Technical Report

Extrapolate and Conquer

TSBK03 HT 2013

Version 0.1

2014–01–19

Extrapolate and Conquer

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

Participants

Name	Tag	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.

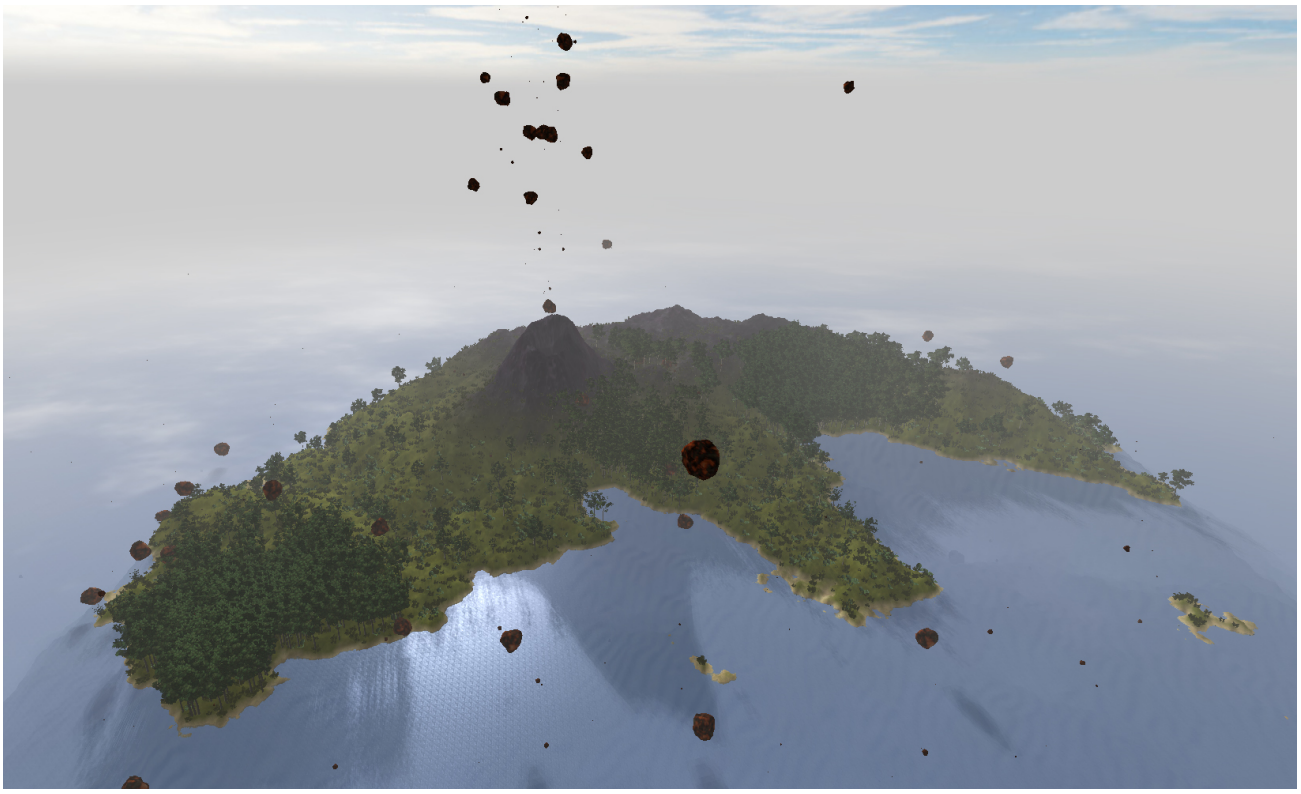


Figure 1.1: A beautiful world. Contarining 3000 trees and 13000 bushes, and a terrain with a vertex density of 5.5.

During the project a general framework for building games/simulations was develeoped, using c++11 template programming to provide an easily used entity-architecture to manage ingame objects. A rendering system cabable of handling large amount of objects without any direct optimizations was also developed alongside the entity-system. The rendering system can manage rendering of large numbers of objects, as well as high quality realtime shadows cast by objects and terrain. A rudimentary world generator based on simplex noise for the terrain generation, and rejection sampling based growth placement.

The project was performed as a part of the course TSBK03 - Techniques for advanced computer games. The code is written in C++11 and GLSL version 150. We use Qt 5.1 as OpenGL wrapper and context provider and OpenCV for some mathematical operations and filtering.

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 scalability as it simplify 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 a component type or a group of component types. How this differs from other traditional architectures can be seen in figure 2.1, which display a few standard game software architectures considering the aspects of structure regarding logic and data.

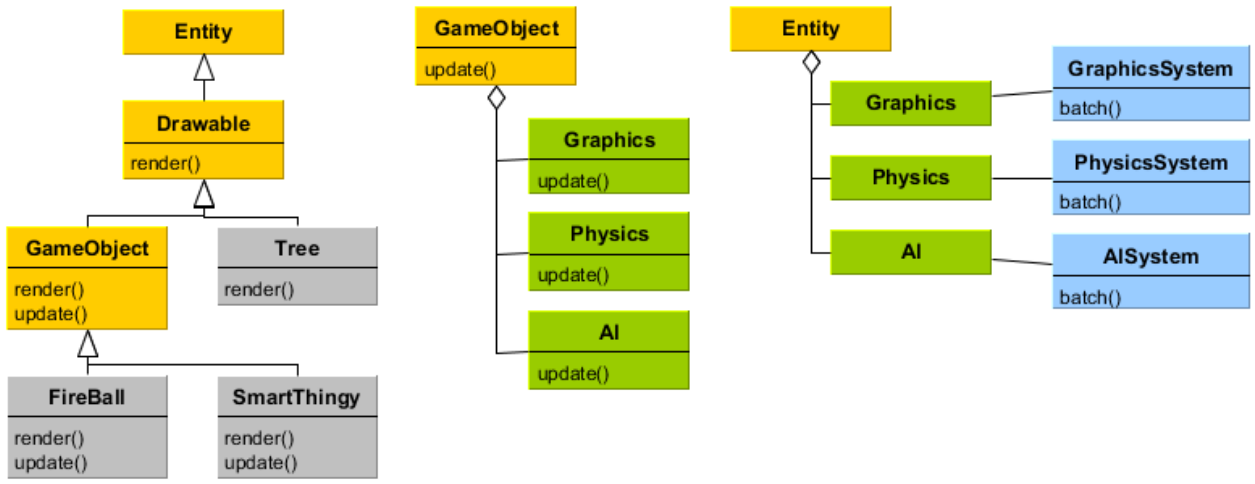


Figure 2.1: A comparison of different game architectures. To the left: A deep hierarchy architecture. In the middle: A Component architecture or an Entity-Component System architecture. To the right: An Entity System.

