Technical Report

Extrapolate and Conquer

TSBK03 HT 2013

Version 0.1

2013–12–23

Extrapolate and Conquer

Teknik för avancerade datorspel, HT 2013
Department of Electrical Engineering (ISY), Linköping University

Participants

Name	Tag	Responsibilities	Phone	E-mail
Gustav Häger	GH		070–649 03 97	gusha124@student.liu.se
Alexander Sjöholm	AS		076–225 11 74	alesj050@student.liu.se
Mattias Tiger	MT		073–695 71 53	matti166@student.liu.se

Examiner: Ingemar Ragnemalm, ingis@isy.liu.se

Contents

List of Figures

List of Tables

1 Introduction

The aim of this project was to develop a computer graphics application combining several state-of-the-art techniques into a beautiful world.

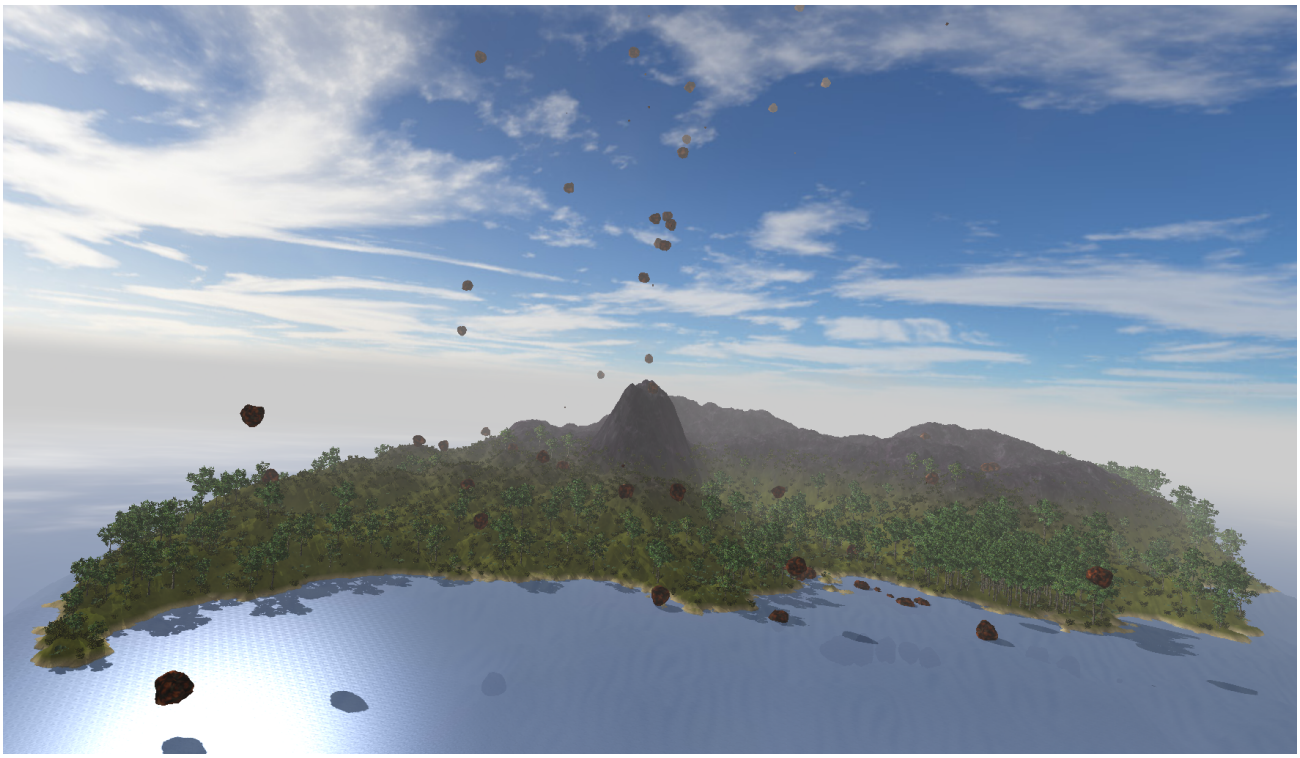


Figure 1.1: A beautiful world.

2 System Core

We built and uses an Entity System as an underlying game engine framework. The idea behind an entity system is that objects should be treated as pure aggregations of data containers, with game logic being separated from objects all together. Instead of having deep class hierarchies and chained method calls, logic for managing specific components is batched on all such components in the system. This provides some advantages over other approaches such that the architecture becomes more flexible and expendable. It is clearer how to add functionality and especially where. Another advantage from the batching is the possibility of much higher performance (maximizing caching and minimizing cache misses) as well as simplifying parallel processing.

2.1 Entity System

An Entity System consists of three main parts: Entities, Components and Systems. An Entity is simply a label or identifier of an object. A Component is a pure data containers, and each entity has a collection of none to several different components. A System consist of logic for working with primarily one, but sometimes several, components. So, an object is an entity label and a collection of components that belong to it. The object is updated by different systems performing tasks on the components. An example used in this project is seen in figure 2.1.

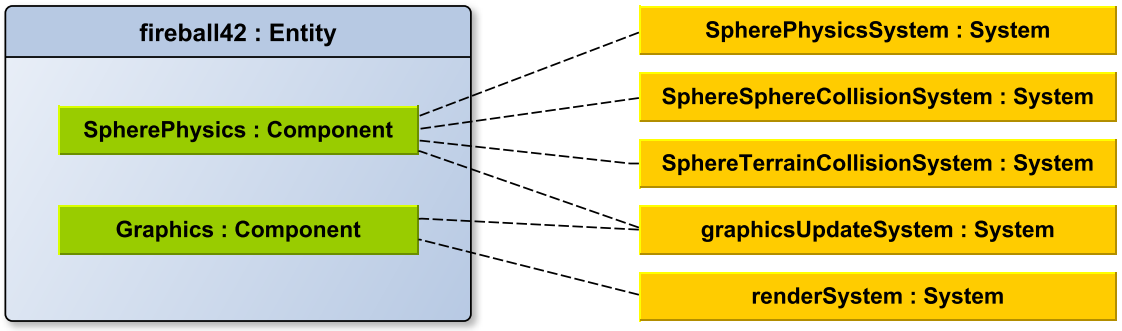


Figure 2.1: Caption

(TODO: reformulera och skriv nedan mer flytande och omarbetat)
* The idea behind: Systems perform on ALL components of a certain type, which is fetched from the Entity-Manager.
* Our system allows for components to require other components. This means that systems which work on multiple components can be proven to always work by fetching the component which require the others. The require check is performed at compile time and code is generated to handle the specific requirement-tree specified. The program writes part of itself to be maximum efficient and robust, by specifications by the developer in the source code.
* Easy to use (a natural work flow of what goes where and how to solve problems. minimal overhead to use the system), easy to maintain, easy to extend, extremely efficient, trivial to parallelize calculations, verifies consistency at compile time...

```
1  Entity & entity = entityManager.createEntity();
2
3  entity.add<SpherePhysics>();
4  entity.add<Graphics>();
5
6  SpherePhysics & physics = entity->get<SpherePhysics>();
7  physics.position = QVector3D(0.0, 0.5, 1.0);
8  physics.friction = 0.5;
9  ...
10
11 Graphics & graphics = entity->get<Graphics>();
12 graphics.object = new Object(resourceManager->getModel("stone1"),
13                               resourceManager->getShader("phongTex"),
14                               resourceManager->getTexture("lava1"));
```

```
entity.has<Component>();
entity.add<Component>();
entity.remove<Component>();
entity.get<Component>();
```

Code generation...

```
// If additional Components are required, add these aswell (recursively)
  meta::FOR_EACH< typename Component::REQUIRED_COMPONENTS, // List (items) to iterate over

                  ADD_COMPONENT,                           // Template to apply
                                                           // on each item.
                  std::tuple<Component, Components...>,    // Additional template
                                                           // parameters to above template.
                  std::tuple<Entity<Components...>&>       // Argument types to the above
                                                           // templatet's execute-function.
                >::execute(entity);
```

2.2 Rendering

The rendering system is outside the entity system mainly because it is the largest part, and it was not known in the start of the project what information it needed exactly, so rather than potentially locking ourselves to an unusable architecture the renderer was placed beside the entity system rather than as a part of it.

