Effects for a Game Engine

Christian Bay Kasper Amstrup Andersen

Department of Computer Science University of Copenhagen

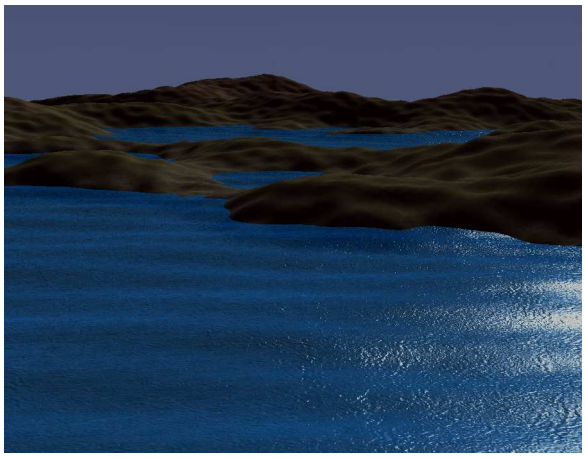


Figure 1: Coastal area. Water is animated and the sun is reflected in the small surface waves.

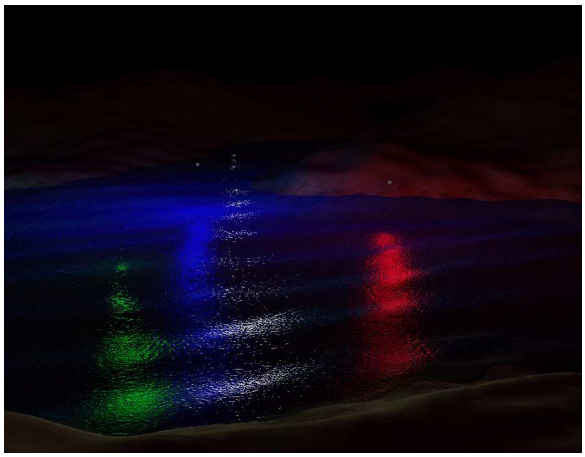


Figure 2: Multiple point light sources reflected in water at night.

Abstract

In this paper we present 3 extensions for the engine used in the Game Animation course at the Department of Computer Science University og Copenhagen (DIKU). The extensions are animated water, streaming landscape, and dynamic directional and point light sources. With the next generation of gaming consoles (Playstation 3, Xbox360 etc.) it becomes important to have a large, complex, and visually pleasing landscape. Our approach introduces an infinitely large landscape with constant framerate and no stalls at runtime. With more computational power, more resources will be used for dynamic lightning, and natural effects like water. We present simple and fast shader approaches for both of these effects.

Keywords: Graphics, shader programming, landscape, lightning, water.

1 The existing game engine

Previously, we have seen seen a large amount of games that have visually pleasing landscapes, but also lots of stalls at runtime due to geometry reorganization. As computers have grown more powerful and with the introduction of GPU's, it seems natural to try to reduce stalls by offloading landscape computations to the GPU. The GPU can continuously stream landscape such that a small part of the landscape is swapped in while another part is swapped out each time a frame is rendered.

The existing game engine used at DIKU is implemented in C++ [Stroustrup, 1997] and uses OpenGL [Shreiner et al., 2005], GLUT [Kilgard, 2001] and OpenTissue [OpenTissue, 2006]. Shaders are written in Cg [CgManual, 2005]. Landscape is implemented as a virtual grid data structure where blocks of landscape geometry are moved around at runtime. The landscape is generated with Perlin Noise [Perlin, 1985]

and [Perlin, 2002]. This means that the game engine has infinite landscape. However, when moving around the player will notice a dramatic drop in framerate due to block loading. This may be acceptable in turn based games but for realtime games, especially 3D shooting games (Half-Life, Quake etc.), this approach makes games practically unplayable. We present a method for rendering an infinite landscape without any stalls and with constant framerate at all times. Our method is inspired by [Pharr et al., 2005, chapter 2] and [Hoppe et al., 2004] but does not use clipmaps which will probably be patented in the near-future [USPatent, Terrain].

The default lightning in the existing game engine is a global point light source and multiple directional light sources stored in shaders. All light sources are static. We present a simple and fast method for dynamic directional and point light sources.

Using sums of sine waves as described in [Erleben et al., 2005] we present a computationally fast method for rendering animated water.

Screenshots from the extended engine are shown in Figure 1 and 2.

2 New game engine effects

In the following section we present our 3 extensions for the existing game engine.

2.1 Streaming landscape

Landscape is stored in GPU memory as a $k \times k$ vertex grid where $k = 2^n$ for some integer n. This grid thus contains $k \cdot k$ vertices $V_{i,j}$.

$$V_{i,j} = (i, j, 0) \tag{1}$$

where i and j are vertex grid indices and $0 \leq i < k$ and $0 \leq j < k$ as shown in Figure 3.