The first architecture shown is a so called deep hierarchy architecture, a typical OOP architecture, in which inheritance is used to create more abstract objects from less abstract objects. In larger projects it easily lead to class explosion and blob class problems. The second architecture is one in which objects are created as aggregations of different components. Either there is one type for each object type, in which the components make up the entity but doesn't define it, or there is only one object (or entity) class and the components it is made up of defines it. The components all have individual communication needs and logic. The last architecture to the right is an Entity System, in which the data of the components is entirely separated from the logic. The logic is batched over all components of each type. Mick West (2007) writes about Entity-Component Systems. Boreal Games (2013) have an article on gemedev.net with a short but illustrative comparison between deep OOP architectures and Entity Systems. Alec McEachran (2013) explains in an interesting way, using for example linguistics, the benefits of using an Entity System compared to other traditional approaches. He writes about why it is generally so hard to keep the game structured and modular when using a deep OOP architecture and why Entity Systems improve code re-usage and promotes extendability and modularity. Adam Martin (2007) give a practical implementation of a Entity System and talks about its benefits in the light of MMORPGs.

So, in an Entity System an object is an entity label (or identifier) 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 2.2.

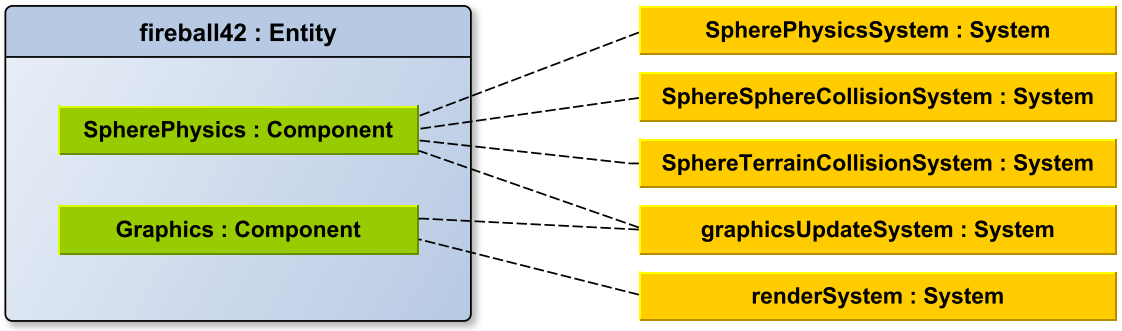


Figure 2.2: A lava stone (or fire ball) used in our demo. The renderSystem is in our current implementation not actually part of the Entity System, but it could easily be integrated.

Our Entity System is built using sophisticated meta template programming techniques. It is a compile time construct, enabling consistency verification, code generation and error correction. All components are controlled by the EntityManager class and each component type is stored continuous in memory. We provide an Entity class which work as an interface to the entity label and the components which builds up the entity, this through the Entitymanager. The Entity interface is shown in figure 2.3 and an example of the construction of the fire ball in figure 2.2 is shown in figure 2.4. All access to components are reference based to minimize copying and the risk of pointer errors. 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 components used and their dependencies are specified by the developer in the source code, and the program is optimized at compile time based on these specifications in regard to efficiency and robustness. The program writes part of itself to be maximum efficient in terms of component storage, data access as well as requirement lookup and enforcement. The later also ensures robustness of the systems as code is generated to handle the specific requirement-tree specified. The code generation and compile time logic is done using C++11 meta template programming.

```
11      entity.has<Component>();
12      entity.get<Component>();
13      entity.add<Component>();
14      entity.remove<Component>();
```

Figure 2.3: Entity interface. Component is a place holder for any component type. *has* return a boolean, *get* return a reference to that component (if the entity consists of such a component), *add* adds the component to the entity and returns a reference to it, *remove* remove the component from the entity.

```
42      Entity & entity = entityManager.createEntity();
43
44      entity->add<SpherePhysics>();
45      entity->add<Graphics>();
46
47      SpherePhysics & physics = entity->get<SpherePhysics>();
48      physics.position = QVector3D(0.0, 0.5, 1.0);
49      physics.friction = 0.5;
50      ...
51
52      Graphics & graphics = entity->get<Graphics>();
53      graphics.object = new Object(resourceManager->getModel("stone1"),
54                                   resourceManager->getShader("phongTex"),
55                                   resourceManager->getTexture("lava1"));
```

Figure 2.4: The creation and initiation of an entity.

A component is simply a *struct* or a *class* which inheritance the Component class. The only requirement is that it implements a getName() method returning the name of the component as a string. A part of the SpherePhysics component is shown in figure 2.5, and it is shown in higher detail in figure 2.9.

```
45  struct SpherePhysics : public Component<> {
46      const std::string getName() override { return "SpherePhysics"; }
47
48      float mass;                     // m            (Positive)
49      float elasticity;               // epsilon      (Between 0 and 1)
50      float friction;                 // friction     (Between 0 and 1)
51      float momentOfInertia;          // Should be a matrix in the general
52      float gravitationalConstant;    // g            (Positive. in Sweden
53
54      QVector3D position;             // x = Integral( v, dt );
55      QVector3D velocity;             // v = P / m;
```

Figure 2.5: A part of the SpherePhysics component. The template argument <> is empty, meaning that no system based on SpherePhysics require any other component to be present as well.