The rendering is done in two major passes, the first calculates the shadows and second draws the world using the shadow information calculated in the first pass.

In order to correctly draw the growth some form of transparency was needed as the foilage used partly transparent textured polygons. As the textures where either completely transparent or completely opaque a simple alpha test was sufficient rather than sorting the trees. However the mipmaps calculated by opengl for the foilage where completely broken by the transparency, so these are disabled for the trees.

In order to be able to draw a sufficient number of trees and bushes a simple form of instancing is used. Each type of tree and bush exists in a separate list, that contains the model and texture for that particular object and an array of model matrices to be uploaded to a shader as an array of uniforms.

The rocks on the other hand are drawn individually, using only a reference to a shared model objects, and individual matrices, as well as a specified shading program, thus the rocks could be drawn using individual shaders if it was needed.

The water is always drawn last, as it is the only object in the world that is sempitransparent, this does have the consequence of making the trees invisible from under the water though.

In general this demonstrates the need to for a rendering pipline that does not connect a particular shader to a particular object, but rather connects objects to different shaders, so that the shader program can be selected first, and then all the objects using a particular shader drawn.

Due to a rather lousy bush model with a great deal of z-fighting in itself the depth mask is disabled while drawing bushes, as this is considerably faster than finding and creating new models.

2.3 Physics

The physics used in this game is divided into three systems, all three processing all *SpherePhysics* components in the Entity System. The physics is limited to 3D sphere physics as well as the interaction between spheres and a terrain mesh.

2.3.1 SpherePhysics

SpherePhysics is a component dedicated to physics data, its content is seen in the code below. The physics is capable of running backwards as well as forwards.

```
struct SpherePhysics : public Component<> {
    const std::string getName() override { return "SpherePhysics"; }

    float mass;                     // m            (Positive)
    float elasticity;               // epsilon      (Between 0 and 1)
    float friction;                 // friction     (Between 0 and 1)
    float momentOfInertia;          // For a sphere: 6/12*m*radius^2
    float gravitationalConstant;    // g            (Positive. in Sweden at sea level: 9.82)

    QVector3D force;                // F = ... external events ...
    QVector3D linearMomentum;       // P = Integral( F, dt );
    QVector3D velocity;             // v = P / m;
    QVector3D position;             // x = Integral( v, dt );

    QVector3D torque;               // T = ... external events ...
    QVector3D angularMomentum;      // L = Integral( T, dt );
    QVector3D angularVelocity;      // w = L * Inverse(I)
    QQuaternion angularVelocity2;   // w = L * Inverse(I)
    QQuaternion rotation;           // r = Integral( w, dt );

    float radius;                   // Radius of the sphere
    QVector3D collisionVector;      // The sum of the vectors from all collisions

};
```

2.3.2 SpherePhysicsSystem

The sphere physics are based on the physics lab, with a generalization to 3D and using quaternions as the representation for rotation. The system require the current time step *dt*. which is used as the integration step in the Euler-forward integration used. This was sufficient for our needs, with no more advanced integration schemes necessary.

The spheres are affected by external forces and torques, which then updates the rest of the physical states. At the end, any forces and torques has been handled and are set to zero, and finally the ever present gravitational force is added.

2.3.3 SphereSphereCollisionSystem

The sphere-sphere collision handling are based on the physics lab, with a generalization to 3D. The collision is handled by reversed impulse. Collisions give rise to torque, making he collisions seem very realistic.

2.3.4 SphereTerrainCollisionSystem

The sphere-sphere collision handling uses the terrains height and normal at the bottom of the sphere. It is otherwise similar to the SpherePshereCollisionSystem apart from that the terrain is considered to have infinite mass. A normal force is applied to the object to minimize noise from the gravity force. A friction force can be used if a more realistic collision is desirable. In our demonstration it isn't used since we had a limited amount of fire balls, so the fewer that got stuck on land the longer the volcano could spew out new fire balls.

3 Graphics

Graphics graphics.

3.1 Generating a World

A world is easily divided into different aspects. There are the sky, the oceans and the land. The land contain different terrain, with different types of ground and vegetation. There is a sun orbiting the world, casting rays of light and in the process creating reflections and indirectly making shadows.

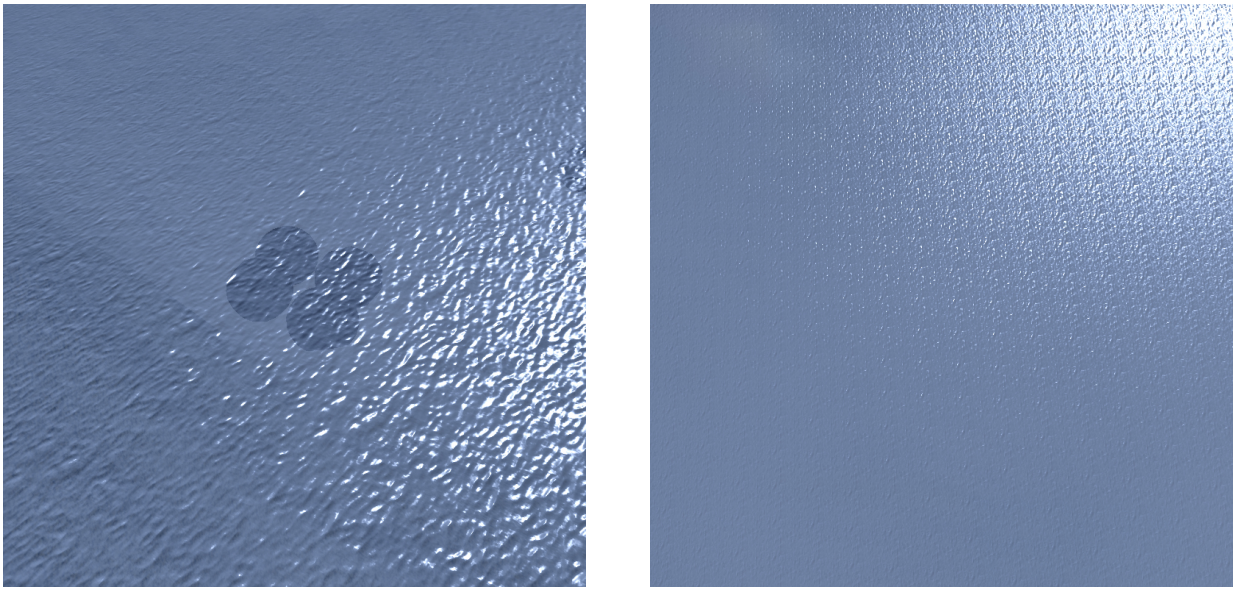
A procedural world is generated by carefully chosen algorithms. Our world is procedurally generated anew in a new unique constellation on every run, or a seed can be provided to generate specific worlds. The terrain is first formed, then the ground texturing and the vegetation, both dependent on properties of the terrain. The shadows depend on the sun and the waves on the terrain as well as on time itself.

3.1.1 Sky

The sky is achieved using a high-resolution texture of a sky mapped to a skybox. The mathematical location of the sun is placed as close as possible to the sun appearing in this texture. This gives shadows and shading a natural feel. The texture has been manually modified at the horizon to fade towards a shadowish gray color. The color is the same as the one objects are distance-fogged with. This makes the sky melt into the ocean in a very nice way.

