The Corn Hungry Martians

Morten Poulsen
mrt@di.ku.dk
Department of Computer Science, University of Copenhagen
January 11th 2008

Abstract
A game engine is one of the central building blocks of a computer game. It supplies game content developers with an interface that can steamline the path from ideas to the computer screen. The subject of this paper is the construction and implementation of a light weighted game engine, and it was written as a part of the course Game Animation at the Department of Computer Science at the University of Copenhagen. During the period of the course a simple game engine has been developed from a basic framework. The engine supports the required functionality and a few additional features. The overall design could use a little extra work, and should be expanded to support world building separate from the code. The most glaring problem of the engine is that it seems to “freeze” when new cells are generated.

Contents

1 Introduction 3
2 General methods 4
  2.1 GPU or CPU? 4
  2.2 Textures and tileability 5
  2.3 Illumination 6
  2.4 Perlin noise 7
3 Building a Game Engine 10
  3.1 Limitations 10
  3.2 The avatar 10
  3.3 Time 10
  3.4 Terrain 10
  3.5 Sun, Skydome and clouds 12
  3.6 Grass and Corn 14
  3.7 Rivers, ponds and oceans 15
  3.8 Explosion 15
  3.9 Building a weather system 16
4 Assessment 21
  4.1 How much has been implemented? 21
  4.2 Under the hood 21
  4.3 Usability for a world builder or level designer 21
  4.4 General performance for an end user 22
  4.5 Conclusion 22
1 Introduction

The availability of affordable personal computers combined with rapidly improving hardware has paved the way for a computer game industry with annual growth rates of 7% and sales of $7.4 billion in the US alone [1]. With the rapid developments in hardware, the visual quality of computer games has improved vastly over the last few years, and with it the need of useful game engines. Ideally, a game engine could be used for a variety of games, such as Gamebryo that has been used for games as diverse as Sid Meier’s Civilization IV [3] and The Elder Scrolls IV: Oblivion [15].

This report has been written as a part of the course Game Animation at the Department of Computer Science at the University of Copenhagen, during which a light weighted game engine has been developed. This report will discuss some of the techniques and issues that come up when building a game engine.

During development it became clear that building a game engine that can render a realistic scene is not done by doing using (or even approximating) scientific principles, but by finding a hack that makes the scene look real, as the “correct” calculations are still beyond the computational ability of current hardware.

The game engine were build using Open Tissue [9] and a project framework developed by Kenny Erleben, Knud Henriksen and Lars Schjøth. The report will not document the project framework, so the reader is assumed to be familiar with it. The game engine was build in collaboration with Dirk Hasselbalch, Stefan Lemvig Glimberg and Toke Mundt Stensgaard Nielsen.

The report comes with a CD, containing the following:

- The source code of the engine (as well as the handed out framework)
- Our modified OpenTissue code
- Installers for externals used by the OpenTissue version that we used: Cg, Boost and CMake
- A few demos

Figure 1: A boid looking for corn.
2 General methods

This section will cover some of the general methods and techniques used in computer graphics.

2.1 GPU or CPU?

As pointed out in the literature, GPUs are getting faster and better at higher rates than CPUs [7]. GPUs are also hardware-optimized to perform operations useful for graphics, and it is getting more and more common to place more and more real-time graphics calculations on the GPU, allowing the CPU to use more work on other operations, such as AI or a physics engine.

There are two processors on the GPU: The vertex processor and the fragment (or pixel) processor. Programs on the vertex processor are executed once per vertex, and programs on the fragment processor are executed once per pixel. Output from the vertex processor is interpolated before execution on the fragment processor. This adds a few extra pitfalls for the programmer, as the bottleneck of a program can be placed on the CPU, either of the GPU processors or load of data transmission. The model is illustrated in figure 2.

![Figure 2: The figure shows the programming model used in Cg. The figure is from [7]](image)

A central question for the game programmer is which parts of the program should be performed on which processor. As a general rule of thumb, we wish to perform as few calculations as possible. So, if a calculation is only performed a few times per frame, it would be prudent to place the calculations on the CPU, and transfer the results to the GPU as a uniform parameter. As programs on the vertex processor are executed once per vertex, it follows that expensive computations where interpolation is acceptable should be placed in the vertex program whenever possible.

Shading languages

A number of different shading languages exist. For real time rendering it seems that nVidia Cg, OpenGL Shader Language (GLSL) and Microsoft’s High Level Shader Language (HLSL) are the most common. In this game engine, we have used Cg, as OpenTissue (and the framework) provides a nice interface. While Cg has a syntax very close to C, there are some notable performance differences that the programmer should be aware of. One important difference is that branching is more expensive, and that it must be possible to unroll loops at compile time. Also, vector operations run in parallel.
2.2 Textures and tileability

Most meshes in a game world will have a texture wrapped around them. If larger areas are to be displayed, there are two options: Either use larger textures or place the textures adjacent to one another. Larger textures increases the data transmission between the GPU and the CPU, and binds memory on the GPU. So, we would prefer to use smaller textures, placed side by side. This is called tiling. For this to look nice the following properties should hold:

1. It shouldn’t be possible to see seams between the textures (ie. no discontinuity).
2. Tiling shouldn’t be obvious (ie. no recurring pattern).

