\mathbf{l}+\mathbf{v})/\|\mathbf{l}+\mathbf{v}\|$. An approximate relationship between the two is given in [AMHH02, page 77] as

$$(\mathbf{r} \cdot \mathbf{v})^{m_{shi}} \approx (\mathbf{n} \cdot \mathbf{h})^{4m_{shi}}$$

Both specular light models have been implemented in the terrain fragment shader and the results can be seen in figure 2.4



(a) Phong specular lighting (b) Phong specular lighting

Figure 2.4: A comparison of Phong specular lighting and Blinn specular lighting.

Since Blinn yields little to no framerate increase, we use Phong specular lighting as the default, as it gives a more pleasing result.

[3][AMHH02, page 77]

The GLSL code for lighting with Phong specular light can be seen in listing 2.1.

```
vec3 phongLighting(in vec3 text, in vec3 normal, in vec2 specProp){
  // Calculate diffuse
  float ndotl = dot(lightDir, normal);
  float diffuse = clamp(ndotl, 0.0, 1.0);

  // Calculate specular
  vec3 vRef = normalize(reflect(-lightDir, normal));
  float stemp = clamp(dot(normalize(eyeDir), vRef), 0.0, 1.0);
  float specular = specProp.x * pow(stemp, specProp.y);

  return text * (gl_LightSource[0].ambient.rgb +
                 gl_LightSource[0].diffuse.rgb * diffuse +
                 gl_LightSource[0].specular.rgb * specular);
}
```

Listing 2.1: A GLSL function calculating lighting given a color, normal and specular intensity. The view/eye- and light direction are uniforms.

2.2 Bump mapping

With the terrain shaded by the sun, it is time to improve on the detail of the lighting by adding 'bumps' to the surface, or more precisely manipulating the normal to create a bumpy feel. This is done by storing the normals of a surface in a normalmap texture, then instead of using the interpolated geometric normal for light calculations, we transform the texture normal into tangent space and use that for lighting.

In order to do this the interpolated normal, tangent and bitangent are used as the axes of a tangent space coordinate system. Since we are using the y-axis as up, the normal will be our y-axis in the tangent space coordinate system. The tangent and bitangent are normalized vectors perpendicular to the normal and lie along the normalmaps texture coordinate axes. Calculating them normally is done as a pre-processing step and they are stored as vertex attributes, or in our case a tangent- and bitangentmap. However due to the heightmap being a 2D array of vertices with texture coordinates proportional to the x- and z-coordinates, we can calculate the tangents directly in the shader. This saves us two texture lookup and the need to bind additional textures.

Since our texture coordinates are axis-aligned, the tangent will lie in the xy-plane and the bitangent is placed in the yz-plane. Below we will show how to calculate the tangent, but the same principle can be applied to calculate the bitangent.

The interpolated normal is our approximation of a vector perpendicular to the terrain surface. It can therefore also be used to approximate vectors along that surface, specifically the tangent along the xy-plane. This is done by taking the cross product between the normal, n, and the unit vector along the z-axis, $z_i = (0, 0, 1)$, which will ensure that the tangent is both perpendicular to the normal, ie. lying in the plane, and perpendicular to the z-axis, which places it in the xy-plane. The calculations for the direction of the tangent are

$$\begin{aligned} n \times z_i &= (n_x, n_y, n_z) \times (0, 0, 1) \\ &= (n_y 1 - n_z 0, n_z 0 - n_x 1, n_1 0 - n_2 0) \\ &= (n_y, -n_x, 0) \end{aligned}$$

All that is left is to normalize the vector to get the tangent.

The tangent, bitangent and normal now make up the basis vectors of the tangent space coordinate-system and can be used to either transform the direction of the light from global space to tangent space or transform the bumped normal from tangent space to global space. For this project the latter was chosen to easier incorporate the shaders into a deferred lighting pipeline in the future.

The GLSL code can be seen in listing 2.2

```
// Extract normal and calculate tangent and binormal
vec3 normal = texture2D(normalMap, texCoord).xyz;
vec3 tangent = normalize(vec3(normal.y, -normal.x, 0.0));
vec3 bitangent = normalize(vec3(0.0, -normal.z, normal.y));
mat3 tangentSpace = mat3(tangent, normal, bitangent);

// Extract normals and transform them into tangent space
vec3 bumpNormal = texture2DArray(normalTex, vec3(srcUV, layer)).xzy;
bumpNormal = bumpNormal * 2.0 - 1.0; // move from [0; 1] to [-1; 1]
bumpNormal = normalize(tangentSpace * bumpNormal);
```

Listing 2.2: Calculating tangent space for a heightmap and rotating the normal in GLSL.

2.3 Texturing the terrain

Several layers of color textures and normalmaps have been added to the heightmap for a more diverse looking terrain. There is a sand layer, a grass layer and a snow layer. Rocks have also been added on steep slopes, but the focus is on layered texturing.

For our layered textures we chose to use the GL_EXT_texture_array extension, which allows us to create an array of textures that are mipmapped individually. Unfortunately the texture array does not blend across the different layers, so this has to be handled by the fragment shader.

An alternative would be placing all the layers in a three dimensional texture, which could then blend between the different layers upon texture lookup in the shader. The downside to this approach is that using mipmapping on a three dimensional texture, will blend the different layers together. In our case creating this creates a brownish looking texture in the second mipmap level and downwards. The only solution to this would be to disable mipmapping, which results in a performance penalty and creates flickering textures.

In the end we opted for texture arrays over 3D textures and implemented the blending ourselves. For each layer several parameters can be chosen

- `startHeight` - At what height the layer should start to blend in.
- `blending` - How many units of height it takes the layer to become completely opaque.
- `spec[i]` - The specular factor of the i^{th} layer, where the x-component specifies a specular intensity scalar and the y-component is the shininess

How much of the i^{th} layer should be visible, $factor_i$, is then trivially calculated by

$$factor_i = (height - startHeight_i)/blending_i$$

Since it does not make sense for a layer to be more than completely transparent or opaque , ie. a factor of 0 or 1, we clamp the factor between these values. The actual index of the layer to be used in the texture array is then quite easy to calculate. We simply sum over all the factors.

And example could be when layer 2 is completely opaque and layer 3 is 40% visible. At that height layer 1 will also be completely opaque. The layer height is then $1 + 1 + 0.4 = 2.4$, which means that the base layer is 2 and that has be blended with 40% of the 3'rd layer.

When specifying the height or the terrain some randomness is added to it from the normalmap. This is done to avoid smooth transitions between the layers and give the terrain a more natural feel.

Chapter 3

Waving Grass

A completely static terrain would appear quite lifeless. Some foliage gently swaying in the wind would make the landscape seem more idyllic and therefore we chose to add waving grass straws. Some considered rendering techniques for grass were:

- Rendering grass straw textures on quads. This allows us to render several grass straws while only having to process 4 vertices. Letting it wave in the wind is also easy as that only means moving the 2 top vertices. A problem however is that the illusion breaks down when viewed from above or on steep cliffs.
- Modeling each grass straw. This technique allows for very flexible grass with correct lighting. It is however very expensive to process all the vertices, so it is usually used in conjunction with the above method for far away grass.
- Rendering a volume texture containing the grass straws.