Every component must be registered in the EntityManager before it is used. If a component is used anywhere in the Entity System without registration a compile time error will be generated and supplied to the developer. Since C++11 doesn't provide template parameter lists we have to use a define at this one place, until the next standard hopefully is released. The registration is shown in figure 2.6.

```
10  // List all components
11  #define Components Name,SimplePhysics,Graphics,SpherePhysics,AI
```

(a)

```
47      EntityManager<Components> entityManager;
```

(b)

Figure 2.6: *Component registration. (a): Component is now an alias for the list of component types used in the Entity System. (b): The EntityManager is implemented with a list of every Component to be used in the Entity System.*

Systems are classes which must inheritance the System class and either overload *processStep* (which is the action to be performed on a single component) or *batch* if something involving multiple entities at the same time much be performed e.g. in the collision detection. An example of a simple dummy-system is shown in figure 2.7.

```
8   class SimplePhysicsSystem : public System<SimplePhysics, Components>
9   {
10  public:
11      void processStep(SimplePhysics & physics) override {
12          physics.position += physics.velocity;
13          Entity<Components> & e = getEntity(physics);
14      }
15  };
```

Figure 2.7: A dummy system. It iterates primarily over all SimplePhysics components. Illustrated is also the access to the entity from its component.

To guarantee consistency for systems requiring multiple component types, dependency between components is supported as shown in figure 2.5. Circular dependencies are supported as well. A system which require component A and B can then for example fetch all A, and if A require B then a B will be available to the system for each A. This since any Entity consisting of A will be guaranteed by the program to also consist of B. This is enforced by the Entity System such that when a component is added to an entity then any component required as well will also be added, recursively. Similarly if a component is removed which other components belonging to the entity require. Then all components no longer supported are also removed, recursively. The code to check, add and remove components due to such requirements is generated by the Entity System such

that no components are checked unnecessary for any component in any add/remove situation. This makes the system have the minimal overhead possible while still allowing arbitrary dependencies between components.

The code generation is written using meta meta templates from our meta-library, also written for the project. An example of such a function call is shown in figure 2.8. What is shown is an except of the *addComponent* method of the EntityManager, which is called by the Entity method *add<Component>()* in turn. It generates all the *has<Component>()* and *add<Component>()* calls necessary to fulfill the requirement the current Component has and then calling all these methods. The *add<Component>()* will in turn go through the same process, but with a possibly different component which requirements must be fulfilled, and so on until all requirements are fulfilled for all components in the dependency graph.

```
1  // If additional Components are required, add these aswell (recursively)
2    meta::FOR_EACH< typename Component::REQUIRED_COMPONENTS, // List (items) to iterate ↩
       over
3
4                    ADD_COMPONENT,                          // Template to apply
5                                                            // on each item.
6                    std::tuple<Component, Components...>,   // Additional template
7                                                            // parameters to above ↩
                                                                template.
8                    std::tuple<Entity<Components...>&>      // Argument types to the above
9                                                            // templatet's execute-↩
                                                                function.
10                 >::execute(entity);
```

(a)

```
1  template<typename Element, typename Component, typename... Components>
2  struct ADD_COMPONENT {
3      static void execute(Entity<Components...> & e) {
4          if(!e.template has<Element>()) {
5              e.template add<Element>();
6              Component c;
7              Element el;
8              std::cout << "Entity (" << e.getID() << ") warning: ";
9              std::cout << "Adding \"" << c.getName() << "\" require \"" << el.getName()
10                       << "\" which is missing, it is now automatically added.\n";
11         }
12     }
13 };
```

(b)

Figure 2.8: Code generation of adding required components. (a): An except of the *addComponent* method of the EntityManager, a meta meta template function call which assures that all components that are required, starting with the current Component is added to the entity. (b): The *ADD_COMPONENT* template used in (a). All lines except the first in the if-statement are there for debugging purposes.

The case of removing a component is slightly more advanced since it is a reverse problem of the problem outlined above: when a component is removed, any component requiring it must also be removed. It is solved in a concise fashion and implemented similarly to above, thanks to the powerful meta-library. See the source code for the actual implementation.

The benefits of our Entity Systems is that it is easy to use and is non-intrusive. It is easy to maintain, easy to extend and extremely efficient. It is trivial to parallelize calculations, e.g. portion the component set in the batching to be handled by different threads in parallel. Our system also verifies consistency at compile time. The later could probably be improved even more and open up vast possibilities of the construction of reliable software. Another advantage from the batching is the possibility of much higher performance, as we maximizing caching since components are continuous in memory, and minimizing cache misses since we minimize the need for conditional branches in many cases due to the System approach in Entity Systems.

2.2 Rendering

The rendering system is outside the entity system mainly because it is the largest individual 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 being integrated into it.

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

In order to correctly draw the growth some form of transparency was needed as the foilage used billboards. 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 breakts the foliage textures due to the background bleeding into the texture at lower resolutions. The solution to this is to make sure the transparent parts of the foliage textures have roughly the same color as the leafs at the edge between opaque and transparaent, so that the parts where the background would be visible if there had been no color instead has roughly the same color as the opaque part of the leaf. This still leavse the issue of the transparency bleeding into the foilage at lower resolutions, this is solved by varying the alpha test value depending on the distance to the observed fragment. At longer distances the treshold is lower, meaning there are some artifacts from the background, and the leafs loosing some shape, but as it is so far away it is not noticable.

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 type and a position,orientation and scale for each instance in the list. The position,scale and orientaion is then used to compute a list of model matrices, that is uploaded to the 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, indvidual 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 and thus the only object that needs to be sorted to be drawn properly. This does have the side effect of making the trees invisible from under the water tough. For future development a sorted render pass might be a good idea to implement, that would support arbitrary numbers of semitransparent objects, and draw them correctly relative to the trees/sorting-indepdent objects.

In general this demonstrates the need for a rendering pipeline that does not draw each object individually, but rather as a set of render passes, for example one for shadows, one for object geometry and one for post-processing.

Due to a rather lousy bush model with a great deal of z-fighting with itself, the depth mask is disabled while drawing bushes as this is considerably faster than finding or creating new models. It does require the bushes to be drawn last and that there are no other models needing this treatment, making it something of a hack.

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. In our current demo some of the physics is tuned to be less realistic but to allow for a greater chance of the lava balls to enter the ocean rather than get stuck at land. This is done in order to provide a better show for the audience. A major problem of the physics is that it is hard to dampen small oscillating motions of the rocks when they have come to an halt, or close to it. Decreasing the energy of the system was tried, which gives some results but makes the physics otherwise less spectacular as well as inhibit interesting backwards phenomenons.