In the following it will be assumed that we are using a 2 dimensional texture.

Removing seams

If we let \( f(x, y) \) be a function that describes the colors of a texture with height \( h \) and width \( w \), \( f \) must satisfy these properties to be tileable:

1. \( f(0, y) = f(w, y) \)
2. \( f(x, 0) = f(x, h) \)

A simple way of removing seams is to open your texture in an image editor (ie. Paint), double the height and width, and place rotated versions of the original texture in the new white spaces. This quadruples the size of the texture, and adds obvious patterns, which usually looks bad. An example can be seen in figure 3.

![Figure 3](image-url)

**Figure 3:** The image to the left is the original image showing cracked earth. The middle image is a tileable version with obvious patterns. The right image shows how it looks when the tileable version is tiled. The original texture is from [6]

If one has access to a texture four times as large as the one to be used (or if it is defined by a function, like Perlin noise), there is a much better way. If we place the origin in the middle of our texture, as illustrated in figure 4, and call the big texture function \( f(x, y) \). The small and tileable texture function is called \( g(x, y) \).

![Figure 4](image-url)

**Figure 4:** The rectangle represents a large texture, that we wish to make into a smaller, tileable texture. The origin is placed in the middle, and the quadrant marked by + will contain the tileable texture.
We can construct a simple, linear weight function that favors values close to the axis, i.e. when $x$ is small, $g$ is close to $f(x, y)$, and when $x$ is large, $g$ is close to $f(x - w, y)$:

$$g(x, y) = \frac{(w - x)(h - y)f(x, y) + x(h - y)f(x - w, y) + xyf(x - w, y - h) + (w - x)yf(x, y - h)}{wh}$$

### Removing patterns

Patterns are easily visible by the human eye and should be disguised. As noted in [16] patterns can be disguised by reducing contrasts and removing obvious patterns in the texture. One way of doing this is to use the same texture twice, once at normal size and once at a much smaller size, with weights depending on the distance to the camera:

$$\text{color} = \frac{\text{texture}(x, y) + \text{distance scale} \cdot \text{texture}(x/4, y/4)}{\text{distance scale} + 1}$$

This doubles the number of texture look-ups, but is fairly effective in masking patterns in a texture, as illustrated in figure 5.

![Figure 5: The picture to the left uses one texture, and there are obvious patterns in the grass. The picture to the right shows how an extra texture lookup can mask the patterns. The pictures are screenshots from an early build of our game engine.](image)

### 2.3 Illumination

In order to render a beautiful scene, realistic illumination is a must. Illumination is generated by one or more light sources that emit photons. When these photons collide with a surface some are reflected, and others are absorbed by the surface. The reflected light collide with other surfaces and some is reflected (and so on as illustrated in figure 6). The colors we see come from the reflected light that reach our eyes.

![Figure 6: What the human eye see is reflected light.](image)
While physical illumination would yield very realistic results, it should be clear that it is very expensive to calculate. Thus, we use a few tricks to make acceptable lighting. From [4] illumination from a light source can be calculated as a sum of an ambient, a diffuse and a specular contribution.

The diffuse light models the self shadowing of the object, the specular light models highlights, and ambient light models the effect of reflected light. For a given component of color \( \lambda \), the illumination on a point \( p \) on an object from a point-sized light source can be calculated as:

\[
I_{\lambda} = \text{Ambient} + \text{Diffuse} + \text{Specular} = k_aO_{\lambda} + f_{att}I_{\lambda} + k_dO_{\lambda} (\vec{N} \cdot \vec{L}) + k_sO_{\lambda} (\vec{N} \cdot \vec{H})^n
\]

where \( I \) is the intensity of the light source, \( n \) is the specular exponent of the object, \( O \) is the color of the object, and \( k \) is the reflection constant of the material. All vectors are assumed to be normalized, \( \vec{N} \) is the normal of the object at \( p \), \( \vec{L} \) is the vector from \( p \) to the light source, \( \vec{V} \) is the vector from \( p \) to the eye, and \( \vec{H} = \frac{\vec{L}+\vec{V}}{|\vec{L}+\vec{V}|} \), as illustrated in figure 7. If more than one light source is present, the contributions are summed. For increased efficiency, the equation from [4] is simplified so that we ignore \( f_{att} \) and roll \( k \) and \( O \) into one material variable.

The model has some drawbacks, especially when the object to be illuminated is transparent (such as water). In this case, everything on a surface pointing away from the light source will be very dark. This can be rectified by using a trick inspired by [10]: By changing the diffuse dot-product from \( \vec{N} \cdot \vec{L} \) to \( 0.5 \cdot (1 + \vec{N} \cdot \vec{L}) \), we can increase the lighting on the surface pointing away from the light source, while still showing surface structure.

If the light source is very far away, such as the Sun or the moon, the calculations can be optimized by using a constant light vector. Also, not all surfaces has an obvious specular contribution, so it would be prudent to write an illumination function that doesn’t calculate the specular contribution.