3.1.2 Ocean

The water is made up of a normal-mapped square and a blue color. The normal map is repeated over the square and is moving in texture space, giving the illusion of a wind. The normal map movement cycles such that when the water normal map has been totally displaced, it starts over again from its original position. This movement makes the water glitter from a far, see figure 3.1.



(a) The water. (b) Glittering water.

Figure 3.1: *Comparison of noise functions*

Waves on the beaches is made by having several sinus waves aggregate horizontally to vary the wave fronts, and by having a sinus wave that control the vertical assent/decent of the waves. It is hard to catch on a screen shot, but an image of the waves in the world is shown in figure 3.2

Figure 3.2: The waves are seen at the beach.

The sky is reflected in the water by using an image which is the projection of the sky onto the ocean as a texture for the water. This texture is moving with the camera in the xz-plane. An example is seen in figure 3.3. It could be improved in the future by taking the height of the camera in consideration as well, or to continuously project the sky onto the water according to the camera position. Currently the camera height do not change the reflection, making it a bit unrealistic if moving up and down.

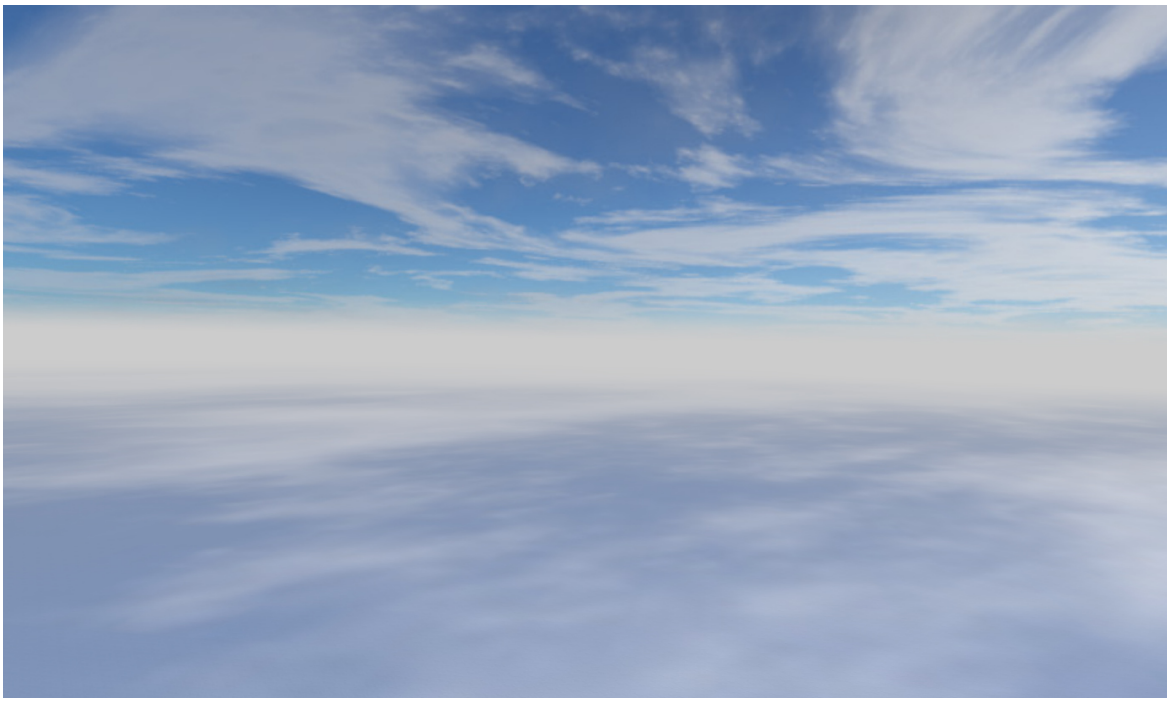
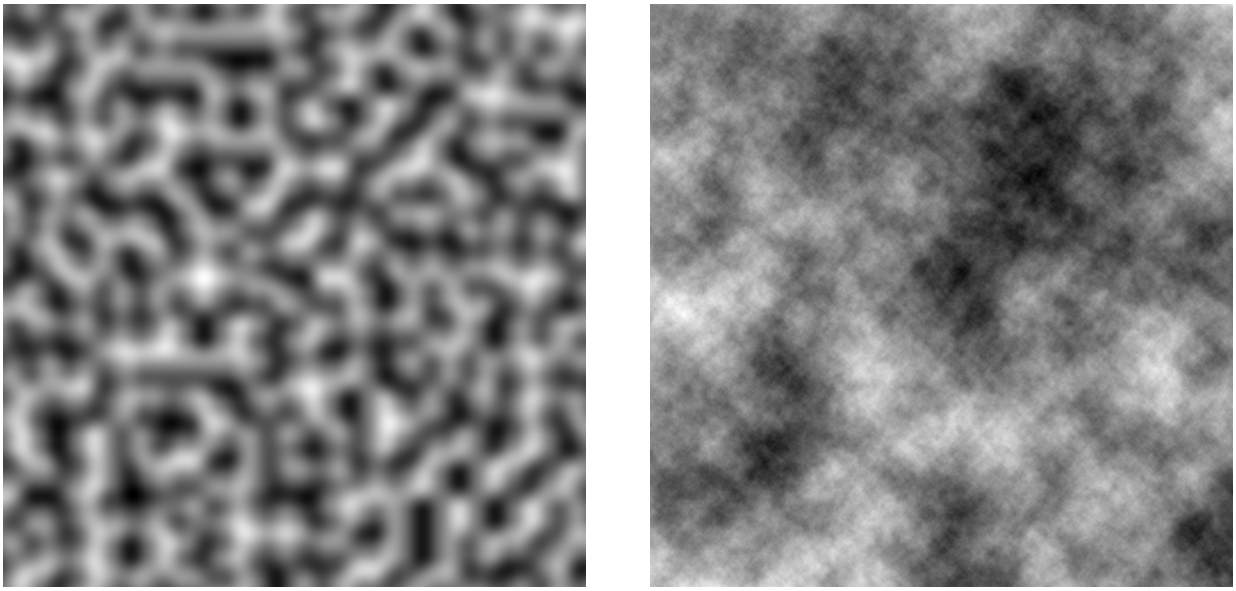


Figure 3.3: The sky reflected in the ocean.

3.1.3 Terrain

The terrain is generated by sampling a noise function and transforming its value into a height for each vertex. The noise in this case originates from a Simplex function. However, to get a realistically looking terrain it is not sufficient to sample this function only once for every vertex.

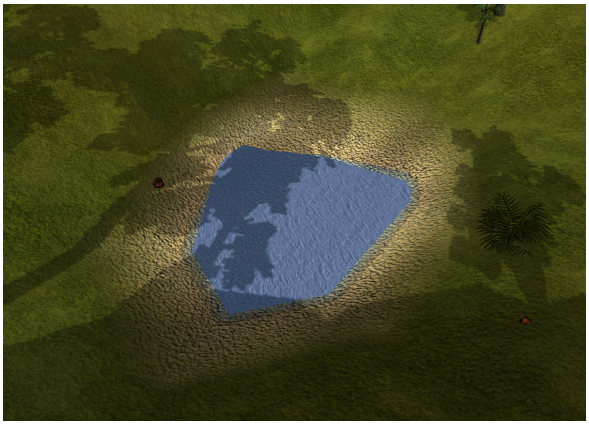
Fractional Brownian Motion is calculated by sampling the Simplex function at different frequencies and calculating a weighted sum over the samples [1]. The result is a nice looking height map. This allows for an arbitrary sample density. A few different sample densities are shown in figure 3.5. A further possible extension would be to use tessellation, which would allow for a terrain with arbitrary fine details.