A combined height and normal map with size $k \times k$ is also stored in GPU memory. Both normals and height values are precomputed on the CPU, all other landscape computations are done in a vertex shader. Height values are generated with Perlin Noise but other height generation algorithms could be used as well. The map is initially transfered to the GPU with the OpenGL function `glTexImage2D`. The map is a 4 channel 32 bit floating point texture. The first 3 channels contain precomputed normal vector values, and

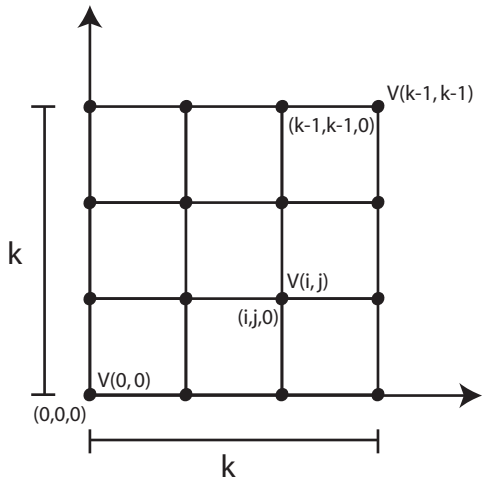


Figure 3: Vertex grid with size $k \times k$ stored in GPU memory.

the 4^{th} channel contains a height value. Thus, each vertex $V_{i,j}$ has a corresponding texel[1] $e_{i,j}$ containing the height and the normal vector of the vertex.

$$e_{i,j} = (\bar{n}_x, \bar{n}_y, \bar{n}_z, z) \tag{2}$$

where $\bar{n}$ is the normal vector and z is the height for the corresponding vertex. For a vertex, the function `tex2D` is used to look up the corresponding texel. We need the proper texture coordinates (u, v) for `tex2D` though.

$$\begin{aligned}(u, v) = (((i + pos_x)\ mod\ k)/k,\\ ((j + pos_y)\ mod\ k)/k)\end{aligned} \tag{3}$$

where $a\ mod\ b = a - \lfloor a/b \rfloor b$, and $\lfloor t \rfloor$ denotes the nearest integer smaller than t. Initially, $pos_x = pos_y = 0$ but this changes as the landscape is moved, as described in section 2.1.1. We define two functions, that will be used in the following sections.

Definition 2.1. `hecmp`$(e_{i,j})$ returns the height component z of a texel $e_{i,j}$.

Definition 2.2. `nocmp`$(e_{i,j})$ returns the normal component $\bar{n}$ of a texel $e_{i,j}$.

[1] Texture element. The texture map equivalent of a pixel.

2.1.1 Position of the vertex grid in the world coordinate system

The landscape is always centered around a position *pos*.

$$pos = (\lfloor x \rfloor, \lfloor y \rfloor, 0)^T \tag{4}$$

where (x, y, z) is a coordinate in the world coordinate system (WCS). Each vertex $V_{i,j}$ needs to be translated to the proper WCS position $V_{i,j}^{WCS}$.

$$V_{i,j}^{WCS} = pos + \begin{pmatrix} i - \frac{k}{2} \\ j - \frac{k}{2} \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ \texttt{hecmp}(e_{i,j}) \end{pmatrix} \tag{5}$$

Here *pos* is passed as a uniform parameter to the GPU vertex shader. The normal for $V_{i,j}^{WCS}$ is extracted with $\texttt{nocmp}(e_{i,j})$.

2.1.2 Computation of surface normals

A fast method for computing approximated normals for vertices on the landscape surface is to use the four adjacent vertices as shown in Figure 4. Here two vectors are created between the adjacent vertices. The cross product of these vectors results in an approximation of the surface normal.

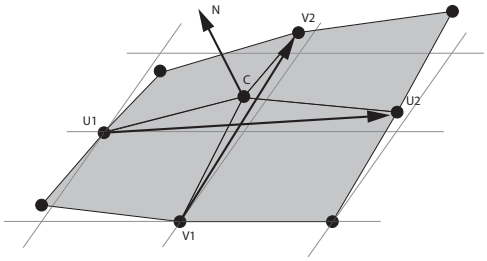


Figure 4: Approximation of the surface normal by using the cross product between $U_2 - U_1$ and $V_2 - V_1$

Normals could be computed on GPU, requiring 4 texture lookups for retreiving the 4 adjacent vertices, but since we need a texture lookup to get the height value anyway we might as well compute normals on the CPU and use the first 3 channels in the texture to store them.

2.1.3 Updating the height and normal map

The normal and height map texture needs to be updated properly. Here, we have used the OpenGL function `glTexSubImage2D`. We need to find the smallest possible area to update since updating the entire texture at each movement step degrades performance dramatically. This can be done by updating a wrapped around region of the texture, using modulo arithmetic. For simplicity, we will use a 2D example.

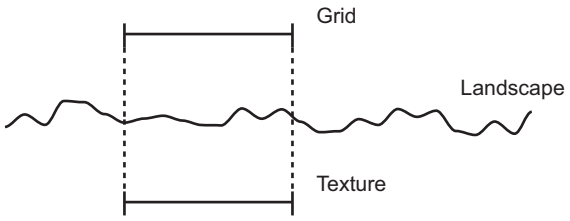


Figure 5: Initial situation. Landscape heights and normals are stored in the texture.

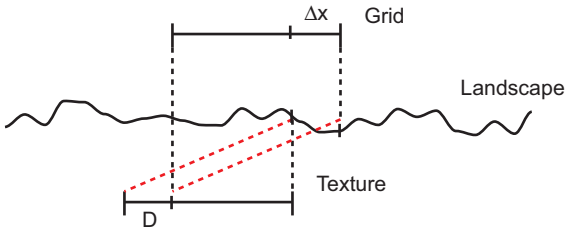


Figure 6: Situation after the landscape is moved Δx units further. The area D in the texture is updated.

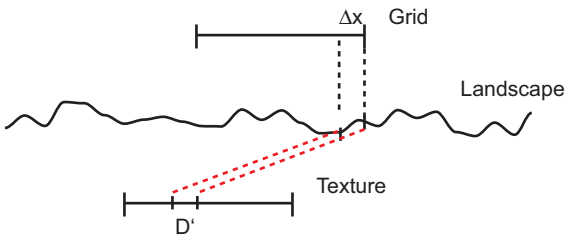


Figure 7: Situation after the landscape is moved another Δx units. The area D' in the texture is updated.