As it is most convenient to send the coordinates and normals of an object to the shaders in the objects own coordinate system, great care should be taken to transform the normal (and the position) of the vertex into eye coordinates for the illumination calculation. The modelview matrix transform from object coordinates into eye coordinates, and should be used to transform positions. The inverse transpose of the modelview matrix should be used for normal so that angles between vectors are conserved.

An alternative is to transform the light vector into object coordinates. This can be done on the CPU, and would save a few calculations on the GPU. This optimization haven’t been implemented due to time constraints.

### 2.4 Perlin noise

Perlin noise were introduced by Ken Perlin in [11], and improved in [12]. Perlin noise can be viewed as soft noise, that varies somewhat continuously and pseudo randomly. This gives it some very useful properties. If one applies Perlin noise to a point on a surface the pseudo random property ensures that a later call to the same point will yield the same result. If called ”close” to the previous point, the value of that point is ”close” to that of the previous point. A Perlin noise texture can be seen in figure 10.
Calculating Perlin noise

In two dimensions, assume that Perlin noise is ordered in some grid. In each of the grid cell corners, we put a gradient. The gradients are supposed to be pseudo random, and can be calculated by using a hash function that uses a permutation table to choose between vectors from the center of a grid cell to the middle of the grid cell wall.

To find $\text{noise}(x, y)$, where $x, y$ are not integers, find the grid cell containing $(x, y)$, and generate the gradients of the grid cell corners. Next, we dot the gradients of each corner with the vector from that corner to $(x, y)$, shown in figure 8. From here we interpolate the four scalars in the $x$ and $y$ direction to find the value at $(x, y)$.

![Figure 8: A grid cell where $(i,j) = \text{floor}(x,y)$ showing gradients and vectors from grid cell corners to $(x,y)$](image)

To improve softness of the noise, we’ll want a function that is dominated by a grid cell corner when close to it. We’ll also want it to have no visible discontinuities between cells, i.e. first and second derivatives should be equal. This can be accomplished by a 5th order polynomial, such as $s(t) = 6t^5 - 15t^4 + 10t^3$, as weight function for the interpolation [12]. Using this function has the additional consequence that $\text{noise}(x, y) = 0$ if $x, y$ integer [18].

It should also be noted that Perlin noise can be applied in more than 2 dimensions. An extra dimension could be used to texture a 3D shape or a time dependent noise.

![Figure 9: When approximating the gradient of a texture in a point p, we look at points a bit to the left, right, above and below p.](image)

Useful functions

Perlin also suggested a number of useful functions using Perlin noise. By calculating the (approximate) gradient, $d\text{noise}$, of the noise function, one can make a normal perturbation that can make a smooth surface look bumpy. This gradient is calculated by evaluating $\text{noise}(p)$ at four points. If the points $p_1, p_2, p_3, p_4$ are placed as in figure 9, $d\text{noise}$ can be calculated as:

$$d\text{noise} = \begin{pmatrix} \text{noise}(p_1) - \text{noise}(p_2) \\ \text{noise}(p_3) - \text{noise}(p_4) \\ 0 \end{pmatrix}$$
The *turbulence* can be calculated by summing a progressively smaller part of the noise-texture with a progressively smaller weight:

\[
turbulence = \sum_{i=0}^{n} \frac{\text{noise}(\vec{p}/2^i)}{2^i}
\]

where \( \vec{p} \) is a point. Turbulence is useful in creating something with a chaotic look, such as fire, clouds or even marble. A turbulence and Dnoise texture is shown in figure 10.

**Figure 10:** From the left we have a Perlin noise texture, a turbulence texture and a Dnoise texture. All textures are tileable.

**Implementing Perlin noise**

Perlin noise can be implemented on the GPU [5], but these computations are a significant drain on GPU resources (our initial implementation used more than 80 instructions and close to 500 instructions for turbulence), and doesn’t work on older GPUs. To ensure better performance and higher compatibility, 2 dimensional Perlin noise textures will be generated at startup, and passed to the shaders that need them. The OpenTissue class ImprovedNoise is used to calculate the textures. Transmitting textures will add some data transmission, but if the textures are kept a reasonable size, such as \( 256 \times 256 \) pixels, it should be negligible. Perlin’s article on GPU implementation of Perlin noise suggests that the GPU should be used for 3D Perlin noise, where data transmission is a serious problem (\( 256 \times 256 \times 256 \) pixels is a lot of data to transmit) [13].

Turbulence and Dnoise will be computed on the GPU on the fly, as these calculations can be made quite fast.

When generating a Perlin texture, we’ll also want it to be tileable, which can be accomplished by applying a weighting function, as discussed in section 2.2.
3 Building a Game Engine

The objective of the assignment is to build a game engine that supports an infinite, interactive and natural world in real time. This section will give an overview of the engine and the theory behind it. Much of this section is based on lectures during the course.

3.1 Limitations

Building a full fledged game engine in 7 weeks is a very large undertaking – even with OpenTissue and the handed out framework. So, we’ll have to make a basic game engine with limited functionality.