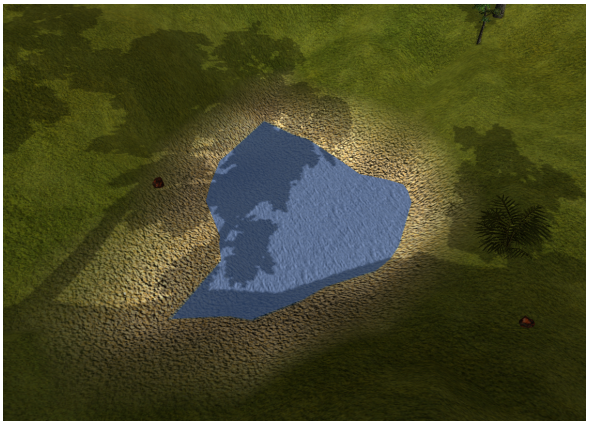
(a) Height map generated from single-octave simplex noise (b) Height map generated with Fractional Brownian Motion

Figure 3.4: *Comparison of noise functions*

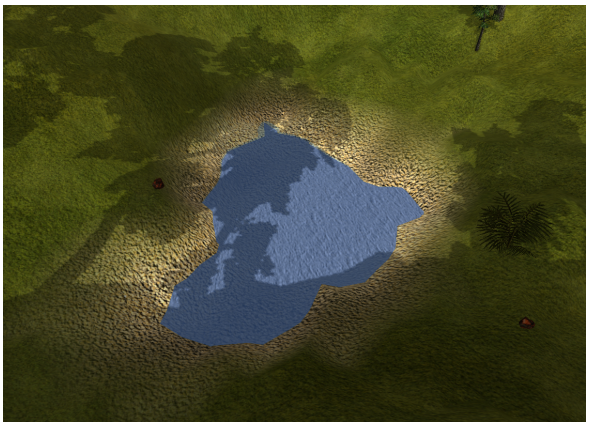
Example of different densities... Arbitrary high density possible...



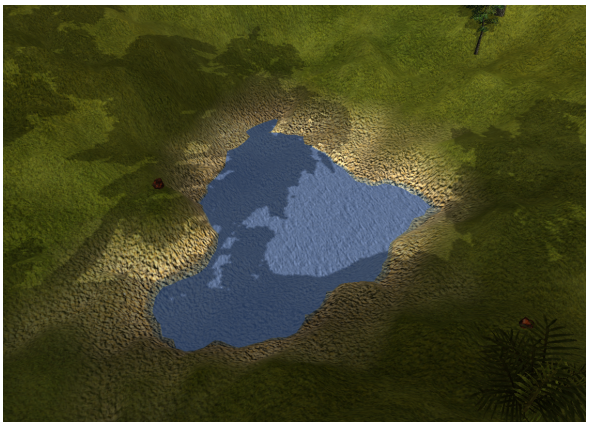
(a) Density 0.5



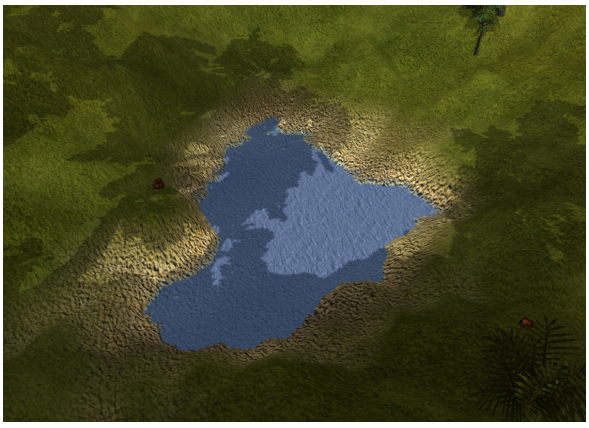
(b) Density 1.0



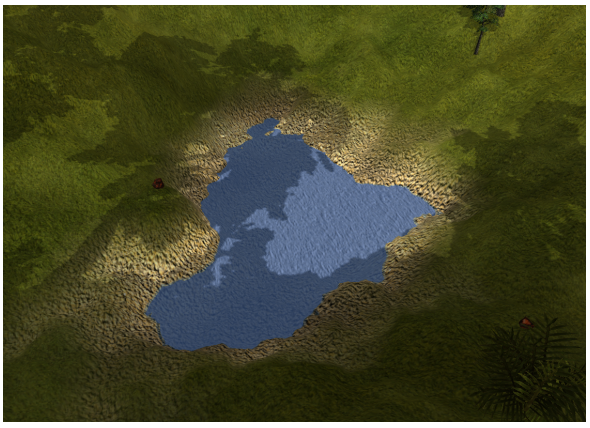
(c) Density 2.0



(d) Density 5.0



(e) Density 10.0

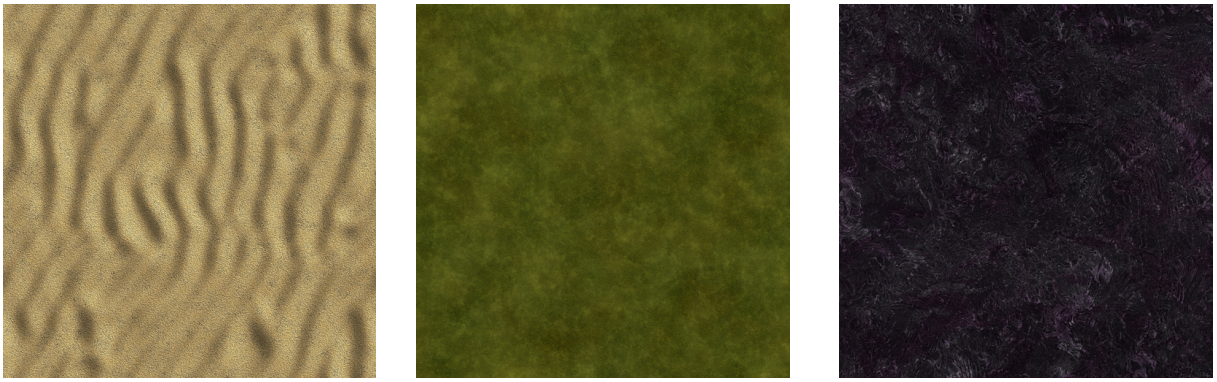


(f) Density 15.0

Figure 3.5: A comparison of different vertex and sample density, from 0.5 to 15 OpenGL length units. Waves are shown in some of the images, as the wave generator currently doesn't distinguish between lakes and oceans.

3.1.4 Ground

The ground is textured using a non-linear multi-texturing approach based on both altitude and terrain slope. Currently only three textures are used: one sandy, one grassy and one rocky, which can be seen in figure 3.6.



(a) The sand/beach texture. (b) The grass/ground texture. (c) The rock/volcano texture.

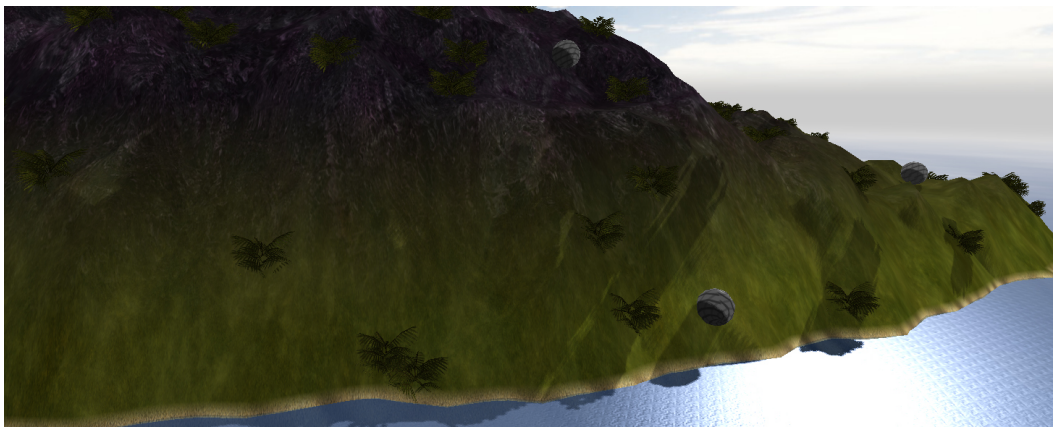
Figure 3.6: *Textures used for the ground.*

The texture blending allows for the mountain to reach almost down to the ocean while still letting the fields and the highland being very grassy. The blending between rock and grass make the landscape more alive and less dull. In figure **??** and **??** a simple blending using linear interpolation between rock and grass is compared to our more advanced method. In our method the interpolation between the different textures vary between being squared, linear and square-root, depending on the slope and altitude.