We opted for grass drawn onto quads, since it is easy to draw a quad in the scene and grass textures are easy to come by. However the entire landscape can not simply be covered with millions of quads, as that would be extremely expensive to render, so there are some design decisions to be made. Another problem with this technique is that part of the grass texture is transparent, and transparency is not easy to render in the OpenGL 2.1 pipeline. A solution would be *depth peeling* or *z-sorting* the straws, but for our purpose we propose a much simpler method in the Handling Transparency subsection.

The basis of the grass implementation is [Fer04, chapter 7]. Here they propose to draw the quads in star patterned grass objects, to achieve a grass effect independent of the cameras line of sight. Our implementation does the same, but allows the user to specify how many quads should be used pr. star object. Setting this to 1 effectively disables the star pattern and simply renders random quads of grass straws instead.

3.1 Placing the grass

As stated earlier we cannot simply draw the grass everywhere without incurring a huge performance penalty, so instead we focus on drawing the grass around the camera. We choose to draw it in a square around the camera, but other shapes should be possible.

On figure 3.1 the basics of the algorithm can be seen. We need to translate the grass bounding box centered around origo to the camera's position. We can not simply use the vector from origo to the camera, since that would make the grass straws float along with the camera. Instead we need the global position of the quads to be moved up to the camera with a step size equal to the length of the bounding box' side. From the camera this would create the effect that quads exiting the camera bounding box on one side would reappear on the other side. In practice this isn't visible from the camera and the illusion of individual grass straws is preserved. The following equation expresses the this desire.

$$-halfSize \leq position + size * n - eye \leq halfSize, n \in N$$

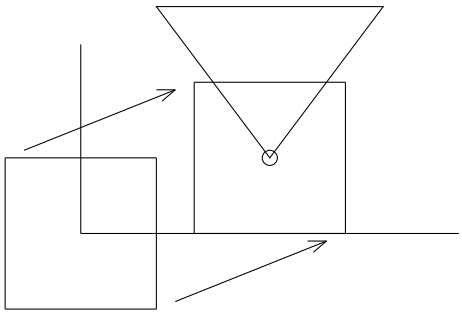


Figure 3.1: The translation of the grass bounding box from origo out to the camera.

Focusing on the left hand side of the equation and remembering that n should be an integer, n can be calculated as

$$
\begin{aligned}
&-halfSize \leq position + size * n - eye \\
&\Updownarrow \\
&(eye - halfSize - position)/size \leq n \\
&\Updownarrow \\
&n = \lceil (eye - halfSize - position)/size \rceil \\
&\Updownarrow \\
&n = \lceil (eye - position)/size - 0.5 \rceil
\end{aligned}
$$

Now the new position of the grass straw is simply calculated by

$$newPos = position + size * n$$

An added benefit to this approach is that even though the grass is dynamically placed, it is deterministic and is always placed the same. This means that if a user is exploring the terrain and returns to a previously visited area, the grass will be placed the same way.

The grass objects have now successfully been placed inside the bounding box around the camera, as can be seen on figure 3.1. However most of the straws still are not inside the frustum. A small trick can be applied to move more of the grass up in front of the camera. Instead of placing the grass around the camera, it can be placed at a position in front of it, relative to the cameras current direction. Since at least half the grass bounding box will be behind the camera, we choose to move the bounding box by the $halfSize$ multiplied by the normalized camera direction. This places the center of the grass in front of the camera, but the border will always be behind the camera. The result is many more grass straws on the screen at the same vertex processing cost.

Since grass placed anywhere will now be translated to the camera, we can create it anywhere we like. More importantly all the quads of grass straws can now be drawn with one single draw call, and the vertex shader will translate it to the camera. This removes a major overhead we would have had if all quads were drawn individually.

Next we must translate the grass objects along the y-axis to place them on the terrain, instead of below or above. Since we're continuesly translating the grass quads, we won't know their correct height at creation time and must therefore look it up in the vertex shader. From grass' xz-position a texture coordinate can be calculated and used to lookup into the heightmap texture, where we can find the correct height for the grass. To ensure that the terrain and grass are equally lit, we use the same procedure to lookup into the heightmap's normalmap.

The last part of the grass placement is to keep it off the sand, snow and steep cliffs. To this end we pass along the center of the grass straw to the vertex shader. The center is then subjected to the same translations as the vertex position. Whether or not the grass straws should be visible can then be estimated from the height of the center. If the center is too low the grass is on sand and should not be visible, likewise if the center is too high it is on snow. A similar scheme is used for deciding if the grass is placed on cliff, where the angle of the normal compared to the up vector is the deciding factor.

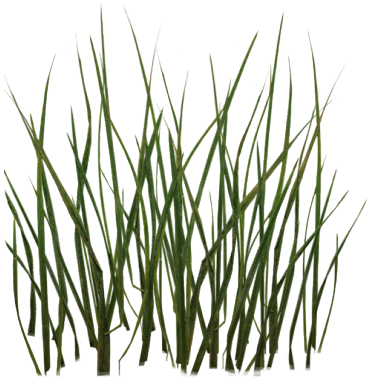


Figure 3.2: The grass straws.

The reason for using the center of the grass objects is that while one side of the quad may be placed perfectly legal, the other side could be placed on sand or snow. In order to collapse the grass objects we needed a constant variable across the objects and that is the center.

3.2 Waving in the wind

Our animation algorithm is the same as the one proposed in [Fer04, section 7.4.4], where the grass objects center is used to create *local chaos*. This means that textures are minimally distorted, since the vertices always have the same distance to their neighbours, and the local chaos creates a more natural wave effect. Of the two cons to the approach described in [Fer04, section 7.4.4], only one of them applies to our implementation. The first con, which is that extra uniform data is required to hold the center of the grass object, does not apply in this case, since the center was needed anyway. The second con does apply to our shader, namely that the complexity of the animation algorithm is limited in order to minimize the cost of the shader. The wave therefore is just a simple cosine function applied to the two top vertices, distinguished by their texture coordinate.

3.3 Handling Transparency

With the grass placed on the terrain, texture transparency needs to be handled. Our grass straw texture can be seen in figure 3.2. Since all grass objects are drawn with one draw call, z-sorting every frame is not an option. Instead the transparency issue is completely ignored, by only rendering completely opaque fragments or discarding transparent fragments. This is done in the fragment shader, where colors with alpha values below a certain value are simply discarded. The reason for not just discarding everything with alpha values strictly below one is that mipmapping has to be taken into account. If two opaque and two transparent pixels are combined in a lower mipmap layer, this would yield an alpha of 0.5, but the fragment should perhaps still be rendered. In practice we found that discarding alphas below 0.6 works well with our grass straw texture.