The game engine will revolve around an avatar, that is assumed to move at ground level or fairly close to ground level. Thus, our engine not will be made for a flight simulator.

Each effect will be implemented in just one pair of shader programs, so a future user will not be able to remove effects for higher frame rates, nor will we take advantage of the engines ability to detect the user’s shader profile to enable or disable features.

All contemporary computer games have sound, but that functionality will not be implemented for this engine.

3.2 The avatar

The avatar represents the player of the game. Usually the player’s point of view will be close the the avatar, sometimes simulating looking through the “eyes” of the avatar. This means that the camera should be able to follow the avatar in some way. So, everything in the world should revolve around the avatar, and the “faraway” parts of the world, such as the sky dome, should be programmed to follow the avatar position. Also, the loaded parts of the world should be dependent on the avatar position.

In this game engine, the avatar will be represented by a martian spacecraft, called a boid. In most games, the avatar can “run” (i.e. move faster) if “Shift” is held down. This will be supported in this game engine. As the boid is a spacecraft, it should also be able to move up and down. To ease generation of demos, automated movement of the boid, using bsplines, has been implemented.

3.3 Time

Keeping track of time can be very useful. Having a clock allows for consistent control over celestial movement, and can tie it together with movement in plants, water, clouds and so on. For convenience, it should also be possible to invert time (make it move backwards), accelerate, decelerate and freeze time – freezing time were already implemented in OpenTissue.

To ensure that the clock doesn’t keep ticking when time is frozen, we maintain a variable saving the time in the previous frame. We define that 24 time units correspond to one 24 hours day/night cycle. Note that avatar movement will be independent of time.

3.4 Terrain

We’ll need terrain for our virtual world. As the avatar moves, we’ll have to generate new terrain and throw the old terrain away. This can be done by partitioning terrain into cells, only rendering those cells close to the avatar. If the avatar moves out of a cell, the most distant line of cells can be deleted and a new line can be added, as shown on figure 11.

When generating the new terrain, there is a short freeze in the game engine. This phenomenon is present in the handed out framework as well. The culprit is the function `make_grass` that (for each cell) first adds 5000 `vector3` type to two empty `std::vector`s, then (if needed) constructs a field of 20,000 vertices and 30,000 indices, and continues to plant grass at all these vertices. The function can be optimized by:

- initializing the `std::vector`s to the capacity we expect them to use
- adding logic to ensure reuse of field
- adding logic to update the grass positions in the field (the current implementation inserts new values into the field)
• reduce the number of grass plants planted in each triangle

It seems that the problem may be the sheer volume of data usage. Due to time constrains neither of the above optimizations have been implemented.

Terrain features
The terrain should not just be flat. There should be hills, plains and valleys. This can be accomplished by using Perlin noise. Each $x, y$ point in the terrain could be used as arguments to a Perlin noise function to yield a height $z$. This gives the terrain a reasonably soft appearance. This was already implemented in the handed out framework.

We would also like to have rivers, cornfields, roads and so on. So, some additional control could be added by using a tileable height-map texture to ensure existence of the structures we wish, while still getting the soft pseudo-random look of Perlin noise, and an (in principle) infinite world.

Terrain textures
Showing a terrain isn’t just about heightmaps. In the real world we may have grass, sand, rocks, flowers etc. To reduce computation we model this by adding tileable textures that look like grass, dirt or rock. These can be applied due to height, so that each texture has a height interval where it exists. Adding some blending near a border gives results that look a lot better. While a complex blending function, could be used, one gets acceptable results from a linear blending function. If Perlin noise is added to the height, the result gets a bit more chaotic, as can be seen on figure 12.
Patterns in the textures can be masked by using the techniques mentioned in section 2.2. Applying textures makes it convenient to calculate illumination in the fragment program (i.e., Phong illumination), as we wouldn’t like to stretch the textures, even though it makes the terrain slower. Some improvement can be gained by ignoring the specular contribution. This can be justified by noting that most terrain features aren’t smooth, and so the specular contribution would be reflected in many different directions.

### 3.5 Sun, Skydome and clouds

A world should have a Sun, a Moon, clouds and stars at night.

As we would like the clouds to be in front of the Sun and Moon, and the Sun and Moon in front of the stars, we’ll use two concentric skydomes, with the Sun and Moon moving between them. To get a proper day/night cycle, the elements must be tied to the time function mentioned in section 3.3. The Sun and Moon are also sources of light, and their positions and properties should be tied to the light sources. It must also be modeled that sunlight seems more reddish at dawn and sunset.

**The outer skydome**

The outer skydome will display the blue day sky or the starlit night sky. Around dawn there should be some blending, so the transition isn’t too sudden. One could also vary sky color depending on distance to the Moon or the Sun. This isn’t done at present. For the night sky we will need to apply a texture. As textures are rectangular and our sky dome is a sphere, we’ll need a clever way to apply the texture.