2.3.1 SpherePhysics

SpherePhysics is a component dedicated to physics data, its content is seen in the code below in figure 2.9. 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

};
```

Figure 2.9: The SpherePhysics component. All lava stones are composed by this component to enable physics and collision handling.

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 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.

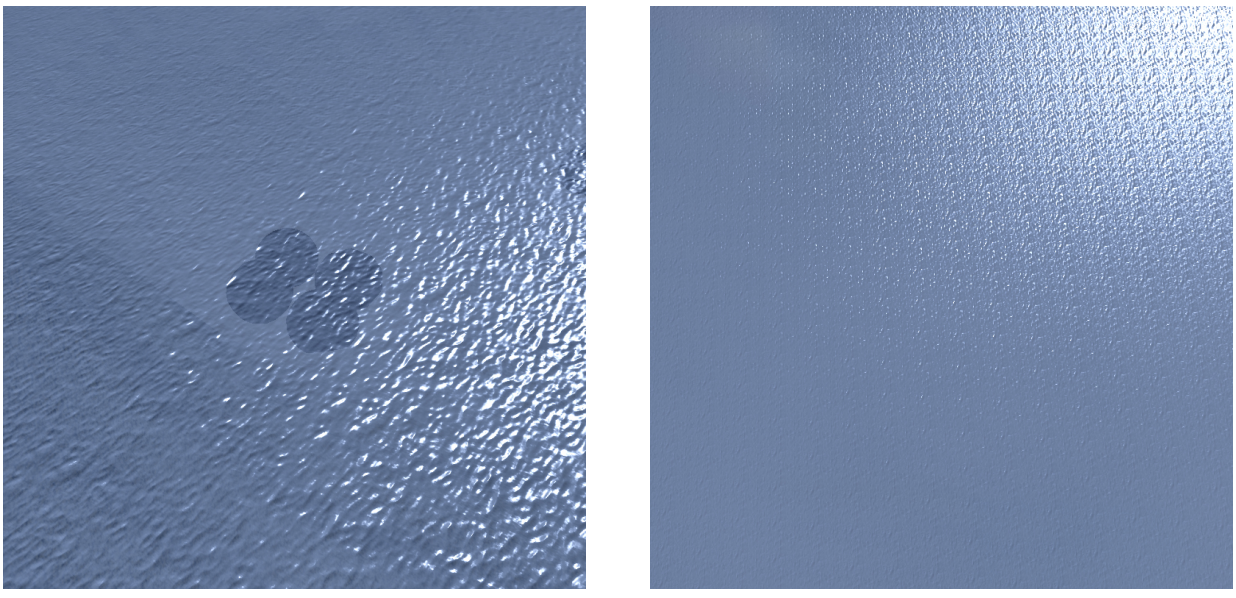
A procedural world is generated by carefully chosen algorithms. Our world is procedurally generated anew in a new unique constellation on every run, or a seed can be provided to generate specific worlds. The terrain is first formed, then the ground texturing and the vegetation, both dependent on properties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

3.1 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.

3.2 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: *Water seen close and from a far.*

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.

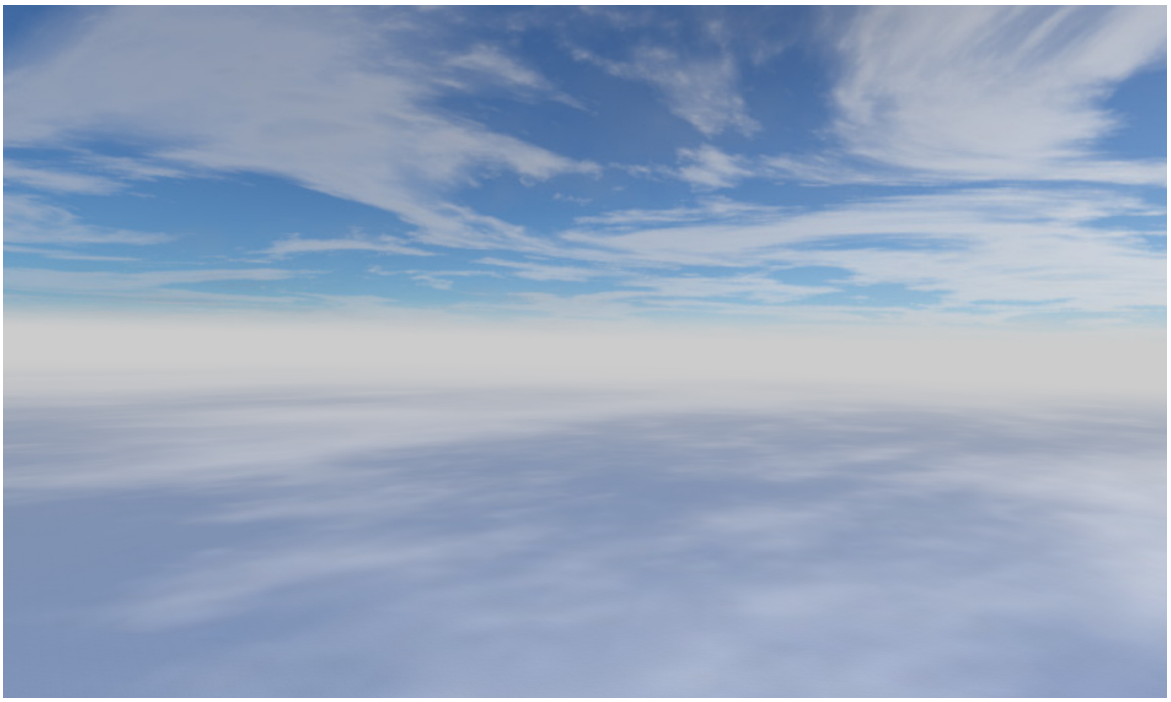