Chapter 4

Sky

The sky is an essential part of an outdoor environment, whether it is night, day, cloudy or clear sky plays an important role in setting the mood of the scene. The sky is an ever changing part of an outdoor scene and depends on many parameters including both the time of the day and on the weather conditions. This makes it a hard effect to simulate and visualize, but because it is such a central part of an outdoor setting, it must be included in the rendering. One possibility is to simulate all important real world components influencing the appearance of the sky. This could involve complex weather forecast models or other large simulations, but because there are so many complex phenomena influencing the skies appearance, the calculations of such models cannot be computed in real-time. Instead we have chosen some of the skies most obvious visual effects and tried to create believable visualizations of those.

When rendering a scene it is important that the overall LOD matches to convince the user of our illusion. This means that if we have a scene with high detailed objects, we also need to render the sky in high detail. If we had a simple scene, then it could be enough to simulate the sky by a simple light blue background color. This approach works fine when used together with for example cartoon shading. But as objects in the scene becomes more detailed and realistic looking this simple approach makes the sky stand out. Simulating a more realistic looking sky can and have been done many different ways. Some authors use what is known as a *sky box* [AMHH02, page 338-339], which is a technique that maps images of the sky onto an axis aligned box. This box is so large that it contains the entire scene and by moving the center of the box according to the camera position, it creates the illusion of far stretching planes by using simple images. One problem with using a box when doing this, is that the edges between the images on different faces of the box can be apparent as artifacts when looking directly at them. To get around this problem, developers instead use a sphere or dome as the basic geometry, and hence what is known as a *sky dome*, when rendering the sky. The main difference between these two approaches is texture mapping the images. When mapping images onto a box a straight forward approach can be used, but when using a dome this step becomes much more tricky. Although the sky dome technique gives fine results, the sky remains frozen in time, which makes it unrealistic when looking at it over longer periods in time. So we have also take up the challenge of simulating or animating the images which is to be mapped onto the sky dome.

4.1 Sun

Before we describe the sky dome we will introduce the source that lights and thereby influences the entire scene. The only light source in our scene is the sun, which is a directional light source that we move around according to the time of day. The direction of the sunlight is used in a lot of our shaders for light calculations and used to create the glowing sun in the sky. Apart from changing the direction of the light, the `SunNode` also changes the diffuse light depending on the time of day. This helps us create a more realistic looking sunset, where the sun first turns orange and then red as it sets. The scene is also lit with the diffuse sunlight, which means that at sunrise and sunset it has a warm orange glow, as can be seen on figure 4.1.

Figure 4.1: The sunlight shines orange at sundown.

4.2 Sky domes

We have chosen to use two sky spheres, one inner sphere for rendering clouds as a grey scale image with alpha transparency, and an outer sphere to render a background color which can change based on the time of day. We also include texture coordinate animation of the cloud texture based on a simple wind model, so the clouds change over time.

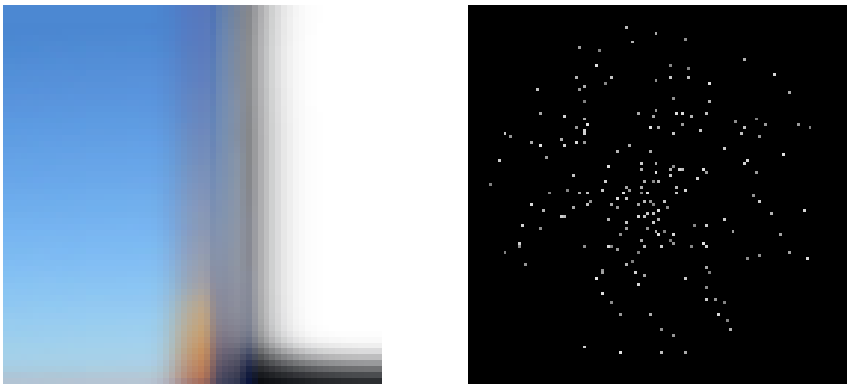
4.2.1 Geometry

We have tried two different ways of generating the sphere geometry. The first method used was the traditional longitudinal/latitude sphere commonly used for geographical maps. This however gave very obvious image stretching artifact at the poles when mapping a rectangular texture directly onto the sphere. Instead we now use geodesic spheres which consist of equilateral triangles. This sphere is beautifully constructed by recursively subdividing the triangles of an icosahedron until the required LOD is attained.

4.2.2 Atmospheric dome

What phenomena drives the coloring of the sky, making it so beautiful with it shades of blue and red? This is the questing asked by researchers studying *atmospheric scattering*. Atmospheric scattering has been studied in great detail and many complex theories and models has been developed within the field. We however have chosen a simple technique to model this complex process. This technique is based on a *color gradient* as described in [Aba06].

We use the color gradient in figure 4.2a to color the fragments of the atmospheric dome. We use the y-component of the texture coordinate for texture mapping vertically on the color gradient and the time of day in the horizontal direction. This divides time into three periods: day, sunset/sunrise, and night by first going from left to right, and then the other way. The white color seen on the right in figure 4.2a is alpha transparency, which is used at night to enable rendering of stars. We render stars simple by pre-generating a black square image with randomly placed pixels, painted shades of white, in a circle round the center of the image as illustrated in figure 4.2b. This images is then alpha blended with the color gradient based on the three dimensional texture coordinates of the atmospheric dome. The x and z coordinates are used (the y axis is up).



(a) Color gradient. (b) Star texture.

Figure 4.2: Textures used to color the atmospheric dome.

4.2.3 Cloud Dome

The cloud dome is simply used to map a cloud texture.

Cloud texture

We have chosen to use a procedural generated image to texture the cloud dome. By taking this approach we can generate very different looking clouds by only altering a few parameters that is supplied to the generation algorithm. We have furthermore chosen a three dimensional image because this eases texture mapping when the image is to be mapped onto the sphere. To generate the cloud image we have been inspired by an Internet homepage[1] The page describes how to use procedural generation and image layering/composition in combination to produce textures that look like real clouds. Because procedural data generation is a large subject we have dedicated a separate chapter to it, chapter 5. Here it is enough to say that we generate a three dimensional texture which can be tiled in all three dimensions.

Texture mapping - animation

The basic texture mapping scheme, is to put the three dimensional texture into a unit cube, and use texture coordinates based on the unit sphere to make a direct mapping which avoids texture stretching. If we only did this, a large portion of the generated texture would not be used. So besides the basic texture mapping scheme, we have introduced texture coordinate animation. This has been done by modeling the wind.

Wind simulation

We have implemented a simple wind simulation algorithm which is inspired by *Brownian motions*. The algorithm works by having a two wind directions referred to as the current and future direction. The future direction is generated randomly based on the current direction and is normal distributed around this. The wind is then changing progressively by interpolating linearly between these based on time. When the future wind direction is reached we overwrite the current by the future direction and generate a new future wind direction.