One option is to place a plane above the sphere with the texture, and then, for each point $p$ on the sphere, calculate the point $p_t$ where a line from the origin through $p$ would intersect with the texture plane, as illustrated in figure 13. Then $p_t$ can get the color of the texture at $p$. This gives a problem with undersampling. One solution is to start the line at the lowest point of the sphere. This significantly reduces the problem for points above the $xy$-plane, as seen in figure 13, but it’s still there for points below the $xy$-plane. As the boid is supposed to move on the terrain, this is acceptable.

![Figure 13: Two ways to apply a texture to a sphere shown in 2 dimensions.](image)

**The inner skydome**

The inner skydome will display the clouds. To render the clouds we will use a tileable cloud map provided by the framework. This gives decent results fast with few calculations unless the camera tries to fly through the clouds. The time function will be used to create a feeling of moving clouds by letting the lookup position be time variant. To make the movement less repetitive, two texture lookups with different speeds will be used.

When applying the cloud texture to the spheric inner skydome, we would like to get an effect of clouds reaching to the horizon. To achieve this effect, we do two things. First we calculate the cloud-texture effects, and then – depending on the height of the point – we
blend it with a horizon color. To calculate the texture coordinates, we place an imaginary plane $\beta$ that is tilted by a small angle $\alpha$ with respect to the tangent plane $\tau$ of the topmost point on the sphere, as illustrated on figure 14. The $u'$ coordinate of our texture lookup is the intersection between $\beta$ and the line from $O$ to $R$.

![Figure 14: Applying cloud textures to a sphere using a tilted texture plane.](image)

From figure 14 it is obvious that $OPS$ and $OTR$ are equiangular triangles. From this it follows that:

$$\frac{u'}{x'} = \frac{z'}{r} \Rightarrow u' = \frac{x'r}{z'},$$

where $z' = z + x \sin \alpha \approx z + x\alpha$ and $x' \approx x$ if $\alpha$ is small. From this it follows that:

$$u' \approx \frac{x'r}{z + x\alpha},$$

As $r$ is just a factor on our texture lookup we discard it in the numerator. To ensure against the denominator becoming zero, we use $r$ in stead of $x$ in the denominator.

**From dusk till dawn**

Illumination of clouds is very expensive and can’t use our illumination functions (see section 2.3), as clouds scatter light inside them. We make a hack by defining cloud colors at midnight, noon and dawn/dusk, and blend them using weights dependent on where how high the Sun is on the sky. As we are blending three values, `lerp` can’t be used directly. In stead we use the following function:

![Figure 15: Screenshot of a dawn.](image)

```cpp
float4 get_cloud_color(float4 t_night, float4 t_dawn, float4 t_day, float t_param)
{
    return lerp(t_night, t_day, t_param)
}
```
+, saturate(-4 * t_param * (t_param - 1.0)) * t_dawn;
}

where \( t_{\text{param}} \) is the interpolant. For the horizon this will be the height of the Moon, and for the clouds the height of the Sun, both ranging from 0 to 1. The first term is a linear interpolation between night and day color, and the second term is a second order polynomial \(-4x(x - 1)\) that has its maximum at 0.5. A screenshot of a dawn can be seen in figure 15.

### Sun and Moon

The Moon can be tied to a Moon texture. The Sun, however, should shift color depending on the time of day, i.e. reddish at dawn and a bright white at noon. To make the Sun more orange at dawn, we multiply the blue and green color components with sun height to the 4th and 2nd power, respectively (orange has rgb code ffa500).

It would also be nice to have an active corona, with a flow from the center to the surface. This flow can be calculated by converting to polar coordinates, and adding a linear time-dependent perturbation term to \( \text{radius} \) and \( \theta \):

\[
\text{radius} = \text{radius} - \text{speed} \cdot \text{time} \\
\theta = \frac{\text{speed} \cdot \text{time}}{2\pi}
\]

If we use the perturbed coordinates to perform a lookup in a turbulence texture (see section 2.4), we'll get a seemingly chaotic radial flow. We add this value to the radius to look into a corona texture (containing colors from white to bright yellow) and calculate blending. In figure 16 one can compare our Sun with a real world corona and Sun. As can be seen there are differences both in color and flow (of the snapshots).

![Figure 16: The left photo is from the Solar Eclipse in France 1999 (photo by Luc Viatour). The middle photo shows the Sun (photo by Lykaestria), and the right image is from our game engine.](image)

### 3.6 Grass and Corn

In section 3.4 we placed textures on the terrain to make it look better. We would also like to add some grass and corn plants. As noted in the literature, i.e. [10], modeling a field of grass as individual grass straws isn’t displayable in real-time on current hardware as the polygon count would be too high.

An alternative is to use billboards. Billboards are images that are placed in the virtual world. When the camera is moving on the ground, the billboards should be rotated to face the camera. If the camera is placed looking down on the billboard, the grass would seem very thin. One solution to this is to cross the grass billboards. Even better results can be accomplished by more complex billboard shapes, as seen in Oblivion. This effect haven’t been implemented. By using alpha-blending, we can make distant grass seem more fuzzy (and eventually fade out), as well as giving a more natural appearance when the camera is close [10].