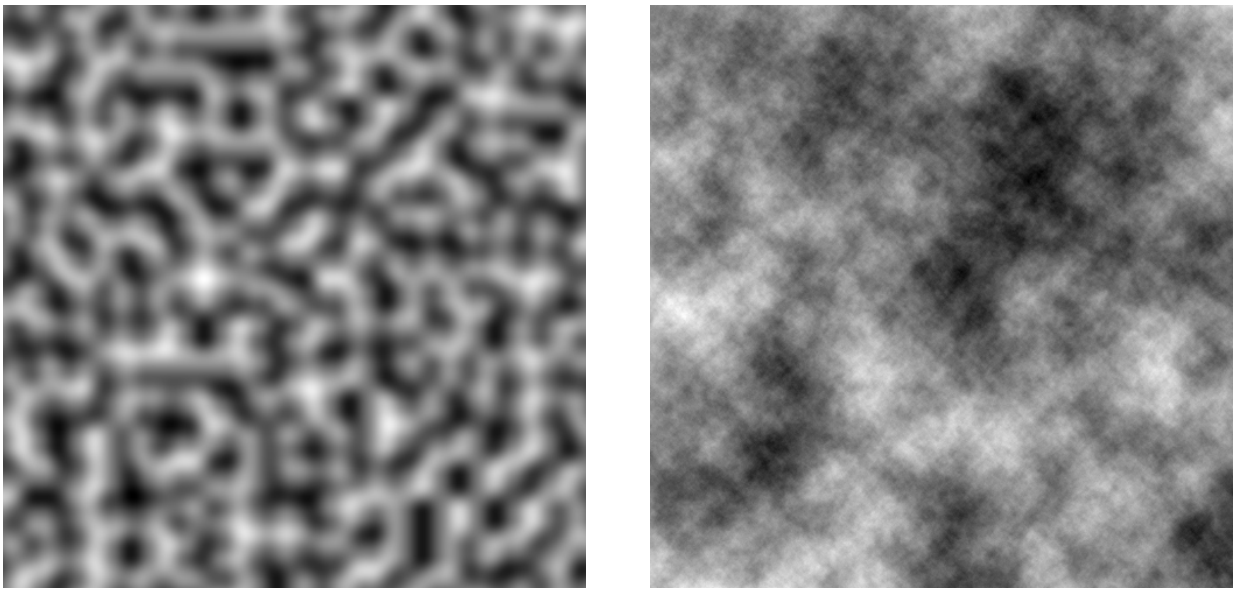


Figure 3.3: The sky reflected in the ocean.

3.3 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, see figure 3.6 which show a few different sample densities. A further possible extension would be to use tessellation, which would allow for a terrain with arbitrary fine details.



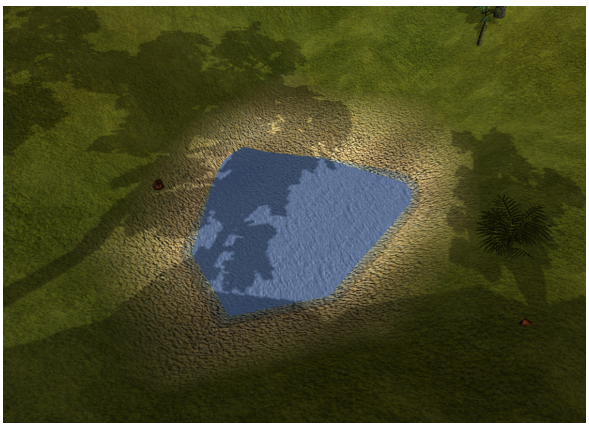
(a) Height map generated from single-octave simplex noise (b) Height map generated with Fractional Brownian Motion

Figure 3.4: *Comparison of noise functions*

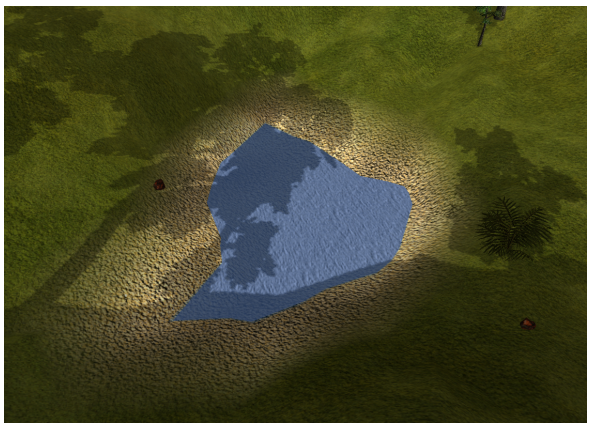
The height map is finally assured to always slope into the ocean. This is done by first setting the all edge values to zero, then weighting the entire map with a thresholded Gaussian kernel, see figure 3.5 below.

(a) Height map with edge set to zero. (b) Thresholded Gaussian kernel (c) Final height map.

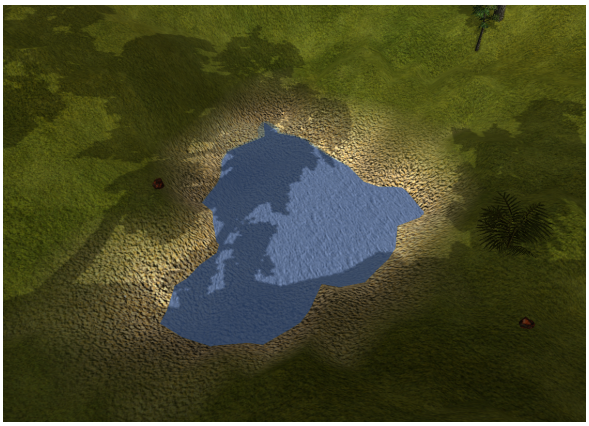
Figure 3.5: *All steps in the creation of the height map.*



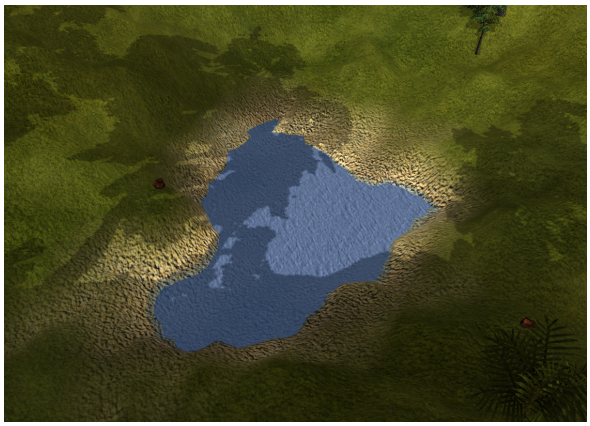
(a) Density 0.5



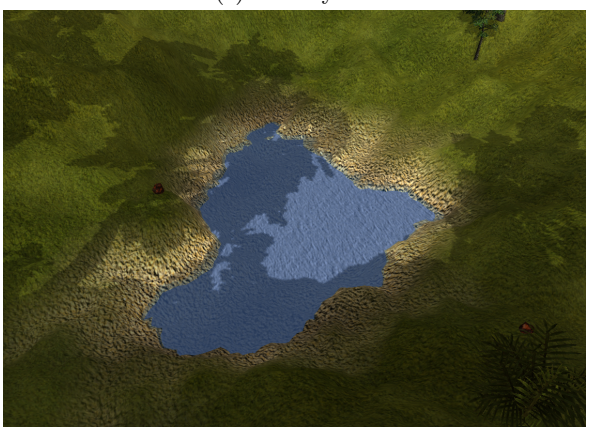