[1] `http://freespace.virgin.net/hugo.elias/models/m_clouds.htm`
Be warned though, this homepage does not use Perlin noise as they say but value lattice noise.

Chapter 5

Procedural generation of content

What is known as procedural or synthetic generation of content is used extensively in computer graphics to enable generation of natural looking details. By generating details, in contrast to doing them by hand, boring work typically done by computer artists can be cut down. The term procedural generation of content is very fuzzy and covers quite a large area of different techniques, but boils down to some kind of algorithm that produces data [Ebe03, page 12]. The algorithm typically has a number of parameters that can be specified to control the generation process in a way that alters the result in an intuitive way based on the parameters.

5.1 Generating irregular data

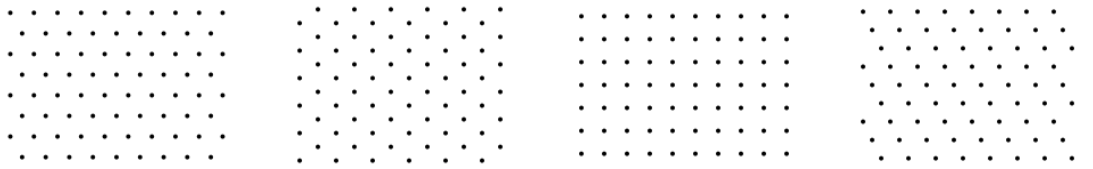
We have chosen to concentrate our effort on irregular data generation minded on textures, therefore only this type of data generation will be covered in the following. The basic building blocks when generating irregular data are noise functions. A noise function is a function that is apparently stochastic and will break up monotony of the patterns that would otherwise be too regular [Ebe03, page 67-78]. Typically a noise function is a function of position so the arguments can be filled directly when rendering, e.g. $f(x, y, z)$. An obvious stochastic primitive is uniformly distributed random numbers with no correlation, which is also known as white noise. In computer graphics however true random numbers are rarely used, because they are not controllable. Instead pseudo random number, PRN, generators are used to produce a fair approximation to white noise. These however are controllable and furthermore produce the same number sequences on two successive runs provided the algorithm is started with the same preconditions. Another property of white noise that we do not want is that it is uncorrelated in all function values. This causes problems in computer graphics because of rounding errors in the floating point representation used when rendering. Instead we use a low-pass filtered version of an approximation to white noise.

There are many different ways of using PRN to produce noise functions, each method generates functions with different properties. Generally the functions are divided into categories based on how they generate the noise. To make an important and often misunderstood point regarding what precisely defines the famous Perlin noise function, we will introduce two basic categories of noise functions: Value and gradient noise, which both also are categorized as lattice noise.

5.1.1 Lattice noise

In geometry a lattice is a discrete subgroup (a set of points) in the same dimension as the basis vectors spanning the vector space. A lattice can be viewed as a regular tiling of space by primitive cells forming a grid structure. The most used lattice in computer graphics is the regular Cartesian grid, which can be viewed as a tiling of squares in two dimensions, see figure 5.1c. For clarity figure 5.1 also provides other examples of lattices.

Lattice noise is generated by having a lattice, and then for each lattice point generating one or more random values for that point depending on the type of noise we want. The low-pass filtering is then done by interpolating the values at the grid points to fill the entire space between the lattice points. Different



(a) Triangle lattice. (b) Rhombic lattice. (c) Square lattice. (d) Parallelogram lattice.

Figure 5.1: Four simple lattice types in two dimensions.

interpolation scheme can be used to calculate these values, which yield different properties for the noise function.

Value noise

Value noise is the simplest type of lattice noise, here the values stored at each grid point are the values that are interpolated and returned when the function is evaluated.

Gradient noise

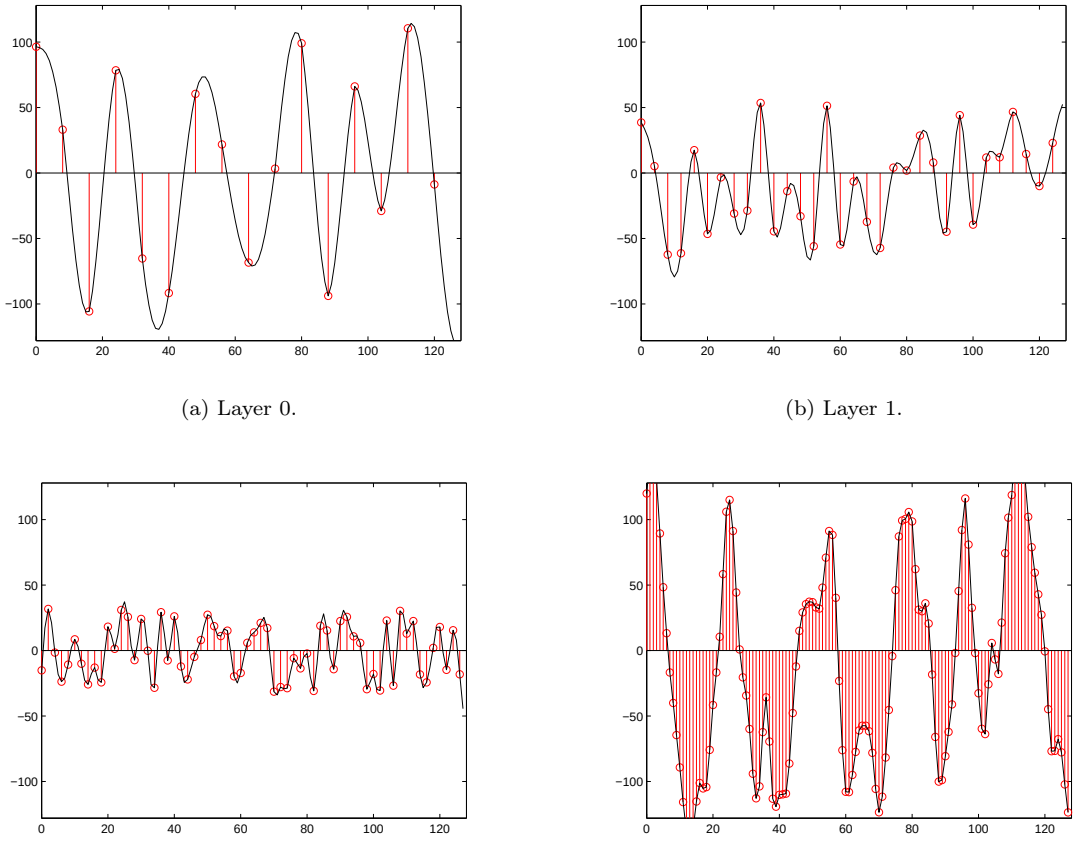
Gradient noise however generates a gradient at each point in the lattice, which is a vector with the same dimensions as the number of dimensions wherein the function is defined. The gradient in each grid point is then used to calculate a continuous function between the grid points, which again can be used to evaluate a noise value. The gradient noise category includes the now famous noise function known as Perlin noise, first described by Ken Perlin in the paper: [Per85]. Here we once again want to warn the reader that many authors of web pages, e.g. `http://freespace.virgin.net/hugo.elias/models/m_perlin.htm`, seems to confuse Perlin noise with value noise, with the result that an Internet search for Perlin noise becomes a non-trivial job. An excellent description of Perlin noise can be found in [Ebe03, chapter 12] which is also written by Ken Perlin.