Real grass and corn move a bit in the wind. So, we’ll want some animation of the billboards. This can be done by using time dependent trigonometric functions in the vertex shader. To
make the grass seem more realistic, we can add a tint to vary the color of the individual grass straws.

### 3.7 Rivers, ponds and oceans

A large proportion of the surface of Earth is covered by water. Ponds, rivers and oceans are a central part of our world, and should be supported by any game engine. The main ingredients of good looking water are waves, reflection and transmission.

As noted in [2] waves can be modeled by summing sine waves. This also has the advantage of giving an analytical expression for the normals, making it easy to apply the illumination model from section 2.3. Summing sine waves gives very round waves, which is fine for ponds and rivers, but not for ocean waves. As an alternative Gerstner waves could be used. These introduce extra calculations, but adds more variety and freedom.

Another possibility is to use normal maps (i.e. textures that are used as surface normals), which can give very realistic water. The drawback of normal maps are the possibility of obvious patterns, when many maps are tiled (see section 2.2). As normal maps are fairly cheap to use, we’ll use normal maps in this game engine. To simulate the feeling of real water oscillating at the riverbank, we’ll add a sine wave.

When light hits a water surface some of it is reflected and some is translated. This gives an effect of both transparency and reflection of nearby terrain. Translation of light implies that it’s possible to see what is below the water line. This can be approximated by adding some blending. Water also absorbs light, and may even be polluted, which can be approximated by letting the blending be dependent on the depth of the water.

Water can reflect the nearby scenery. The reflected scenery can be approximated by generating a reflected texture and applying this to the water. The texture can be generated by mirroring the camera position in the water plane, removing everything below the water line, and render a frame to the texture. To ensure a reasonable frame rate, not all of the scene should be rendered.

![Figure 17: The screenshot shows an explosion.](image)

### 3.8 Explosion

After implementing a corona, it’s fairly simple to implement an explosion. First, we have to make sure that when the explosion starts, it actually expands out to a maximum size and then collapse. This can be done on the CPU by calculating a time dependent factor on the radius of the explosion. If \( size \) is a coefficient controlling the size of the explosion, the radius factor could be calculated as:

\[
\text{radiusFactor} = size \cdot \text{timeFactor}^2 \cdot (1 - \text{timeFactor}),
\]

where \( \text{timeFactor} = \frac{\text{currentTime} - \text{explosionStartTime}}{\text{explosionDuration}} \).

To ensure that the explosion movement is more chaotic in the explosion fragment shader than in the corona shader, the turbulence contribution is cubed and multiplied with the cubed \( \text{radius} \) and \( \text{radiusFactor} \):

\[
\text{modifiedRadius} = \text{turbulence}^2 \cdot \text{radius}^2 \cdot \text{radiusFactor}
\]

The modified radius is used to calculate the color of the particular point of the explosion and blending. The effect is illustrated in figure 17.
3.9 Building a weather system

A believable world has a weather system. While designing and implementing a complete, self-modifying weather system is beyond the scope of this assignment (and most game engines), some basic controls should be implemented, so that a world builder could design a weather pattern as needed. Some weather types should also influence other parts of the solution, i.e. a thick cloud cover should reduce lighting from celestial light sources.

This section will discuss the construction of a weather system. The first subsections will discuss some general mechanics will be discussed, and then these will be applied to sample weather types in the following subsections.

Transitions

Usually, it isn’t a sunny day one second, and a snow storm the next. So, there will need to be fairly smooth transitions between different weather types. This can be handled by maintaining a current weather, a target weather and a speed denoting how fast the change occurs, i.e. a cloudCoverage and a cloudCoverageTarget to handle the thickness of the cloud cover.

In each frame the current weather can be updated by calculations of the type:

\[
\text{currentWeather}(\text{time}, \text{speed}) = \text{currentWeather} + \Delta \text{time} \cdot \text{speed} \cdot \Delta \text{weather},
\]

where \( \Delta \text{time} \) is the time passed after the last update and \( \Delta \text{weather} = \text{targetWeather} - \text{currentWeather} \). As \( \Delta \text{weather} \) is calculated in each frame, it will decrease as \( \text{currentWeather} \) gets nearer to \( \text{targetWeather} \), which means that \( \text{currentWeather} \to \text{targetWeather} \) when \( \text{time} \to \infty \). The speed can be used to control the softness of the transitions. An alternative would be to calculate \( \Delta \text{weather} \) when the weather changes, or even having a constant \( \Delta \text{weather} \), but this would add an extra variable per weather type.

This solution will have some problems with the invert time feature (section 3.3) – unless a variable is used to save the previous target and is applied when \( \Delta \text{time} < 0 \). This feature will not be implemented.

Usage

To ensure a consistent weather system, the user should not be able to edit the current weather directly, but set targets for the weather system, and speeds of transitions. To make it even easier, methods should be constructed to target certain types of weather, i.e. a cloudy day or similar.

Clouds

Cloud coverage should be adjustable, as different weather types will have a different cloud coverage. There should also be a connection between the fraction of the sky covered by clouds and the celestial light at the surface of our world. A simple implementation would be to multiply light intensity by \( 1 - \text{cloudCoverage} \).