Initially, we have the situation depicted in Figure 5. Then we move the landscape Δx units resulting in the situation depicted in Figure 6. Here, the area marked D is updated in the texture. Note that equation 3 ensures that the proper area of the texture is always

used upon lookup. Again we move the landscape Δx units resulting in the situation depicted in Figure 7. Here, the area marked D' is updated in the texture.

2.2 Animated water

Water can be modeled in many ways ranging from simple 2D water surfaces to complex, full 3D water. Our approach lies in between. We use a displacement map generated with sine waves combined with bump mapping to achieve both surface displacement and water-like effects.

Water is stored in GPU memory as a grid of vertices $V_{i,j}$.

$$V_{i,j} = (x, y, z) \tag{6}$$

where z is constant, i.e. $z = c$ for some real number c. Water can be raised or lowered by changing z. Vertices are then displaced in a vertex shader in the samme manner as the streaming landscape described in section 2.1.1. We define a function for extracting the (x, y) part of $V_{i,j}$.

Definition 2.3. `xycmp`$(V_{i,j})$ returns the (x, y) part of a vertex $V_{i,j}$.

The method used for surface displacement is heavily inspired by [CgManual, 2005]. Here a number of sine and cosine waves with different frequency and amplitude and slightly different directions are summed. This gives the final waves a random look. The disadvantage of this approach is that the resolusion of the waves is only as good as the resolusion of the grid used. Thus very small waves cannot be simulated efficiently using this method. To simulate small waves on the water surface and to achieve water-like lightning effects we add bump mapping. This is shown in Figure 9 and 10.

An equation for computing approximations of wave normals is described in [CgManual, 2005].

$$\bar{N} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} + \sum_{waves} \begin{pmatrix} -\cos(\texttt{xycmp}(V_{i,j}) \cdot W) h f w_x \\ -\cos(\texttt{xycmp}(V_{i,j}) \cdot W) h f w_y \\ 0 \end{pmatrix} \tag{7}$$

where W is the projection of the wave direction on the xy-plane, h is the wave height and f is the frequency of the wave. w_x and w_y are the x and y components of W respectively. We sum over all waves contributing to the normal. The resulting vector $\bar{N}$ needs to be normalized before use.

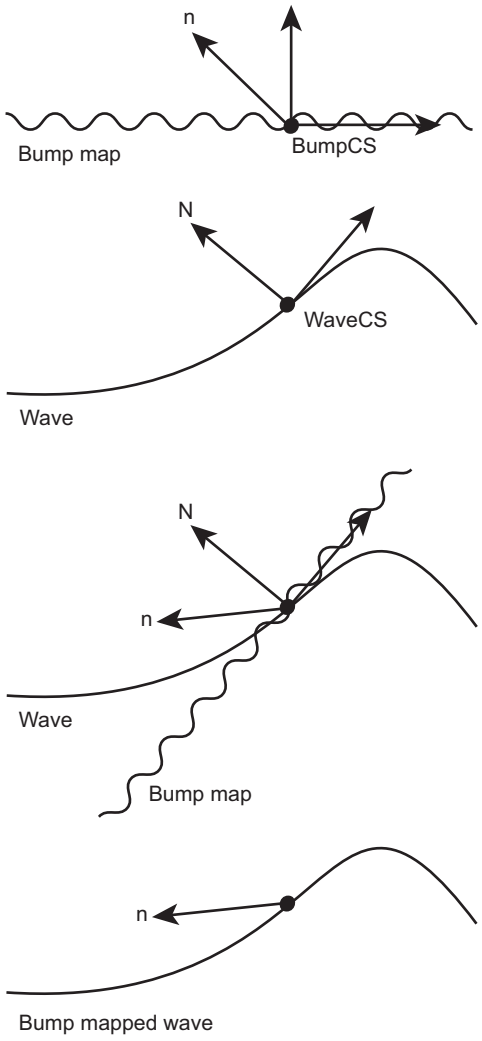


Figure 8: Adding bump mapping to waves. Initially, the bump map contains normal vectors relative its own local coordinate system. The normals are then transformed to the correct wave point coordinate system.

2.3 Adding bump mapping

When using the bump map in a shader the normals $\bar{n}$ are encoded, by the Photoshop plugin, as 4 channel RGBA colors, C. More precisely, they are encoded as floating point values ranging from 0 to 1. The normal