(b) Density 1.0



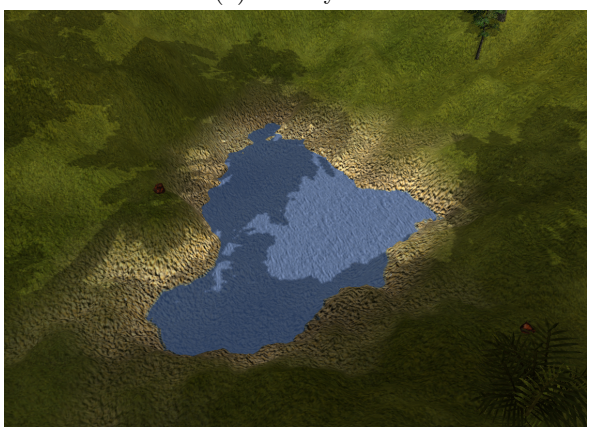
(c) Density 2.0



(d) Density 5.0



(e) Density 10.0

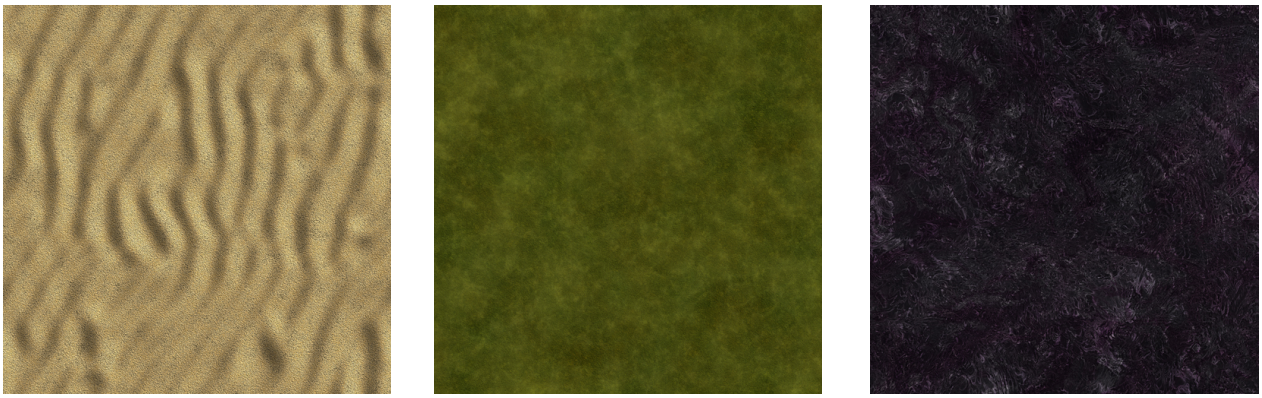


(f) Density 15.0

Figure 3.6: A comparison of different vertex and sample density, from 0.5 to 15 vertexes per OpenGL length units. Waves are shown in some of the images, as the wave generator currently doesn't distinguish between lakes and oceans.

3.4 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 3.7. These textures were generated in Spiral Graphics Genetica Viewer and are procedurally generated tiling textures.



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

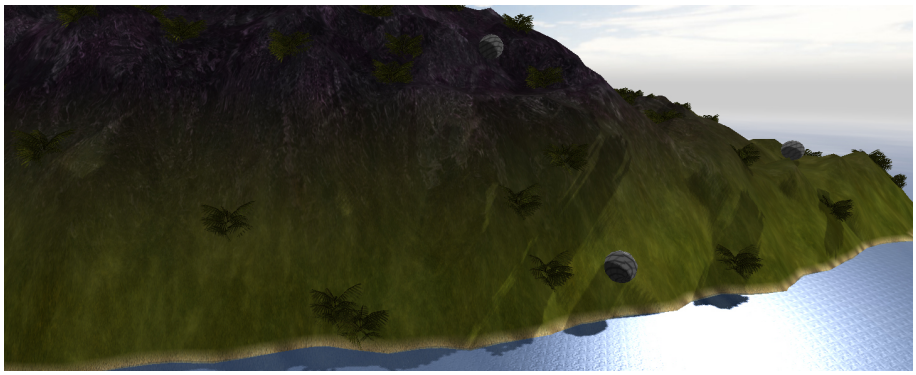
Figure 3.7: *Textures used for the ground.*

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 3.8 and 3.9 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.

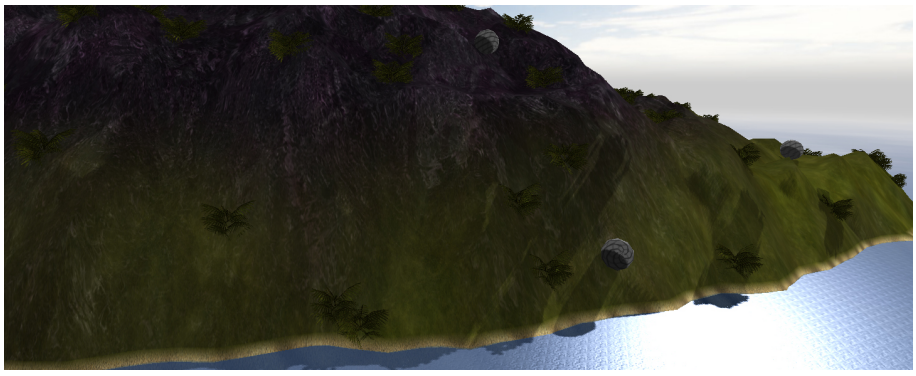
(a) Simple linear interpolation.

(b) Advanced non-linear texture blending.

Figure 3.8: 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 deep mountain slope to the right in (b).