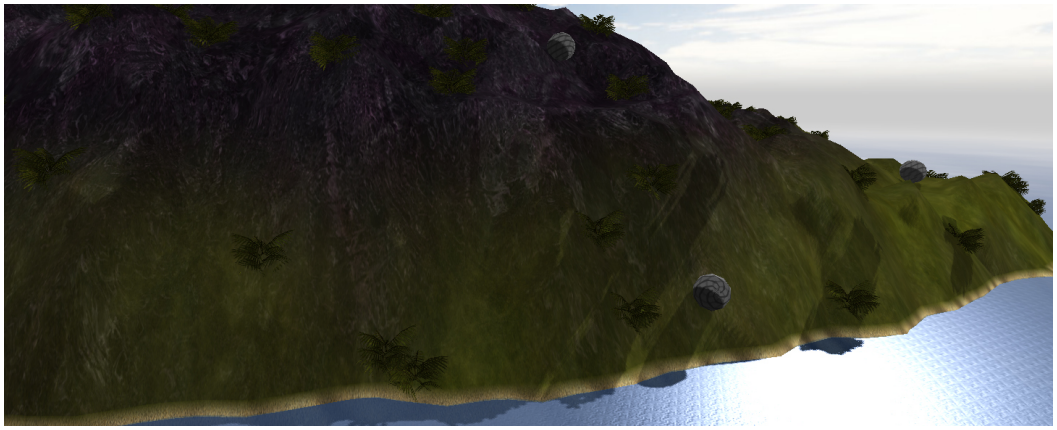
(a) Simple linear interpolation.

(b) Advanced non-linear texture blending.

Figure 3.7: A comparison between a simple interpolation of the textures (a) and our method (b). Notice the varying tone of the grass as well as the deep mountain slope to the right in (b).



(a) Simple linear interpolation.



(b) Advanced non-linear texture blending.

Figure 3.8: A comparison between a simple interpolation of the textures (a) and our method (b). Notice the varying tone of the grass as well as the much greener upper region to the right in (b).

3.1.5 Vegetation

In order for the world to be more interesting than just an island in the middle of the sea, it needs some additional objects aswell, thus plants are placed using two slightly different approaches.

First a base layer of smaller bushes are placed uniformly over the landmass, with a randomized orientation and scaling. Then a few points are designated as forest centers, around these trees are placed using a gaussian distribution, again the trees have some scale and rotation randomly selected.

Placing the objects is done by generating a coordinate, and checking it for suitability, for example nothing can be placed in the water, or to far up on a mountain/voulcano.

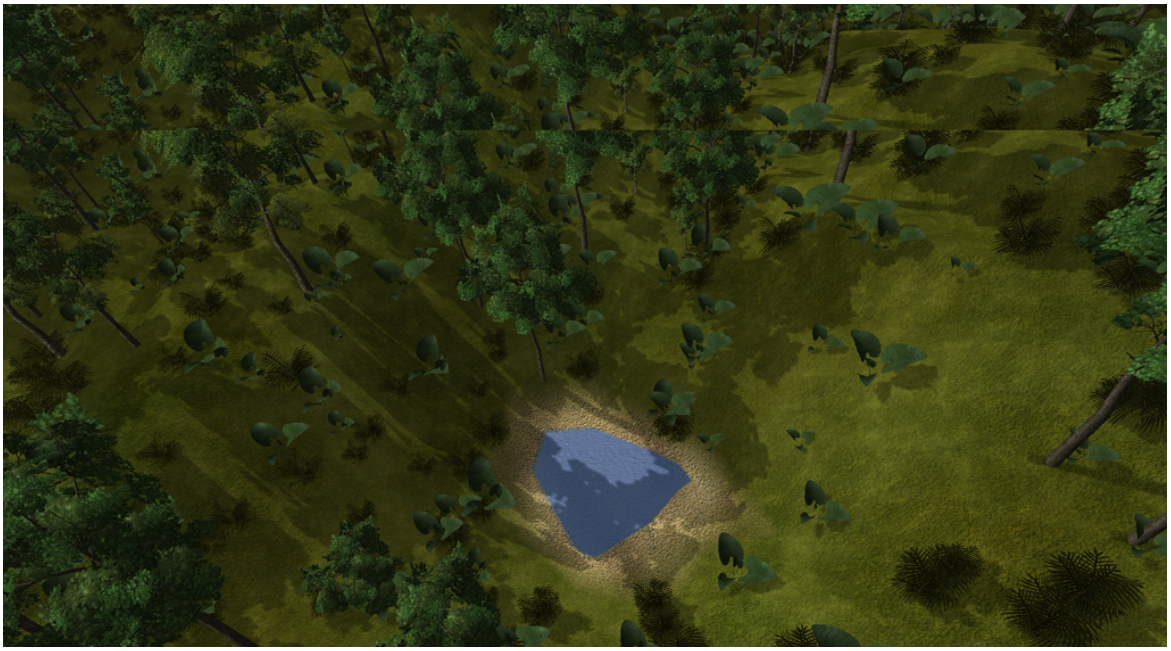


Figure 3.9: A small water hole in the middle of the forest at the late afternoon.

3.1.6 Volcano

The volcano generator changes the terrain on a spot to that of a volcano, such as the one seen in figure 3.10.

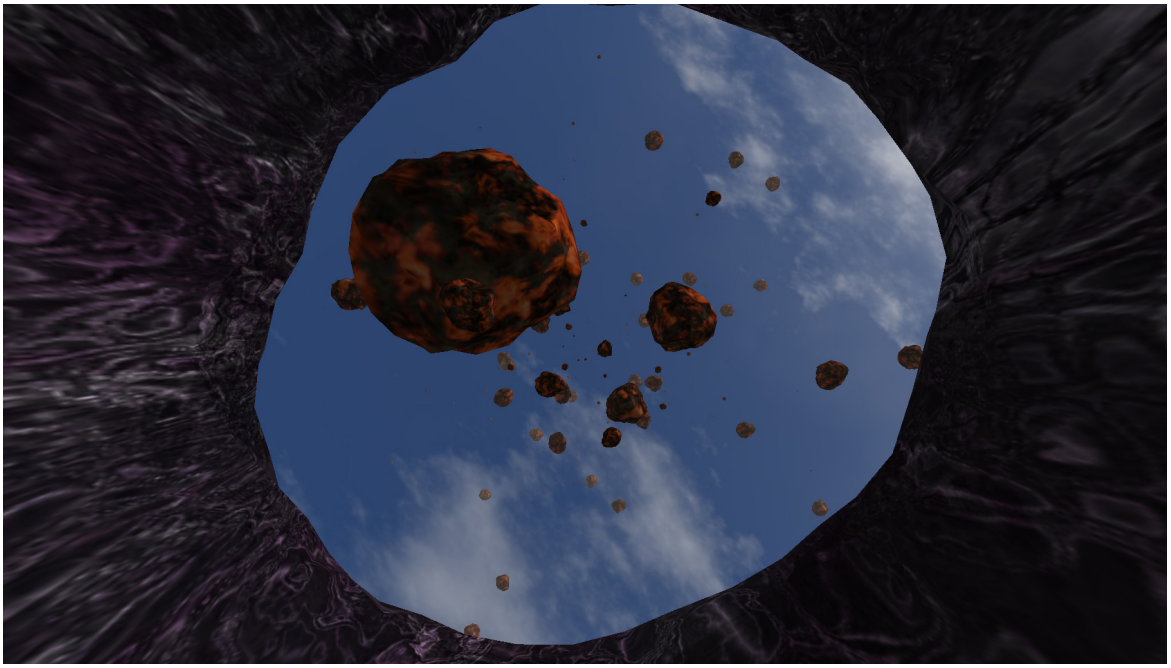
Figure 3.10: A vulcano.