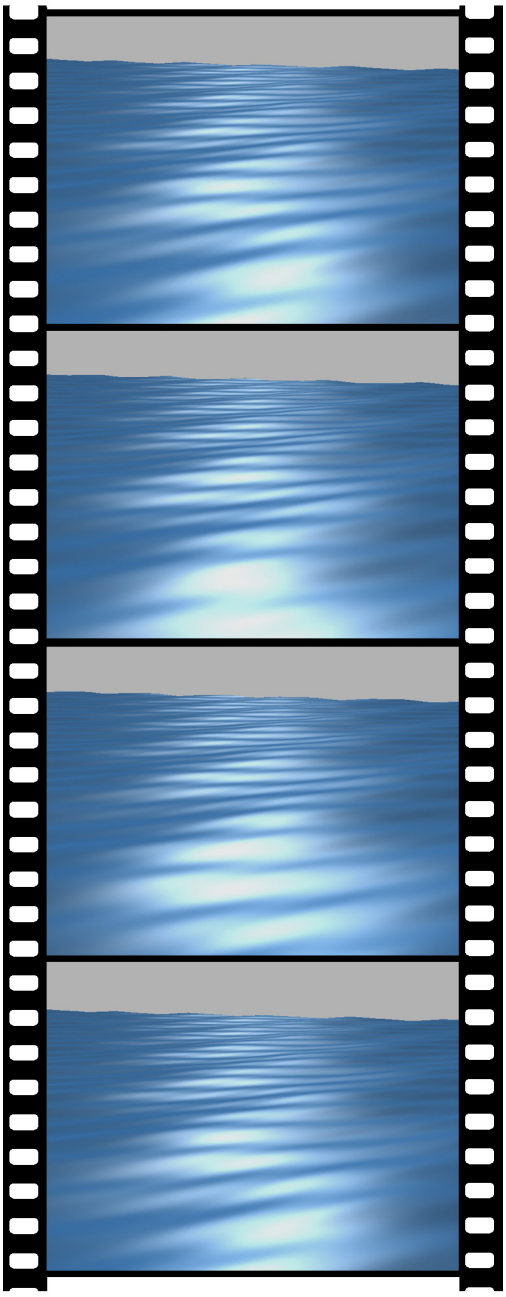


Figure 9: Water with no bump mapping.

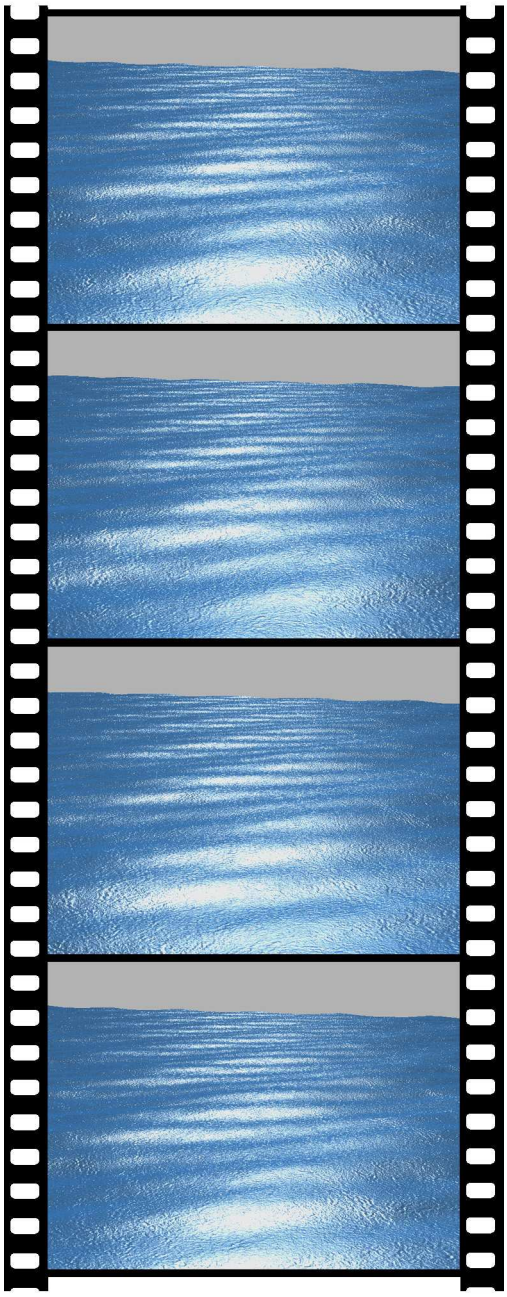


Figure 10: Water with bump mapping.

values thus needs to be transformed back to the range -1 to 1.

$$\bar{n} = 2C - 1 \tag{8}$$

To add bump mapping to the water surface we need a transformation that aligns the bump map coordinate system with the coordinate system of each point on the water surface. This is illustrated in figure 8.

Definition 2.4. Each point p on the water surface has is own coordinate system with origo in p and binormals $\bar{B}'$ and $\bar{B}''$ and normal $\bar{N} = \bar{B}' \times \bar{B}''$ as axes.

A homogenous rotation matrix can be created. Remembering that for orthogonal matrices columns are images of the base vectors [Foley et al., 1997, chapter 5] we use binormals $\bar{B}'$ and $\bar{B}''$ and the normal $\bar{N}$ at each point on the wave surface to create the proper rotation.

An approximation of the binormal is the y-axis base vector.

$$\bar{B}' = \bar{N} \times \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \times \bar{N} \tag{9}$$

The cross product of B' and the normal produces the other (approximated) binormal.

$$\bar{B}'' = \bar{N} \times \bar{B}' \tag{10}$$

Finally, the rotation matrix can be created.

$$R = \begin{pmatrix} \bar{B}''_x & \bar{B}'_x & \bar{N}_x & 0 \\ \bar{B}''_y & \bar{B}'_y & \bar{N}_y & 0 \\ \bar{B}''_z & \bar{B}'_z & \bar{N}_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{11}$$

where B' and B'' are binormals and N is the normal. Thus, we can rotate the normals.

$$\bar{n'} = R\bar{n} \tag{12}$$

2.4 Reflections and landscape sensitive waves

Although we have not implemented reflections, a simple and fast approach suitable for planar surfaces, like water, is presented in [Foley et al., 1997]. Here the camera is reflected about the surface and an inverted image is rendered from the reflected viewpoint. The rendering can be done to a texture which can then be merged with the image rendered from the original viewpoint.

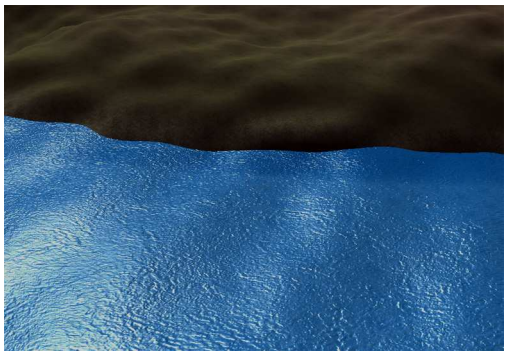


Figure 11: Waves near coastal areas.