(a) Simple linear interpolation.



(b) Advanced non-linear texture blending.

Figure 3.9: 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).

3.5 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 as well, thus plants are placed using two slightly different approaches.

First a base layer of smaller bushes and trees 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.

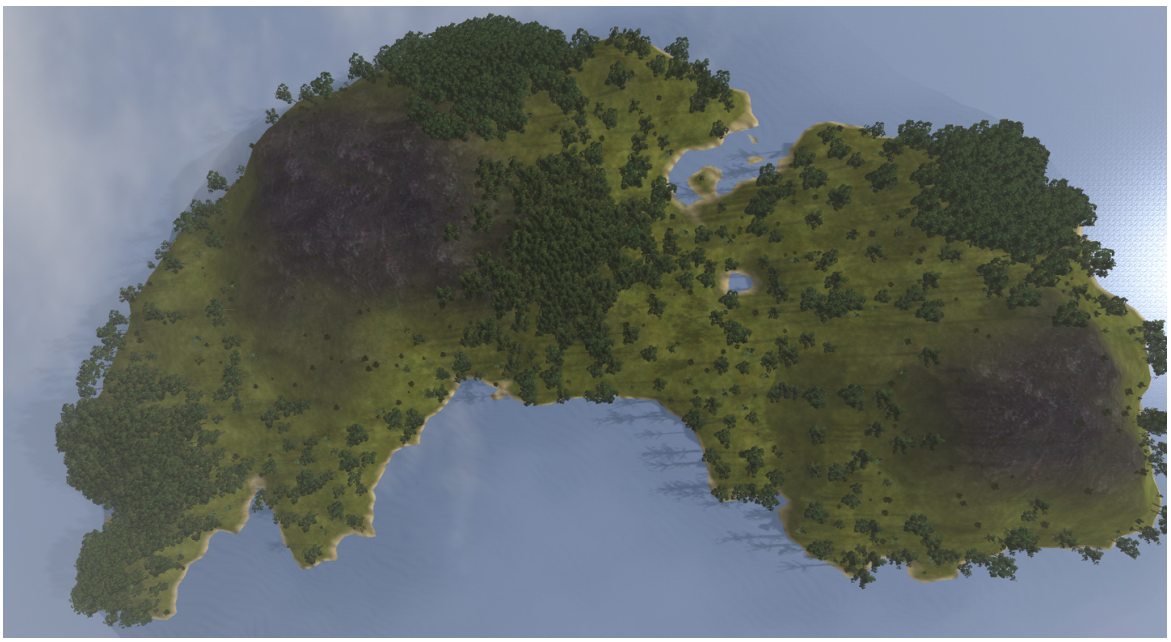
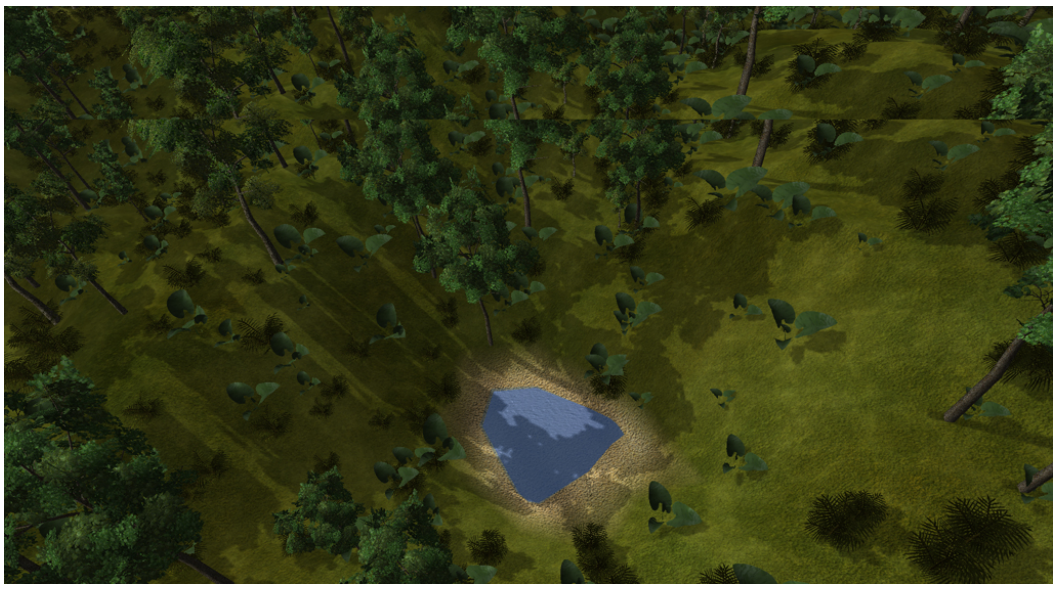


Figure 3.10: Island viewed from the top. The different forests are clearly visible as separate areas without having sharp edges, along with some bushes scattered across the island.

Placing a tree or a bush is done by first generating a coordinate, checking it for suitability and then placing a tree if it is not unsuitable. For example no trees can be placed in the water, or to far up on a hill.

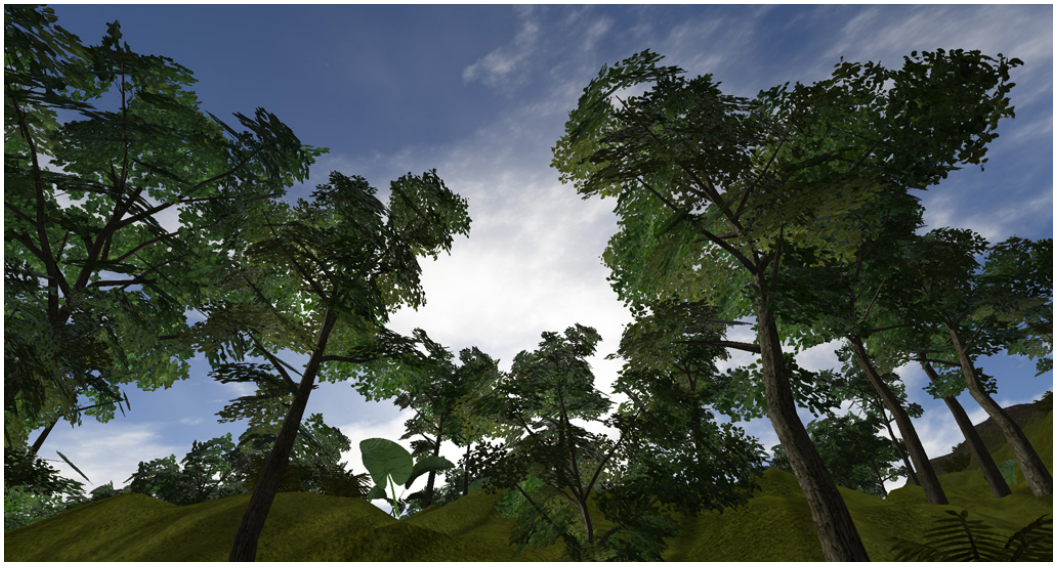
All trees consist of two models, one trunk and one with the foilage, they aligned so that if the trunk and foilage is placed at the same coordinate, the foilage will end up in the correct place relative to the trunk.