The volcano is made from one addition of a 2D-Gaussian to the terrain, and of a subtraction of a smaller 2D-Gaussian which creates the eruption hole. It erupts and spews out lava stones as a particle effect. Currently

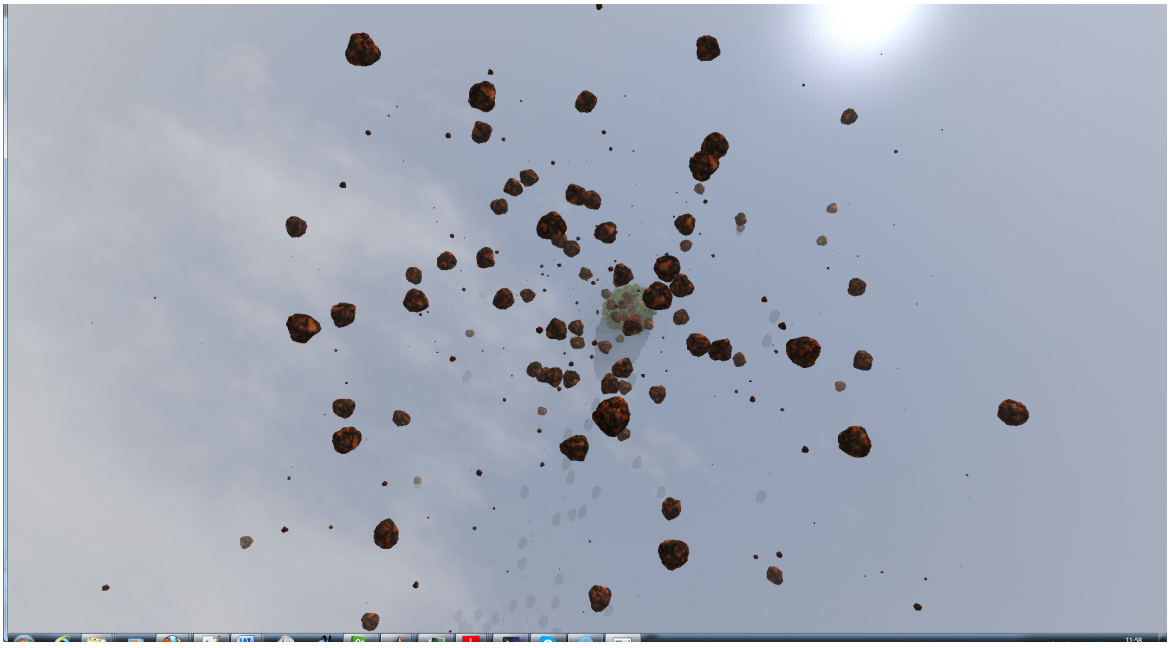
only one volcano is generated on each island, and it is placed on the highest point on the terrain.

Lava stones are made up of 5 different stone models of different sizes. They have full 3D sphere physics, including collision with each other and the terrain. Lava stones are created in the eruption hole with a randomized linear and angular momentum, forcing them to burst out of the volcano in a spraying fashion. When the lava stones enter water and sink too deep, they are re-spawned in the volcano and sent out though the hole again.

The physics can be reversed in time, making the stones return to the volcano hole. When they enter the volcano they are placed where they were when they disappeared into the water last, with all the physical properties reset to how they were at that time as well. This makes lava stones come up through the water and moving in a reversed trajectory into the volcano, repeating the feat over and over, and providing a nice visual for the audience.



(a) ...



(b) ...

Figure 3.11: *Volcano eruption.*

3.2 Visual Effects

Boom hacka lacka

3.2.1 Shadows

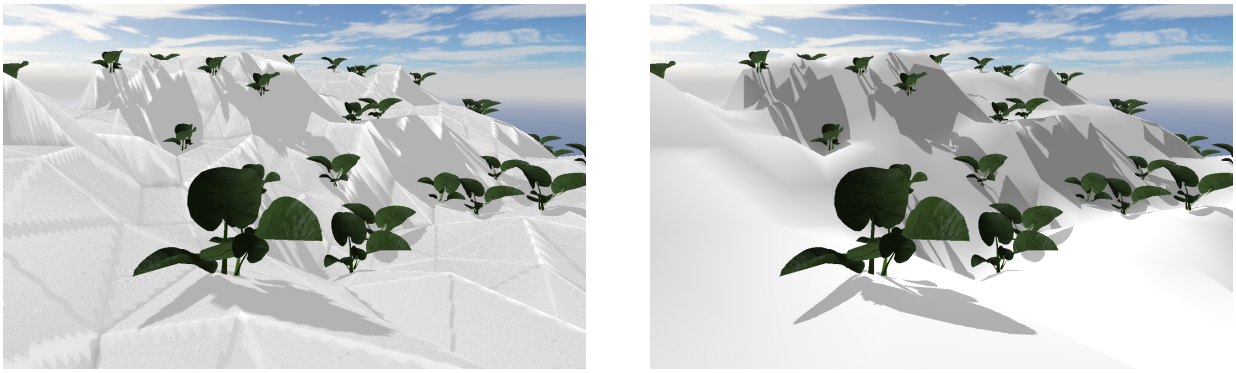
Shadows are one those things that can make a scene really come to life. It will help the viewer to understand the structure of the terrain and location of objects much better than with only shading. There are many techniques in which shadows can be achieved with different pros and cons. We have chosen to use *Shadow Mapping* [2] which by now is a fairly old technique, 1978, but still used in many modern computer games. Shadow mapping offers real-time shadows for arbitrary scenes in a theoretically straight forward way.

The quality of these shadows can be increased arbitrarily, but the computational complexity grows linearly with the size of the rendered screen [2]. Therefor one has to limit this size and go for a number of improvement techniques to make shadows shine.

More specifically our implementation utilizes *Light Space Perspective Shadow Mapping* [3] which can be summarized in the following steps:

- Place the camera in the light source and adjust the camera frustum to cover the part of the scene that will be visible in the final render.
- Render the scene with as simple shaders as possible and store the depth buffer. This is the shadow map.
- Place the camera in its final-render location.
- For each vertex:
  - Transform into light-space coordinates.
  - Compare the distance from the light source with the corresponding value in the shadow map.
  - If the vertex is further away than the shadow map suggests, it will be shadowed.

Once these steps got into place one could naively think that the shadows are done once and for all, but thats not the case. The result out-of-the-box looks something like figure 3.12a. The terrain is full of errors and the shadows have a pretty bad resolution. The erroneous self-shadowing on the terrain is called *Shadow Acne* [4] and is caused by precision errors in the depth test. The value in the depth map and the real depth are too close so the depth test randomly fails. This can be fixed by adding an offset to the depth value which will remove the shadow if the two depth values are too close. The improved result can be seen in figure 3.12b.