Currently waves are similar all over the water surface. This is not very realistic, see Figure 11, since waves tend to act different according to whether they appear on open water or near a coast. Here we could use the normal and height map texture already in GPU memory used for the streaming landscape.

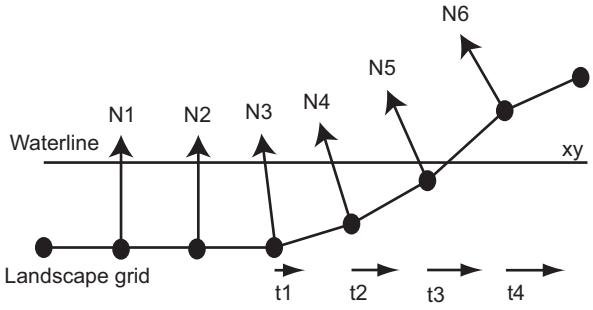


Figure 12: Turning waves.

The projection of the landscape normals N_1 to N_6 on the xy-plane are used to find the coast directional vectors t_1 to t_4, as seen in Figure 12. These are scaled according to the height diffrence between the landscape and water so that small, deep ocean bumps does not change the direction of the waves. t_1 to t_4 can then be used to alter the direction of the waves.

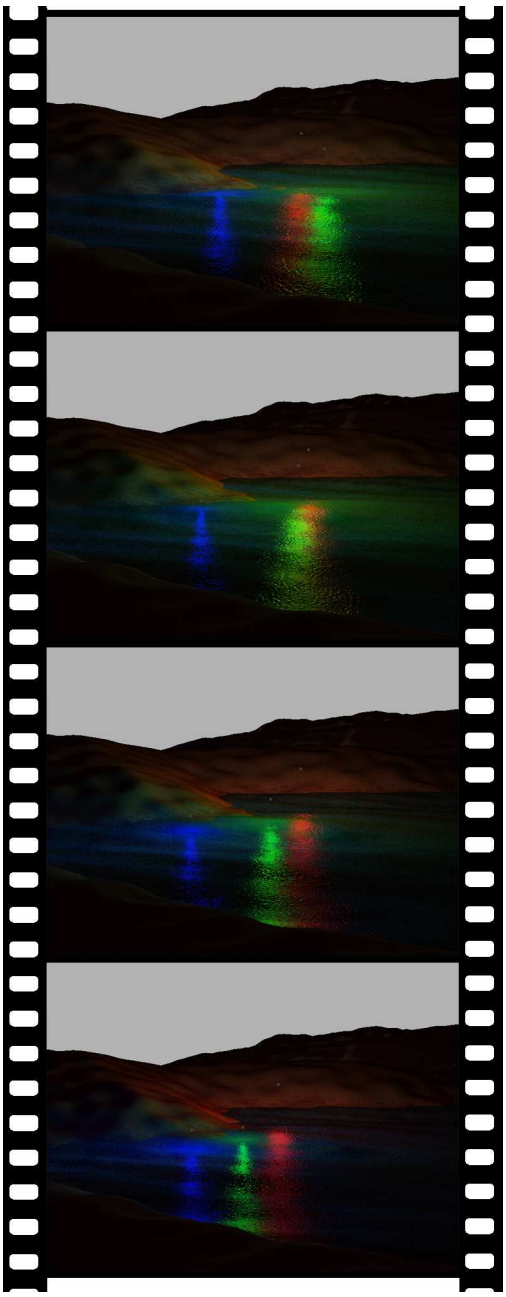


Figure 13: 3 animated point light sources: A red, a blue and a green.

2.5 Dynamic lightning

A lightning model that supports an unlimited number of both point and directional light sources is the Phong illumination model described in [Foley et al., 1997]. We have used a modified version of this model: We have removed the reflection coefficient k_λ since we want to upload as little data to the GPU as possible, and since k_λ can easily be combined with an object's material color for both diffuse and specular light.

$$O_{\lambda,k} = k_\lambda O_\lambda \tag{13}$$

This gives us the complete equation for computing lightning.

$$\begin{aligned} I = O_{a\lambda,k} + \sum_{lights} f_{att}(O_{d\lambda,k} L_d(\bar{n} \cdot \bar{l}) + \\ O_{s\lambda,k} L_s(\bar{n} \cdot \bar{h})^n) \end{aligned} \tag{14}$$

where I is the resulting illumination in the vertex or fragment, and $O_{a\lambda,k}$, $O_{d\lambda,k}$ and $O_{s\lambda,k}$ is the object's ambient, diffuse and specular material color respectively. L_d and L_s is the light source's diffuse and specular color. $\bar{n}$ is the normal vector in the vertex or fragment. $\bar{l}$ is the light source direction vector. $\bar{h}$ is the halfway vector, which is halfway between the direction of the light source and the eye. n is the specular reflection exponent. In equation 14, all vectors are assumed to be normalized.

For point light sources, we need to calculate a light vector from the vertex or fragment to the light source $\bar{l} = p_l - p_o$; where $\bar{l}$ is the light vector, p_l is the position of the light source, and p_o is the position of the vertex or fragment, that we currently are illuminating. All lightning computations are done in the WCS.