Figure 18: Clouds of a coming storm.
It should also be possible to control how dark the clouds should be. This is implemented by maintaining one variable, that is sent to the cloud shader as a uniform variable, and used for a linear interpolation between the time dependent cloud color (see section 3.5) and a dark cloud color. The effect of this can be seen on figure 18.

**Rain**

Rain is rendered by applying a texture to two cones, placed with the camera inside.

When it is raining, one would expect to see a thick layer of gray clouds. With cloud controls implemented this can be done by setting the cloud cover target to a reasonable value.

It should also be possible to control the falling speed of the rain drops. While this is fully possible and easy to implement, varying the speed gives very ugly results, due to the way the rain shader is implemented.

After a bit of raining, puddles will start to emerge, and after longer periods small lakes, even floods if the ground is unable to absorb the water dropping from the sky. This effect will be modeled by raising the water level while it rains. Due to the way cells are given properties, this will have to be controlled by static methods.

Rain also plays a role in how fast snow melts. This can be implemented by multiplying the melting speed by the raining speed.

**Snow**

Snow is rendered similar to rain by applying a texture to two cones, placed with the camera inside.

As with raining, it is usually cloudy when snowing. While the cloud cover will reduce light from celestial light sources, the light that does get through will get reflected by the snow on the terrain. So, the reduction should be lower as the snow coverage increases. This has not been implemented.

It should also be possible to control the falling speed of the snow flakes. Snow flakes seem to fall slower than rain drops.

![Figure 19:](image)

*Figure 19: The screenshots show how the snow coverage looks after different periods of snow fall.*

After a bit of time, snow will start to pile up on the ground. Which parts of the terrain to be covered first is controlled by numerous factors, among these are the slope of the surface and the wind direction. For now, we’ll only look at the slope of the surface. When it stops snowing (and if it is warm enough), the snow will start to melt. To simulate snow coverage, we make a variable, `snowCoverage`, that keeps track of how much snow has fallen. This variable can be updated each frame, increasing while it snows (or until everything is covered in snow), and decreases when it doesn’t snow. If it should start snowing the variable should start increasing again.

The variable can be accessed by shaders to calculate snow coverage. A simple way to show snow coverage on the terrain would be to make a linear interpolation between the color (due to the textures, see section 3.4) and snow color (white). The interpolation value would be based on snow coverage and slope of the surface of the terrain:

\[
\text{color} = (1 - \text{snowFactor}) \cdot \text{textureColor} + \text{snowFactor} \cdot \text{snowColor}
\]
where $snowFactor = snowCoverage \cdot \vec{N}(0.3, 0.3, 0.9)$, $\vec{N}$ is the normal of the surface and the last vector is used to favor surfaces with vertical normals, and can be seen as a terrain dependent factor.

This very simple (and computationally cheap) model yields reasonable results, as can be seen on figure 19. In [8] Ohlsson and Seipel proposed a more realistic model of calculating snow accumulation by using depth maps to calculate coverage. They conclude that their method (in 2004) were to slow for current hardware.

Wind
Wind should play a role in movement of plants and clouds. Note, that wind at the surface isn’t necessary the same as wind at cloud level.

A simple way of implementing it for clouds and water could be to send the wind to the vertex shader as a uniform parameter. As the vertex shader already has a wind parameter (constructed from two contributions), we just multiply it with the new wind parameter.

Wind could also be implemented to have influence on rain and snow speed and falling direction. As rain and snow falling direction isn’t supported, this will not be implemented.

Due to the problems with changing speeds of clouds and water, wind will not be implemented with a target – it will just be set directly.

Temperature
A player in a game world will not be able to see a direct effect of temperature, in the same way as the player will be able to see the effects of wind, rain and snow. But temperature plays a subtle role in a few instances. How fast (or if) snow melts is highly dependent on the temperature. Also, the game developers may want to implement the possibility of frost bite, where temperature plays an essential role.

Fog
Fog can be used to create some nice visual effects. If one stands in a fog, distant objects seem to disappear in a grayish mass, as shown on figure 20. Fog can be described as a cloud on the ground [17]. Fogs form when the difference between temperature and dewpoint is less than $3^\circ C$, which makes the water vapor in the air condense. So, if conditions of fog formation were to be implemented scientifically correct, we would have to keep track of humidity and temperature. While this is easily doable, we would still have to keep track of fog movement, and that is more complex.

Fog dissipation is heavily dependent on humidity and air temperature. Also, wind could play a role by blowing the fog away. This will not be implemented, as a world builder may like to be able to conjure up “magical mist”, fog will be implemented to be independent of other weather variables.

Figure 20: The picture shows fog obscuring a tree and graying out everything behind it. The picture is from [17]
Fog can form slowly or very fast, so it’ll be advantageous to have a separate, user defined speed for how fast the fog is formed. The speed of dissipation and forming is dependent on temperature and wind. Fog seem to have a grayish color, though somewhat darker at night. Sometimes, however, it is useful to be able to have fog in another color, such as yellow (for swamps), so a color variable should also be included for those instances.