(a) Surface suffering from severe acne

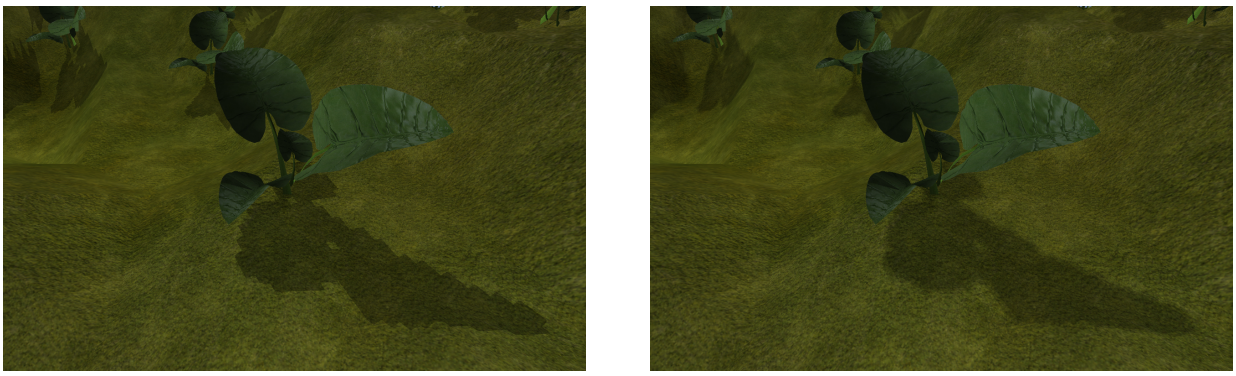
(b) Surface healed by addition of small offset

Figure 3.12: *Comparison of shadow mapping with and without acne*

Care has to be taken when adding this depth offset. If the offset is to large the shadow may be disconnected from its object and cause so called *Peter Paning* [4]. A trade-off has to be made between Shadow Acne and Peter Paning. Thanks to good resolution in our depth buffers we did not need any big offset to remedy our acne problem and did therefore not suffer from any noticeable Peter Paning.

The shadows now look correctly but have a very low resolution when you get close up. The edge is sharp and one can easily distinguish pixels in a not so pleasant may. Increasing the resolution of the shadow map depth buffer will reduce the pixel size of the shadows in the final render but this will decrease the frame rate. We

settled on 2048 x 2048 pixels in our buffer, figure 3.13a. Now my first thought was to low pass filter the shadow map, and this is sort of used in the improvement technique called *Percentage Closer Filtering* (PCF) [5][6]. By randomly sampling the surroundings of the target shadow map pixel and weighting them, in our case with a Gaussian kernel in accordance with [7], one get a smoother shadow edge. The result is definitely a good looking improvement, figure 3.13b.



(a) Out-of-the-box shadows (b) Percentage closer filtered shadows

Figure 3.13: *Comparison of shadow mapping with and without percentage-closer filtering*

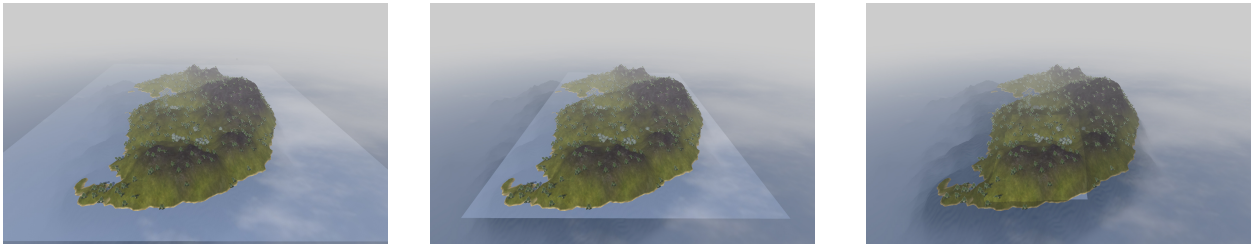
One can keep tweaking these filtering methods forever but they will by themselves never become more than smoothed low-resolution shadows. To reach the next level of shadow mapping, and current industry standards, we need to look for something better. The answer seems to be *Cascade Shadow Mapping* (CSM) [7]. Once I dug into this I started to see CSM everywhere. There are some characteristic transitions in the detail levels that easily can be spotted in many computer games.

The concept of CSM is simple:

- Increase shadow resolution close to the camera
- Decrease shadow resolution further away from the camera

In the optimal case one would adjust the pixel density to always be at least as high as for the target screen density.

The implementation is done by rendering several shadow maps of increasing sizes. One small map just in front of the camera and then bigger and bigger maps further away. The resolution of the maps are the same but by adjusting their sizes the pixel density is high close to the camera and lower further away from the camera. In figure 3.14 some different shadow map sizes can be seen and in figure 3.15 the corresponding increase in quality on low-level inspection.



(a) Shadow mapping level 1 (b) Shadow mapping level 2 (c) Shadow mapping level 3

Figure 3.14: *Overview of different shadow mapping levels*

(a) Shadow mapping level 1 (b) Shadow mapping level 2 (c) Shadow mapping level 3

Figure 3.15: *Close-up of different shadow mapping levels*

It would as conclusion be interesting to take a look at some modern examples, more precisely BattleField 3 (2011), Counter Strike: Global Offensive (2012), SimCity (2013) and League of Legends (2014). One can see that all of these are using shadow mapping in one way or another. The first three are using CSM and PCF combined to improve their shadows, but all are still suffering from ugly transitions between their cascade levels. SimCity holds a nice example of *Perspective Aliasing* [4]. League of Legends uses the simplest out-of-the-box version with no filtering or cascading at all. My point is: this is the way to go according to the cutting-edge game industry. One can also conclude that real-time shadowing is far from a finished topic. In all of the above mentioned games it is easy to find artifacts from optimizations. These are highlighted and amplified in figure 3.16 below.

(a) BattleField 3 (2011) (b) Counter Strike: Global Offensive (2012)

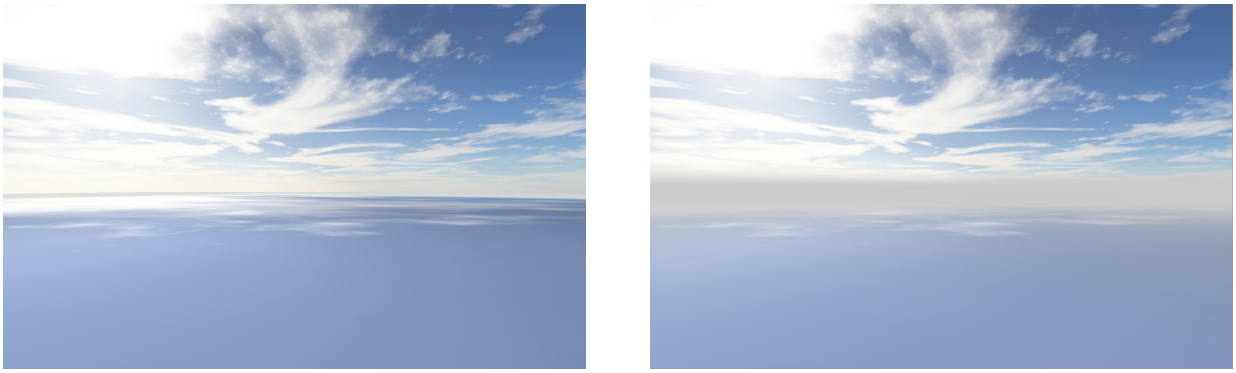
(c) SimCity (2013) (d) League of Legends (2014)

Figure 3.16: *Comparison of current game industry standard shadow quality*

3.2.2 Distance fog

The transition between different parts of the world can sometimes be very sharp in an unpleasant way. For instance, at the border between the sky and the ocean seen in figure 3.17a.