2.6 Lightning in vertex and pixel shaders

We light the landscape in a vertex shader which means that landscape is illuminated with Gouraud shading. Water is illuminated in a fragment shader so here we use Phong shading. One problem with multiple light sources, when using shaders, is that when a shader program is compiled it needs to know the size of the array containing the lights. This means that a finite maximum number of lights has to be known before compiling the shader program. To set the array size a shared array parameter with a size equal to the number of lights can be used. This is shown in algorithm 1 in appendix C.

When uploading the shader program to the GPU, we need to disable automatic compilation and connect the shared parameter from algorithm 1 to the program. This is shown in algorithm 2.

Now, array elements can be modified like all other parameters. Each light source l' is contained in a 4x4 matrix such that updating all light sources in the array can be done in a single function call.

$$l' = \begin{pmatrix} \bar{l}_x & \bar{l}_y & \bar{l}_z & l_w \\ L_{d,x} & L_{d,y} & L_{d,z} & - \\ L_{s,x} & L_{s,y} & L_{s,z} & - \\ c_1 & c_2 & c_3 & - \end{pmatrix} \quad (15)$$

where $\bar{l}$ is the light source direction vector, L_d is the light source's diffuse color, and L_s is the light source's specular color. l_w is used to determine if the light source is a point or direction; $l_w = 1$ means point, $l_w = 0$ means direction. c_1, c_2, c_3 are user controlled attenuation constants that are used to compute f_{att} as described in [Foley et al., 1997, equation 16.8].

$$f_{att} = min\left(\frac{1}{c_1 + c_2 d_L + c_3 d_L^2}, 1\right) \quad (16)$$

where d_L is the distance the light travels from the point light source to the surface. Entries in equation 15 marked with $''-''$ are unused. The result of using multiple point light sources is shown in Figure 13.

3 Analyzing performance

We have benchmarked the existing and the extended game engines, and compared the results. Benchmarking was done with landscape alone, with water, and with a single and multiple light sources. The camera was moving at all times and was positioned with a 90^o camera angle relative to the landscape. Further, frustum culling and occlusion queries were disabled in both engines. We used an AMD Barton 2500+ with 1GB ram and a GeForce 6600GT with 128MB ram. Results for the existing engine is shown in Table 1 and appendix A, while results for the extended engine is shown in Table 2, 3, 4 and in appendix B. It should be noted that comparisons are only approximate since the different underlying implementations make it impossible to make exact comparisons.

The measure for whether the engines perform realtime, is if rendering is possible at an average of minimum $30fps$.

The existing engine performs realtime, but has significant framerate drops. This is marked on the graphs in appendix A. The framerate drops are due to block loading. After each drop, we momentarily experience an increase in the framerate. This is because that while the CPU computes geometry for the new blocks, the GPU finishes all its current work, and thus waits for the CPU. When the CPU then feeds geometry to the GPU again, it can be processed fast since the pipeline is empty.

Size	*Low FPS*	*Avg. FPS*	*High FPS*
65×65	53	61	68
117×117	40	61	80
273×273	7	30	41

Table 1: Landscape. 1 directional light source.

The average performance of the extended engine is practically equal to that of the existing, and hence it performes realtime. With multiple light sources and water enabled, however, only landscape sizes up to 64×64 can be used for real time purposes. The old engine does not support water and/or multiple directional and point light sources so these extensions can not be compared.

Size	*Low FPS*	*Avg. FPS*	*High FPS*
64×64	59	61	62
128×128	59	61	64
256×256	29	30	31

Table 2: Landscape. 1 directional light source.

Size	*Low FPS*	*Avg. FPS*	*High FPS*
64×64	59	61	63
128×128	30	30	31
256×256	12	12	12

Table 3: Landscape. Water. 1 directional light source.

Size	*Low FPS*	*Avg. FPS*	*High FPS*
64×64	59	61	63
128×128	14	15	16
256×256	5	5	5

Table 4: Landscape. Water. 3 point and 1 directional light sources.

The bottleneck in the extended engine is the texture lookups in the vertex program; vertex texture lookup are much more expensive than texture lookups in fragment programs [GPUGuide, 2005] and [Gerasimov et al., 2005]. However the next generation of GPU will definitely have better vertex texture performance so it is expected that our method will seem even more attractive in the future. In [Hoppe et al., 2004] they were able to render terrain the size of North America using vertex texture

lookups and by using level of detail (LOD). An additional render pass is needed, though, to get a texture suitable for landscape rendering. Currently, our method does not have any LOD. On the other hand we do not use a render pass to render a texture. By extending our method and still using only one render pass, performance could be improved significantly. Our method could be improved further by using frustum culling and occlusion queries.

4 Discussion

We have presented 3 extensions to the existing Game Engine used in the Game Animation course at DIKU. The extensions are:

- Streaming landscape.
- Animated water.
- Dynamic directional and point light sources.

Streaming landscape is implemented by storing a height- and normal map in a floating point texture in GPU memory, and by always updating the smallest possible area of this texture. The result is a solid method for rendering infinite landscapes with constant framerate at all times. Our method can be used for simulation and role playing games where a lot of outdoor terrain geometry is needed. Games with indoor environments do, on the other hand, not benefit from our method. On current generation GPU's texture lookups from vertex programs are expensive, and thus our method is not yet suitable for games where the GPU has to work on more than just landscape.

Water is implemented as the sum of sine function combined with bump mapping. The result is a cheap approximation to realistic water, but does not in any way resemble any of the physical properties of real water. Our model can easily be extended with reflections, refraction, caustics etc.

Dynamic point and directional light sources are created and maintained by the CPU and then transfered to the GPU which performs the light computations. Our method support multiple light sources by using arrays of matrices in shader programs.