5.2 Composition of layers

Noise alone does not make realistic looking textures, only by carefully choosing the noise parameters and cleverly combining many layers of different looking noise results in natural beauty. In the following, we will focus on generating a volumetric texture of clouds, but the description illustrates the overall process of how to generate and combine noise functions. To describe how we combine noise functions, we will use terminology borrowed from the field of digital signal processing, DSP. In DSP the term *sample frequency* describes the space between to consecutive samples, and the term *amplitude* the maximum value of the function.

To generate the basic shape of some clouds we use a low frequency noise with a large amplitude as illustrated for a one dimensional case in figure in 5.2a. We add layers of details onto this low frequency noise by combining it with noise of higher frequencies. Because we want the basic shapes to be the most dominant the amplitude of noise with higher frequencies are lowered, this is shown in figure 5.2b and 5.2c. We change these frequency and amplitude in octaves because octaves frequently occur in nature and give good visual results. For each level of detail that is added, the frequency is doubled while the amplitude is halved. To illustrate this, we have used value noise to generate three layers and combined them into the result show in figure 5.2d.

Moving into two and three dimensions are straight forward. Figure 5.3 shows four layers of two dimensional white noise rendered as alpha transparency on a fully white image, the blue background has been inserted so the transparency can be seem. These layers are the basic primitives for the upcoming combination into an image that looks like clouds. When sampling into an array, as has been done in figure 5.3 the increase in sampling frequency is the same as using a larger array. If this array represent an image or texture, as in the figure, then the different layers do not have the same height and width. To solve this, when algorithmically combining two layers, the layer with the lowest resolution is re-sampled to the same resolution as the other layer.



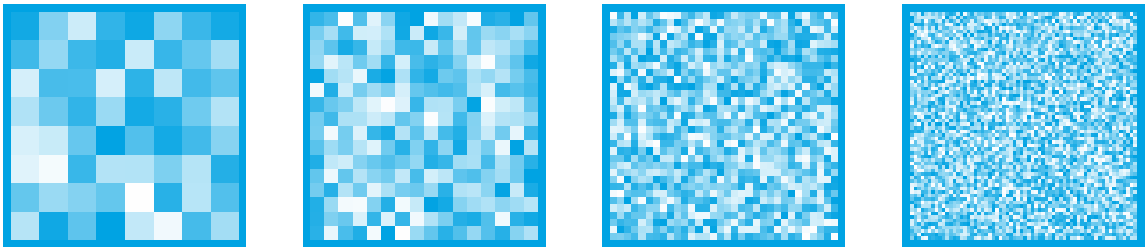
(a) Layer 0.

(b) Layer 1.

(c) Layer 2.

(d) Result.

Figure 5.2: One dimensional example of the combination process.



(a) Layer 0.

(b) Layer 1.

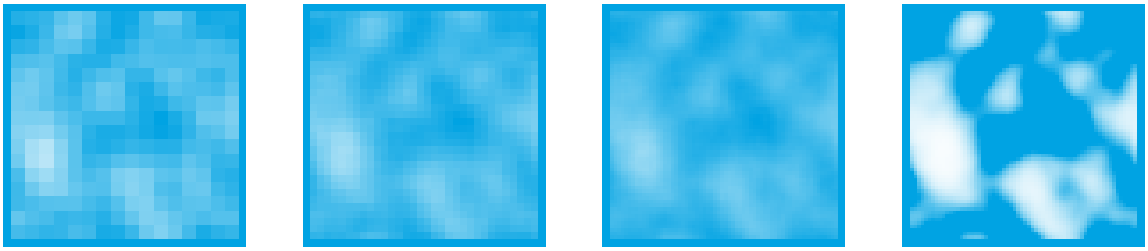
(c) Layer 2.

(d) Layer 3.

Figure 5.3: Four noise layers.

The layers are then combined two at the time. The process starts by combining the two layers with lowest resolution. Then the result of the previous combination is combined with the image with third lowest resolution and so on. Because we use linear interpolation between our lattice point, the result of combining the layers, becomes very blocky looking, like the image in figure 5.4a. To get around this we blur the result after each new layer is added, which can be seem in figure 5.4.

The final touch, after the image is generated, is to filter the image with an exponential function which creates the look seen in figure 5.4d.



(a) Combo of layer 0 to 1. (b) Combo of layer 0 to 2. (c) Combo of layer 0 to 3. (d) Result.

Figure 5.4: Snapshots of the combination process.

5.3 Implementation

When implementing noise functions, one must choose where the computations are placed. One option is to do all computations as pre-computations and then use the resulting noise texture for look up at run-time. Another option is to do all the computations at run-time. The main advantage of doing the calculation as pre-computation is an increase in run-time performance, where as doing the calculations at run-time enable dynamic adaption in LOD in relation to the camera position. For the run-time computations to be fast enough for real-time rendering they are usually done on the GPU, but although noise functions exists in the GLSL API [Ros06, page 145-146], they are not recommended because their behavior is not fully described in the specification and are implemented differently by different vendors and on different hardware. Developers that still want to do the calculations at run-time, instead use a hybrid solution, generating a small noise texture which can be used to implement more sophisticated noise function on the GPU. We however have chosen to do all the calculations as pre-computations implemented for CPU execution. Our implementation is based the `RandomGenerator` class of OpenEngine which again is based on the PRN from [FPTV92, page 278-283]. The core of the entire algorithm for generating noise, composing layers, and blurring them is implemented as the recursive shown in listing 5.3 function (the full implementation can be found in the `ValueNoise` class of the `TexUtils` OpenEngine extension):

```
static FloatTexture2DPtr Generate(unsigned int xResolution, unsigned int yResolution,
                                  unsigned int bandwidth, float mResolution,
                                  float mBandwidth, unsigned int blur,
                                  unsigned int layers, RandomGenerator& r) {
        FloatTexture2DPtr noise =
              CreateNoise(xResolution, yResolution, bandwidth, r.UniformInt(0,256));
        if (layers != 0) {
            FloatTexture2DPtr small =
                Generate(xResolution * mResolution, yResolution * mResolution,
                         bandwidth * mBandwidth, mResolution, mBandwidth,
                         blur, layers-1, r);
            noise = Combine(noise, small);
            Blur(noise, blur);
        }
        return noise;
}
```

Listing 5.3: The `Generate` function from the `ValueNoise` class of the `TexUtils` extension.