As water vapor is heavier than air, fog will naturally settle in lower areas (especially in hilly terrain), as seen in figure 21.

![Figure 21](image)

**Figure 21:** The picture shows how fog seems to settle in lower areas. The picture is from [17]

A simple and cost effective way to implement fog would be to make a linear interpolation between the calculated color of each (visible) object in the world and the fog color, where the weight is an increasing function of the distance from the vertex or pixel to the camera [14]. This would include a distance calculation – which is free, as we calculate the distance when projecting in the shader – and a linear interpolation in either the vertex or fragment shader (dependent on how coloration of the object is implemented – fog should be added as the last thing). As for traffic, only four \texttt{float}s would have to be passed to the GPU: One describing fog coverage and three to describe the color of the fog. If one only wanted gray fog, this could be reduced to two \texttt{float}s.

Fog is particularly easy and cheap to implement for grass and corn, as these are billboards that have a visibility distance. We simply use the fog coverage to reduce the visibility distance. This can be done on the CPU.

Making fog settle in lower areas first is a bit more tricky. We’ll have to have access to a height variable for both the terrain and any object in the terrain. This variable should be in world coordinates. For the terrain, this isn’t a problem, as the positions passed to the vertex shader are in world coordinates. The problem is with all the other objects. A hack could be to use object coordinates, but this is not certain to yield good results. A screenshot showing fog settling in lower areas is shown in figure 22.

![Figure 22](image)

**Figure 22:** Fog settling in lower areas of the world, as handled by the game engine.

**Further work**

This section discuss some expansions on the weather controller, that could be applied.
Changing winds It would be nice to have shifting winds. In the real world, wind isn’t constant on the ground. There are forceful gusts of wind as well as almost windless periods. This could be implemented by letting the wind change around its current value. The main challenge, however, is updating the water and cloud shader to handling changing speeds. This could be done by changing the code to using time dependent increments when generating the new frame. This will give the added benefit of optimizing the shaders.

Changing rain and snow speeds Currently snow and rain start in a very sudden manner, where they should start over a short period of time. This is similar to the above expansion in that the current implementation of snow and rain doesn’t support changing speeds. When this issue is resolved, shifting winds should be implemented to have an effect on rain and snow.

Wind and grass Currently, wind doesn’t affect grass, corn and other objects that should shift in the wind.

Fog dissipation Possibility for the user to allow/disallow factors on fog dissipation from wind and temperature.
4 Assessment

The assessment of the game engine will be discussed on four levels:

1. How much has been implemented?
2. Under the hood: Quality of the underlying design and code.
3. Usability for a world builder or level designer.
4. General performance for an end user (the player).

Finally, a conclusion will be formulated.

4.1 How much has been implemented?

We have been able to implement an interactive world with the following features:

- An avatar with per pixel shading and automated movement.
- An infinite world from the avatars perspective.
- Terrain with multiple layers: The bottom layer has basic textures, the others includes grass, corn and static objects such as rocks and a house.
- Moving grass and corn.
- A sky dome with dynamic cloud movements, Sun, Moon and stars.
- A day/night cycle with dawn/dusk effects.
- Realistic water with surface waves and reflection of terrain and avatar.
- A controllable weather system.

The most notable problem with the engine is the short freeze when “new” cells are generated (see section 3.4).

4.2 Under the hood

During the implementation of the game engine, we used the coding style of the framework. This had the unfortunate effect that almost everything from boid movement to light sources were handled directly in `application.cpp` with little or no encapsulation. This has turned out to be a major design flaw, adding significant clutter to `application.cpp`, and reducing reusability of code. An obvious example is that all boid movement and attributes are defined directly in `application.cpp`, even with a separate shader and geometry. As separately moving objects are used in many games, it would have been wiser to define a class for moving objects, with generic shaders, and declare and initialize the boid as one of these objects.

The classes `celestial_control` and `weather_control` are examples of how the object oriented approach should have been used to encapsulate functionality. For this game engine to be easy for a future programmer to use and expand upon the design should be reworked to adhere to the object oriented paradigm.

4.3 Usability for a world builder or level designer

As previously mentioned, most of the functionality that a world builder would need has been implemented. It’s just not very convenient for a world builder to use. Only the height map and placement of some terrain features can be edited outside of the source code. Ideally, all placement of objects and properties of the world should be saved separately from the game engine, as this would make it easy to make a graphical user interface for the world builder, increasing the productivity of the world builder. It would also allow the game developers to release a construction kit to allow players to make and distribute plug-ins, which has been used by games such as Civilization IV and Oblivion.
4.4 General performance for an end user

From the perspective of the end user (i.e., the player of the game) the most obvious problem is the previously mentioned “freeze”. From the player's point of view this is a serious problem. The player will also notice that there is no effect of being underwater. Also, the player will experience the annoying effect of grass that suddenly fade into existence when the avatar moves.

4.5 Conclusion

The game engine supports the functionality outlined in the project description, and a few things more such as a house and a weather system. The overall design could use a bit more work, and should be expanded to support world building separate from the code. The most glaring problem of the engine is that it seems to “freeze” when new cells are generated.
References