References

[Erleben et al., 2005] Erleben, Kenny et al. *Physics-Based Animation.* Charles River Media 2005.

[Foley et al., 1997] Foley, James D. et al. *Computer Graphics: Principles ans Practice - Second Edition in C.* Addison-Wesley 1997.

[Gerasimov et al., 2005] Gerasimov, Philipp et al. *Shader Model 3.0 Using Vertex Textures (Whitepaper).* NVIDIA corporation 2005.

[Kilgard, 2001] Kilgard, Mark. *The OpenGL Utility Toolkit.* `http://www.xmission.com/ nate/glut.html`

[GPUGuide, 2005] NVIDIA corporation. *NVIDIA GPU Programming Guide version 2.4.0.* NVIDIA corporation 2005.

[Hoppe et al., 2004] Hoppe, Hugues et al. *Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids.* ACM Transactions on Graphics (Proceedings of SIGGRAPH 2004) 23(3) 2004.

[CgManual, 2005] NVIDIA corporation. *Cg User's Manual.* NVIDIA corporation 2005.

[OpenTissue, 2006] Opensource Project, Physical based Animation and Surgery Simulation. `http://www.opentissue.org`

[Pharr et al., 2005] Pharr, Matt et al. *GPU Gems 2: Programming Techniques for High-performance Graphics and General Purpose Computation.* Addison-Wesley 2005.

[Perlin, 1985] Perlin, Ken. *An image synthesizer.* SIGGRAPH: Proceedings of the 12th annual conference on Computer graphics and interactive techniques 1985.

[Perlin, 2002] Perlin, Ken. *Improving noise.* SIGGRAPH: Proceedings of the 29th annual conference on Computer graphics and interactive techniques 2002.

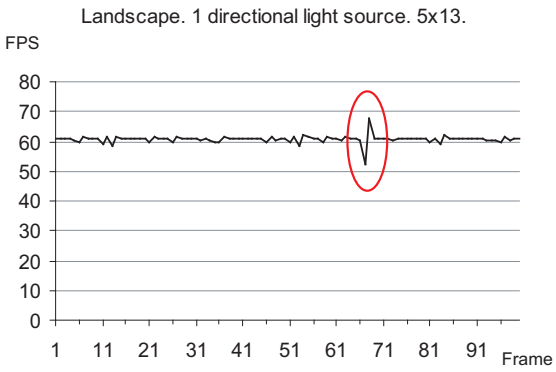
[Shreiner et al., 2005] Shreiner, Dave et al. *OpenGL Programming Guide.* Addison-Wesley Professional 2005.

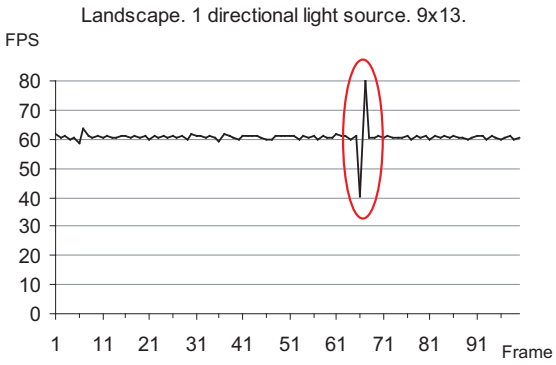
[Stroustrup, 1997] Stroustrup, Bjarne. *The C++ Programming Language.* Addison-Wesley 1997.

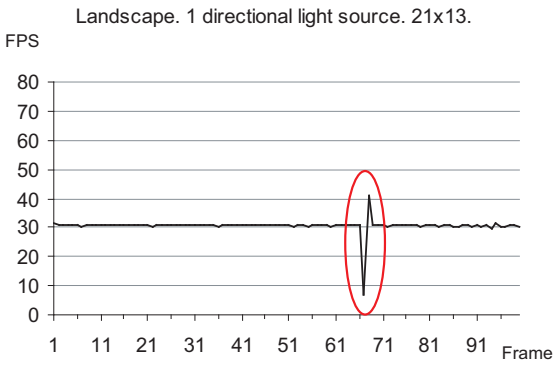
[TextureTools, 2005] NVIDIA corporation. *NVIDIA Texture Tools.* `http://developer.nvidia.com/object/ nv_texture_tools.html`

[USPatent, Terrain] Hoppe, Hugues et al. *US Patent Application 20050253843: Terrain rendering using nested regular grids.* `http://www.uspto.gov/patft/index.html`