Chapter 6

Post Processing

In the previous sections geometry has been used to render the landscape and sky to the screen. Now we turn to *screen space effects*, which do not rely on geometry, but will instead use the final color image and the depth buffer to create effects. *Depth of field*, *motion blur*, a screenwide *glow* and many more effects can all easily be produced through post processing.

We will try to create a general framework inside OpenEngine that can handle most effects where the programmer needs to do as little setup as possible. It should also be possible to stack post processing effects on top of each other, for example to combine a *cartoon shader* with *edge detection* or optimizing blurring by using the two pass *separable convolution technique*[1].

6.1 Structure

In this section we will first outline the new scene node called `PostProcessNode` and how it will help the programmer setup post processing. Then we will discuss the specifics of rendering the scene, applying the post process effect and how we eventually chose to structure the rendering.

Post Process Node

The `PostProcessNode` will apply a fragment program to manipulate the image rendered while visiting subnodes in the scene graph, the subscene. Therefore it of course contains a pointer to OpenEngine's shader abstraction, `IShaderResource`, that points to the effect. It contains a pointer to an OpenEngine `FrameBuffer`, which scene subnodes are rendered to. The `PostProcessNode` also holds another `FrameBuffer` where the final image can be *blitted* into after the effect has been applied. This is useful for some implementations of motion blur, that relies on the previous frame being available.

In order to allow users of the `PostProcessNode` to focus on writing their effects, the node will handle most of the setup as long as a few naming conventions are followed. If a uniform named `depth` is detected then the subscenes depth texture will be bound to this uniform. Similarly if `imageN`, where `N` is any integer, is seen, then the `FrameBuffer`'s N^{th} color attachment will be bound to that uniform. The same goes for the final image and the uniform name `finalImageN`. The node can also decide to improve performance by not blitting the final image, if this is not needed. Of course the previous image doesn't make sense in the very first rendering, so when initialized the node binds the not-processed scene image as the final image, and then after the first frame it switches to the final image. And finally, if needed the node will pass the time to the shader. All of this has allowed us to cut back on doing setup and spend our time writing cool and fun effects.

Rendering

When the `RenderingView` visits the `PostProcessNode`, the very first thing that needs to be done is storing the name of the current render buffer and the viewports dimensions. These are needed

[1] [Fer04, section 21.2.3]

in order to restore them after the subscene has been rendered. With the previous state saved, the `PostProcessNode`'s `FrameBuffer` is bound and the subscene rendered to it.

```
GLint prevFbo;
glGetIntegerv(GL_FRAMEBUFFER_BINDING_EXT, &prevFbo);
Vector<4, GLint> prevDims;
glGetIntegerv(GL_VIEWPORT, prevDims.ToArray());

Vector<2, int> dims = node->GetDimension();
glViewport(0, 0, dims[0], dims[1]);
glBindFramebufferEXT(GL_DRAW_FRAMEBUFFER_EXT, node->GetSceneFrameBuffer()->GetID());
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

node->VisitSubNodes(*this);
```

When the scene has been rendered to the node's own `FrameBuffer`, the previous framebuffer and viewport are restored. The post process is then applied by binding the post process effect and drawing a rectangle across the screen. To increase performance the depth test is set to always allow all fragments.

```
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, prevFbo);
glViewport(prevDims[0], prevDims[1], prevDims[2], prevDims[3]);

// Gently disable the depth func (while preserving depth writes)
glDepthFunc(GL_ALWAYS);
// Then render the effect
node->GetEffect()->ApplyShader();
glRecti(-1,-1,1,1);
node->GetEffect()->ReleaseShader();
glDepthFunc(GL_LESS);
```

If the post process effect relies on the final image from the previous rendering, then now is the time to save the image. However recall from earlier that the previous image was originally set as the current image. Therefore if this is the first render pass, the final image texture has to be bound to the effect, so it is ready for use in the next pass. The final image can then be blitted from the current renderbuffer into the renderbuffer that will hold the final image.

```
FrameBuffer* finalFb = node->GetFinalFrameBuffer();
if (finalFb != NULL){
  if (finalFb->GetID() == 0){
    // Initialize the final frame buffer and assign the
    // textures to the effect shader.
      ...
  }
  // Blit the images from the previous framebuffer to the final framebuffer
  glBindFramebufferEXT(GL_DRAW_FRAMEBUFFER_EXT, finalFb->GetID());
  glBlitFramebufferEXT(prevDims[0], prevDims[1], prevDims[2], prevDims[3],
                       0, 0, dims[0], dims[1],
                       GL_COLOR_BUFFER_BIT, GL_LINEAR);

  // Reset to previous fbo
  glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, prevFbo);
}
```

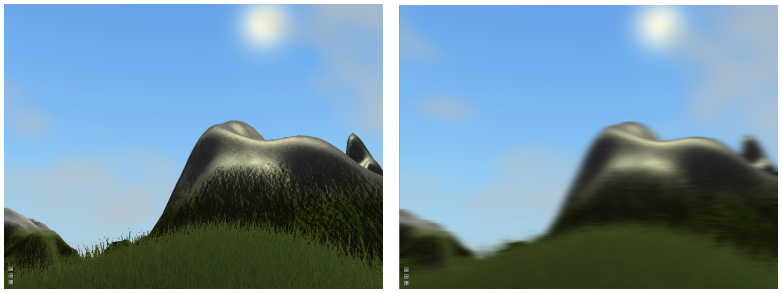
6.2 Post Process Effect

In this section we will take a short look at some of the different post process effects that we have created.

6.2.1 Motion Blur

While motion blurring may not be critical to outdoor environments, many projects still benefit from it. Therefore one of the criteria to the post process node was that it would allow an easy motion blur implementation for existing and future OpenEngine projects.

Motion blurring can be done in several ways. The simplest way conceptually is to store the last n images and then blend them together. This creates a nice blurring effect across the last few frames. The problem is that we then need to store the last few frames in memory, giving us a memory overhead of



(a) Non blurred image. (b) Blurred image.

Figure 6.1: Images used to produce the glow effect.

possibly 8-16 textures. Another problem is that the blurring then becomes framerate dependent, which means that low framerates will make the blurring excessive, while high framerates will make it non existent.

Another approach was proposed in [Ngu07, chapter 27], where the previous *ViewProjectionMatrix* is used to calculate the previous position of a fragment and then accumulate the colors between the current and previous position. This approach however only creates blurring if the camera is moved. A car that drives past the camera will be crystal clear. Another problem is that the shader becomes quite complex and time consuming.