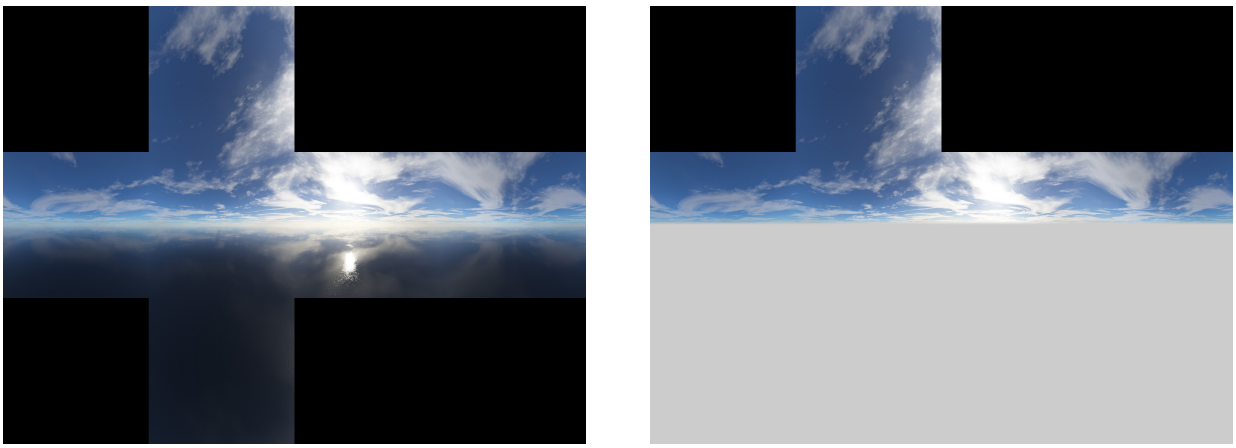
This can be remedied be adding some distance-fog to the ocean. If the color of the fog matches the color of the skybox at its horizon the transition will be seamless. Our skybox has been modify to fade into the color of the fog at its horizon, which can be seen in figure 3.18 below. Notice that we have chosen to not let the skybox be affected by any fog. By doing so one can always see the sky when looking up, which is rather pleasant.



(a) Horizon without fog

(b) Horizon with fog

Figure 3.17: *Comparison of noise functions*



(a) Original skybox

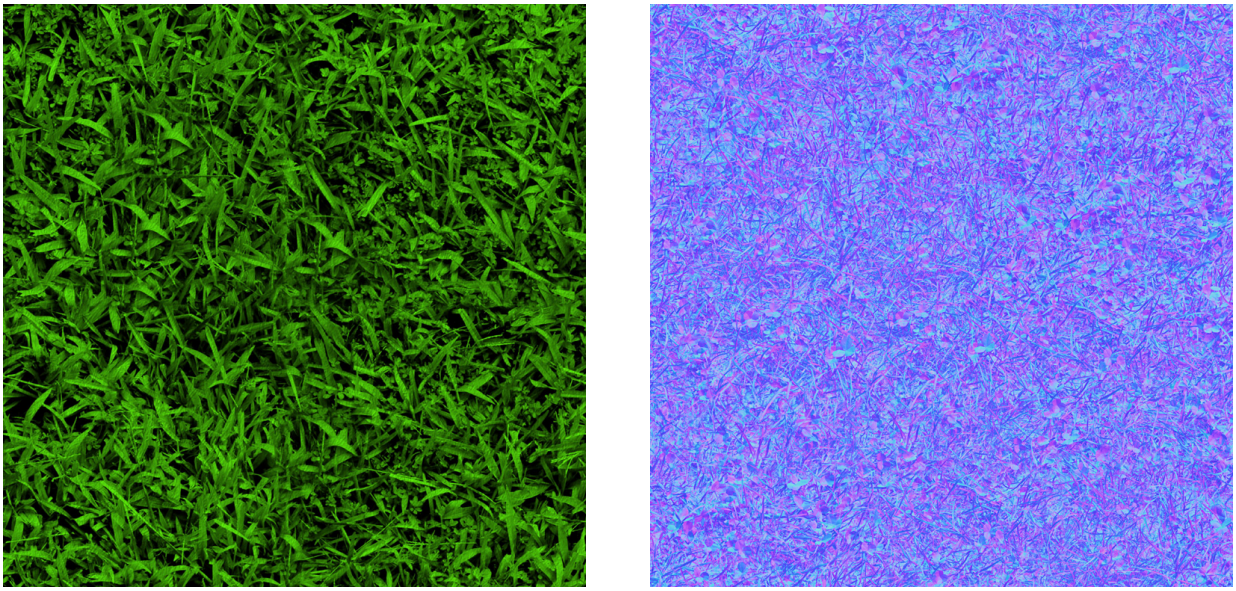
(b) Skybox modified to fade into fog

Figure 3.18: *Comparison of skyboxes*

The distance-fog is also good for constraining the rendering size of the current scene. By adjusting the distance at which the fog appears one can adjust how much that is needed to be rendered of the scene. This enables an arbitrarily large world to be present without killing your computer since only the visible part of the world inside the fog-limit needs to be rendered.

3.2.3 Normal Mapping

Normal mapping is a technique for adding fine details to an object without adding more vertices, which saves a lot of geometry computations. A normal map is generally an image where the RGB-channels represent x,y and z coordinates for a normal vector. This texture is used in the fragment shader when computing the shading for the current fragment. Before calculating the shading the normal vector is rotated to match the object on which it is to be mapped. Notice that the normal vector is not translated since we want it to remain as an normalized direction and nothing more. A normal map is often used in combination with a texture, to give the texture an illusion of depth. An example of a normal map can be seen in figure 3.19.



(a) Grass texture.

(b) Grass normal map.

Figure 3.19: *Example of a normal map.*

4 Conclusions

Awesome

References

[1] Mandelbrot, B. & Van Ness, J. (1968)
"*Fractional Brownian Motions, Fractional Noises and Applications*"
SIAM Review, Vol. 10, No. 4, pp. 422-437

[2] Williams, L. (1978)
"*Casting Curved Shadows on Curved Surfaces*"
Computer Graphics Lab, New York Institute of Technology, Old Westbury, New York 11568

[3] Wimmer, M., Scherzer, D. & Purgathofer, W. (2004)
"*Light Space Perspective Shadow Maps*"
Vienna University of Technology, Austria

[4] Microsoft, Dev Center (2013)
"*Common Techniques to Improve Shadow Depth Maps*"
http://msdn.microsoft.com/en-us/library/windows/desktop/ee416324(v=vs.85).aspx

[5] Reeves, W., Salesin, D. & Cook, R. (1987)
"*Rendering Antialiased Shadows with Depth Maps*"
SIGGRAPH '87, Anaheim
Pixar, San Rafael, CA

[6] Bunnell, M.& Pellacini, F. (2003)
"*Shadow Map Antialiasing*"
Chapter 11, GPU Gems, nVidia

[7] Microsoft, Dev Center (2013)
"*Cascaded Shadow Maps*"
http://msdn.microsoft.com/en-us/library/windows/desktop/ee416307(v=vs.85).aspx

[8]

[9]

[10] Gardel, A., Bravo, I., Jimenez, P., Lazaro, J.L. & Torquemada, A.
"*Statistical Background Models with Shadow Detection for Video Based Tracking*,"
Intelligent Signal Processing, 2007. WISP 2007. IEEE International Symposium on?? Page: 1-6.

[11] Zivkovic, Z. & Heijden, F.
"*Efficient Adaptive Density Estimation per Image Pixel for the Task of Background Subtraction*,"
Pattern recognition letters, Vol. 27, No. 7. (2006), pp. 773-780.

[12] Bernardin, K. & Stiefelhagen, R (2008)
"*Evaluating Multiple Object Tracking Performance: The CLEAR MOT Metrics*,"
Interactive Systems Lab, Institut für Theoretische Informatik,
Universität Karlsruhe, 76131 Karlsruhe, Germany

[13] "*CAVIAR: Context Aware Vision using Image-based Active Recognition*,"
EC Funded CAVIAR project/IST 2001 37540
http://homepages.inf.ed.ac.uk/rbf/CAVIAR/

[14] Hirschmüller, H (2008)
"*Stereo Processing by Semiglobal Matching and Mutual Information*,"
IEEE Transactions on Pattern Analysis and Machine Intelligence, Volume 30(2) pp. 328-341.

[15] OpenCV *Open source computer vision*
http://docs.opencv.org/
Accessed on 2013-12-13