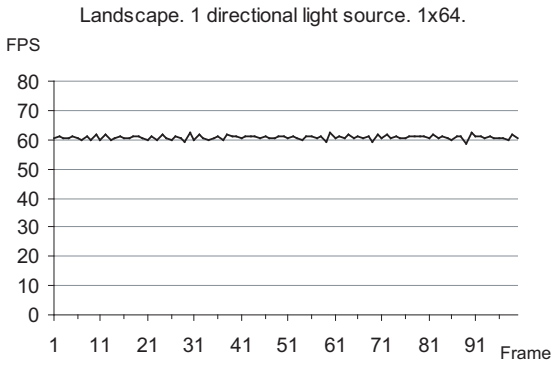
A Existing engine performance

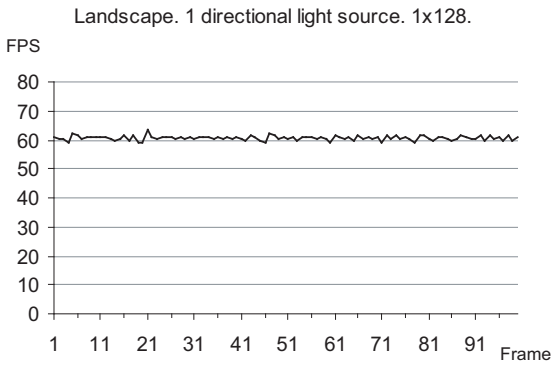


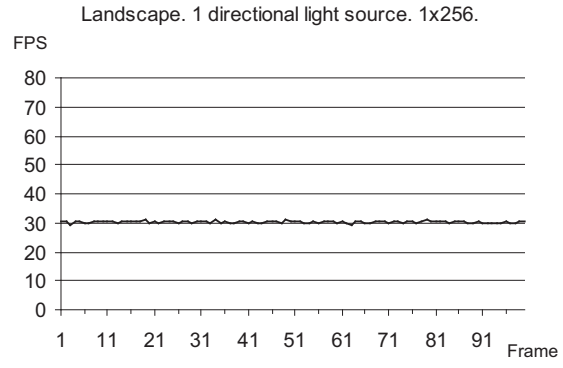




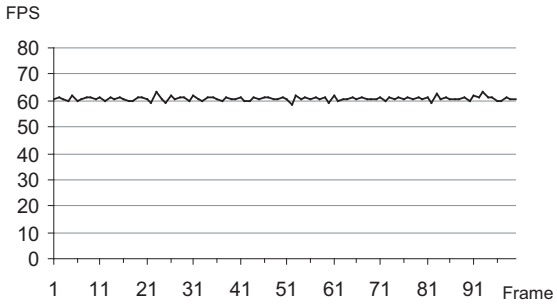
B Extended engine performance



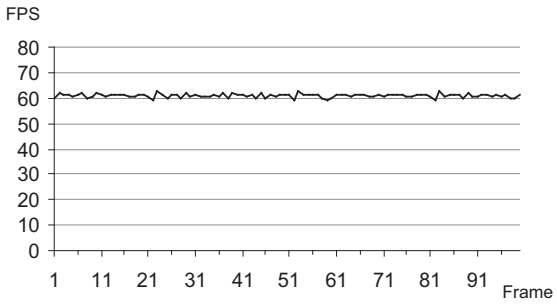




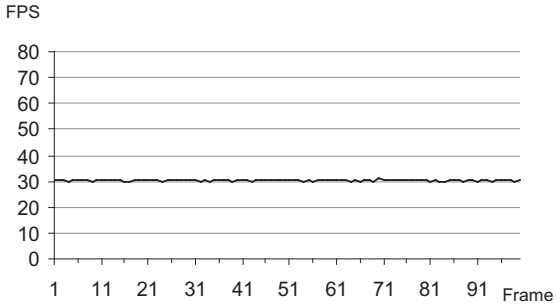
Landscape. Water. 1 directional light source. 1x64.



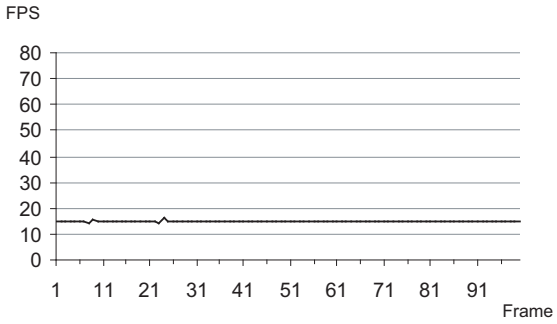
Landscape. Water. 3 point and 1 directional light sources. 1x64.



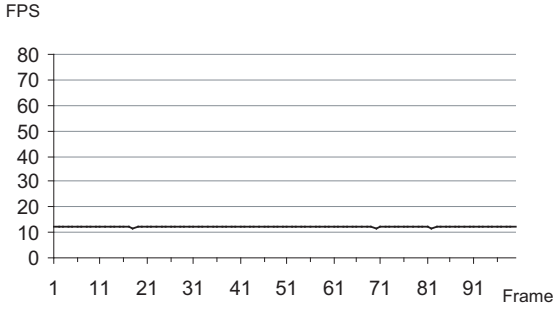
Landscape. Water. 1 directional light source. 1x128.



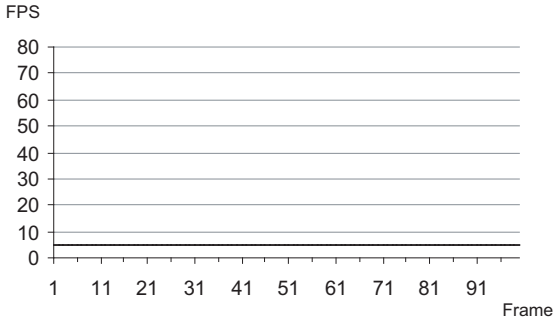
Landscape. Water. 3 point and 1 directional light sources. 1x128.



Landscape. Water. 1 directional light source. 1x256.



Landscape. Water. 3 point and 1 directional light sources. 1x256.



C Pseudocode for setting up dynamic lightning

```
    Function : InitSharedParams()
1.1 size = maximum number of light sources;
1.2 sharedparam = cgCreateParameterArray( ..., size );
```

Algorithm 1: Setting the size of the array containing light sources.

```
    Function : InitProgram()
2.1 cgSetAutoCompile( ..., CG_COMPILE_MANUAL );
2.2 program = cgCreateProgramFromFile( ... );
2.3 param = cgGetNamedParameter( program, "Lights" );
2.4 cgConnectParameter( sharedparam, param );
2.5 cgCompileProgram( program );
2.6 cgGLLoadProgram( program );
```

Algorithm 2: Creates a program with an unsized uniform array parameter, and sets the array size to the same size as a shared parameter.