The algorithm that we settled on is *ghosting motion blur* which combines a low memory footprint with a fast shader. We use an accumulation texture for storing all the previous images blended together, and then blend that texture together with the current image. This way the contribution from old images will tend toward 0 and the motion blurring will produce nice looking blurred 'tails' on moving objects. Of course since we can only blur between rendered frames, a fast turning camera will produce artifacts. The result can be seen in figure 6.3

6.2.2 Glow

When looking at natural outdoor environments in bright sunlight, colors seem to blur or smear together. We would like to reproduce this effect using a fullscreen glow post processing effect, as is also proposed in [Fer04, chapter 1]

Initially the effect was created in just a single shader. This shader did a blur on the surrounding fragments and then blended the result with the original fragment. This produced the results we where after, but at a huge performance overhead. The problem was texture lookups. To blur the surrounding fragments of an $N \times N$ rectangle, we required N^2 texture lookups. The performance problem was solved by applying the separable convolution technique. In the first post process pass a horizontal blurring is performed, then in the second pass a vertical blurring is performed on the resulting image of the previous pass and these images are then blended together with the original non-blurred scene. For an $N \times N$ rectangle we have reduced N^2 texture lookups to $2N$, while achieving the same effect.

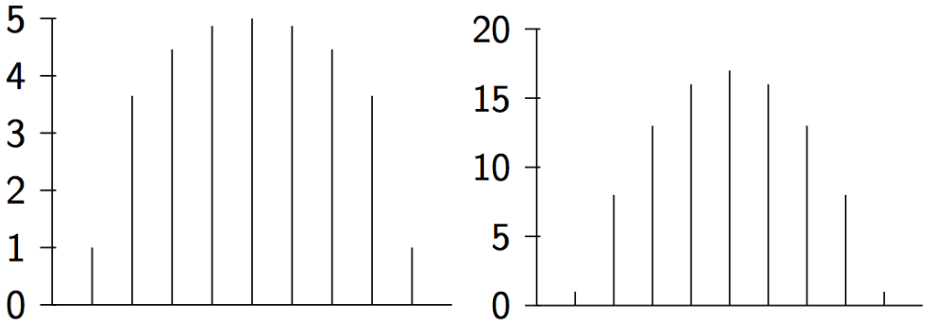
As can be seen on the image on the front page, a subtle smearing gives a smoother and more natural looking image. The non blurred and blurred image can be seen in figure 6.1 and the shader that does the vertical blurring and blend it with the original image can be seen in the file:
`project/Terrain/data/shaders/Glow.frag`.

6.2.3 Depth of field

When taking pictures or filming through a lens an effect called Depth of Field occurs. Essentially everything not in focus by the camera becomes blurry, and the greater the distance between the object in focus and the out-of-focus object the more blurry the object will appear.

With the color texture and depth texture of the rendered scene it is possible to emulate this effect. In our example the focus of the camera will always be the center of the screen.

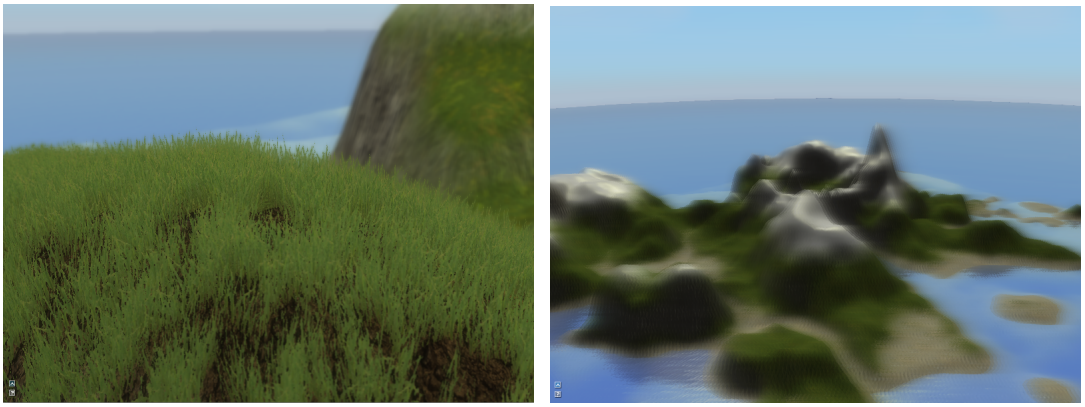
To create the effect we first have to determine how much out of focus a fragment is, which determines the fragments blurring offset. This is done by computing the difference between the depth at the center



(a) Half circle kernel.

(b) Half circle kernel.

Figure 6.2: Blurring kernels



(a) Depth of field is visible on the background

(b) The landscape is motion blurred as the camera pans to the side.

Figure 6.3: Post processing effects.

of the screen and the depth at the fragment.

As before, in the glow effect, we again create the blur by sampling a box around the fragment. This time though the distance between sample points is not constant, but proportional to the blurring offset. The fragments in focus or near focus have a small blurring offset and thus will not be much blurred, while fragments with a high blurring offset will be sampled inside a larger square and thus be blurrier.

We decided to blur with a half circle kernel, as can be seen on figure 6.2. To increase effectiveness we tried squaring the sample weights and came up with the kernel in figure 6.2, which while faster still provided a pleasing result.

The result can be seen on figure 6.3 where the hill in the background and the horizon are slightly blurry. The shader that creates the horizontal depth of field blurring can be seen in the file: `project/Terrain/data/shaders/HorizontalDepthOfField.frag`.

Chapter 7

Conclusion

We have created an exciting dynamic outdoor environment that changes look and feel according the time of day. To do this we have used OpenGL's programmable pipeline and some of OpenGL new extensions.

The time of day is easy to see when looking at the sky. At day time the sky is bright blue with animated clouds, at night star fields fill the sky, and at sunrise and sunset yellow and red scattered sunlight can be seen along the horizon. Details have been added to the terrain by using advanced lighting techniques, such as bump mapping, and waving grass makes it come alive. A final level of detail is provided by the our new post processing extension which includes effects such as motion blur, glow, and depth of field. Other effects such as edge detection and an underwater post processing effect have already been created and applied to other OpenEngine projects, so we also succeeded in creating an easy-to-use interface, which will hopefully be of use to many.

All in all, we are quite satisfied with the look and feel of the end result.

7.1 Future work

Although we have come a long way there are still things that we wish to do even better. The following techniques have been discussed during the development of the project.

The current terrain implementation could be replaced by a clipmap implementation. Level of detail should also be extended to not only apply to textures and geometry, but also the shaders. An example would be removing specular lighting and bumpmapping from shaders on distant terrain. An alternative to the current geomorphing LOD system is alpha blending, which would make transitions between LODs not require that morphing is handled, but of course means we'd have to handle transparency instead.

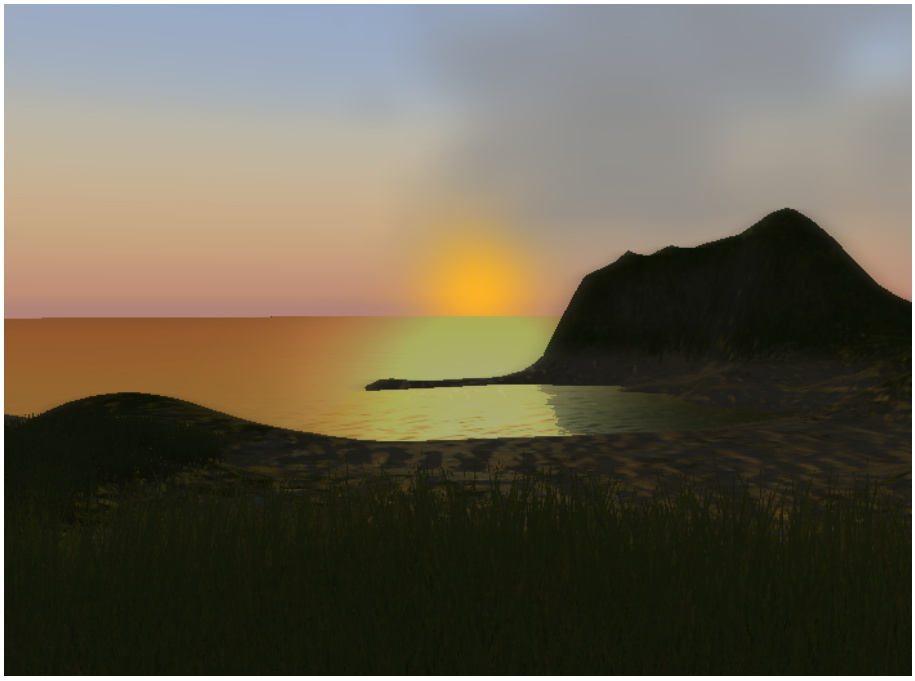
Our sky implementation currently have two minor visual defects: The first problem is that the atmospheric dome do not take the angle of the sun into account. This results in a uniform coloring of the dome in the vertical direction on the entire sphere. A more realistic coloring would be to only color the dome shades of yellow and red in a radius around where the sun is setting or rising. We have also talked about going even further an implementing a full atmospheric scattering simulation algorithm like the effect described in [PF05, chapter 16]. The second problem is that the texture mapping of the cloud dome does not have the right perspective. To fixed this we have talked about using the texture mapping scheme based on a plane as described in [Tru07, page 32].

When generating the cloud texture we use value noise as suggested in the web page that inspired our work[1]. Some authors however use Perlin noise to generate clouds, which could be interesting to implement and compare with our current value noise.

The next step for the `PostProcessNode` is to allow it to be placed anywhere inside the scene graph and then merge the post processed image with the rest of the scene. An optimized versions of some of the most commonly used effects could also be made to increase performance and in the case of Depth of Field remove the flickering that comes from only sampling one focus point.

During our work with texture binding the idea of creating a texture binding manager that reuses *texture unit* using `glActivateTexture`[SWND06, page 440] has surfaced. This manager could cache texture ids and texture units so the last x number of bound textures more quickly can be activated.

[1] `http://freespace.virgin.net/hugo.elias/models/m_perlin.htm`



Appendix A

OpenGL extensions

A.1 GL_EXT_texture_array

The texture array extension does just what its name suggests, it allows us to store several textures in an array. This is an improvement in several ways. Firstly we only need to call glBindTexture only once and then all the textures in the array are at our disposal. Secondly the GPU only allows a certain number of textures to be bound, usually 8 or 16. Using a texture array only takes up one of these slots. In the case of our terrain, using texture arrays allows us to have an array of 4 color textures; sand, grass, snow and cliffs. For each of these we're using a normal map to do bump mapping, which can also be stored in their own texture array. We can then bind these two arrays, effectively giving us 8 textures at the price of 2 texture slots.

A.2 GL_ARB_framebuffer_object

FrameBuffers allow programmers to render colors or depth directly to textures, instead of rendering to the a renderbuffer and then performing an expensive copy to texture. This is useful when creating post processing effects such as depth of field or motion blur, since the effect is a shader applied to the rendered image.

To facilitate easy setup of frame buffer objects, FBO's, we have created the `FrameBuffer` abstraction in OpenEngine. Users only need to specify what dimensions their frame buffer object should have, how many color buffers should be attached, wether or not they want to use a texture for the depth buffer and then OpenEngine will setup the entire fbo. This abstraction makes devlopment less error prone to create FBO's and attach textures to them.

Appendix B

Resources

Our project makes use of a lot of textures. These textures have been obtained in several places and here we would like to give credit for them where credit is due.

The following textures have been created using FilterForge[1]: `dirt.tga`, `dirtNormals.tga`, `newGrass.tga`, `newGrassNormals.tga`, `rockface.jpg`, `rockfaceBump.jpg`.

`EarthClearSky2.png` was borrowed from the open source framework Caelum[2].

`waterDistortion.jpg` and `waterNormalmap.jpg` where both found at Michael Horsh blog[3].

`grassStraw.tga` is part of the free version of Unity3D.

`sand-old.jpg`, `sandNormals-old.jpg` and `snow.tga` where all found on Psionic's 3D Game Resources[4].

The clouds, stars and snow normalmap have all been generated by us.

[1] `www.filterforge.com`
[2] `http://www.ogre3d.org/tikiwiki/Caelum`
[3] `http://www.bonzaisoftware.com/water_tut.html`
[4] `http://www.psionic3d.co.uk`

Bibliography

[Aba06] J.A. Abad. A fast, simple method to render sky color using gradients maps. 2006.

[AMHH02] Tomas Akenine-Möller, Eric Haines, and Natty Hoffman. *Real-Time Rendering*. A. K. Peters, Ltd., Natick, MA, USA, 2nd edition, 2002.

[Ebe03] D.S. Ebert. *Texturing & Modeling: A Procedural Approach*. Morgan Kaufmann Pub, 3rd edition, 2003.

[Fer04] Randima Fernando. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Pearson Higher Education, 2004.

[FPTV92] Brian P. Flannery, William H. Press, Saul A. Teukolsky, and William Vetterling. *Numerical recipes in C, The Art of Scientific Computing*. 2nd edition, 1992.

[Ngu07] Hubert Nguyen. *GPU Gems 3*. Addison-Wesley Professional, 2007.

[Per85] Ken Perlin. An image synthesizer. *SIGGRAPH Comput. Graph.*, 19(3):287–296, 1985.

[PF05] M. Pharr and R. Fernando. *GPU gems 2: programming techniques for high-performance graphics and general-purpose computation*. Addison-Wesley Professional, 2005.

[Ros06] Randi J. Rost. *OpenGL (R) Shading Language*. Addison Wesley, USA, 2nd edition, 2006.

[SWND06] Dave Shreiner, Mason Woo, Jackie Neider, and Tom Davis. *OpenGL programming guide: The official guide to learning OpenGL, Version 2*. Addison-Wesley, 5th edition, 2006.

[Tru07] Rene Truelsen. Real-time Shallow Water Simulation and Environment Mapping and Clouds. Master's thesis, Department of Computer Science, University of Copenhagen, 2007.

[Wag03] Daniel Wagner. Terrain geomorphing in the vertex shader. 2003.