The bushes are simple billboard models. The bushes and the trees, both textures and models, were take from TurboSquid.com.



(a) A small water hole in the middle of the forest at the late afternoon.

(b) The same water hole seen in (a), but perceived through the vegetation.



(c) Trees in the outskirt of a forest, portrayed against the sun.

3.6 Volcano

The volcano generator changes the terrain on a spot to that of a volcano, such as the one seen in figure 3.12.

Figure 3.12: A volcano in full eruption.

The volcano is made from one addition of a 2D-Gaussian kernel to the terrain, and of a subtraction of a smaller 2D-Gaussian kernel which creates the eruption hole. It erupts and spews out lava stones as a particle effect. The generator is currently set to generate one volcano and to place it 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 through the hole again.

The physics can be reversed with respect to 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.

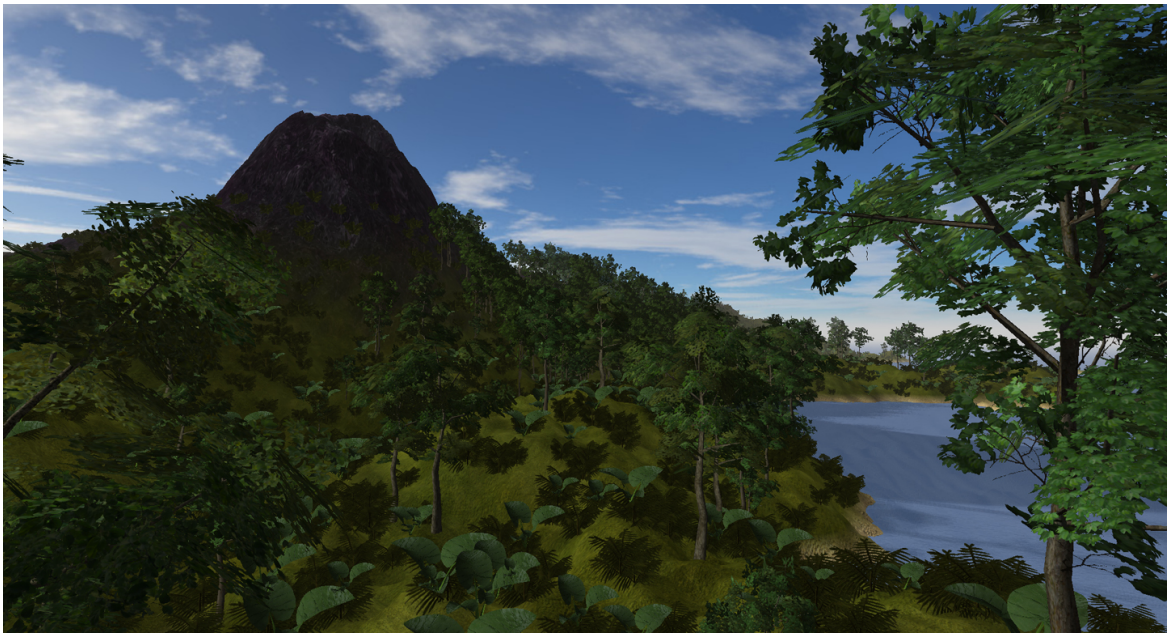
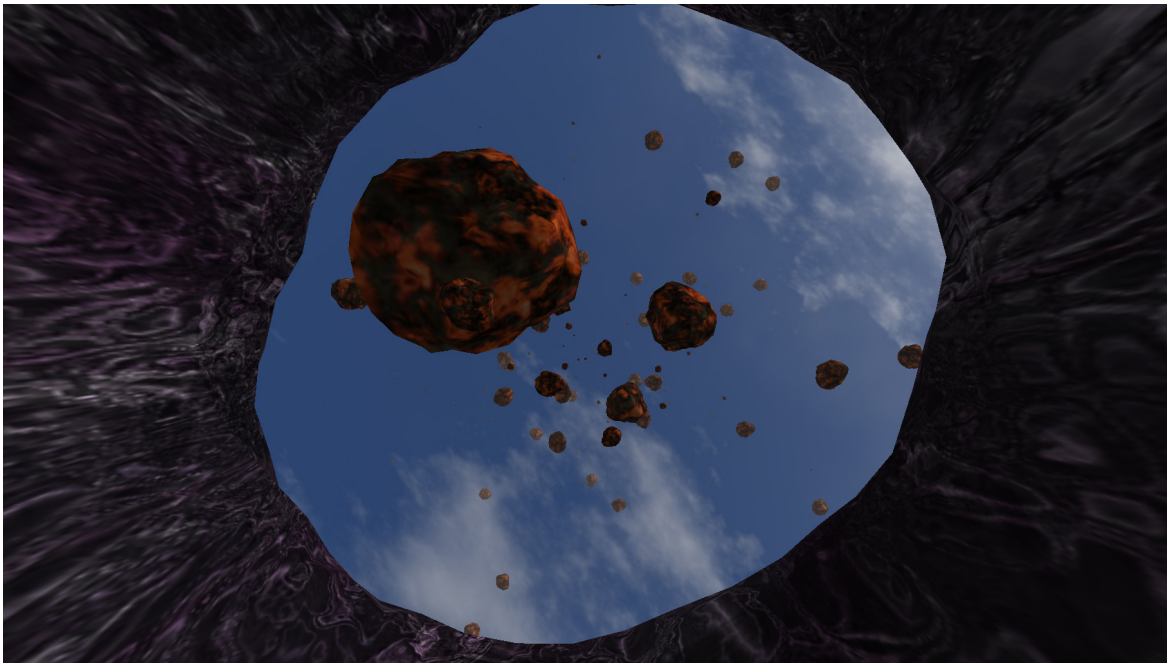
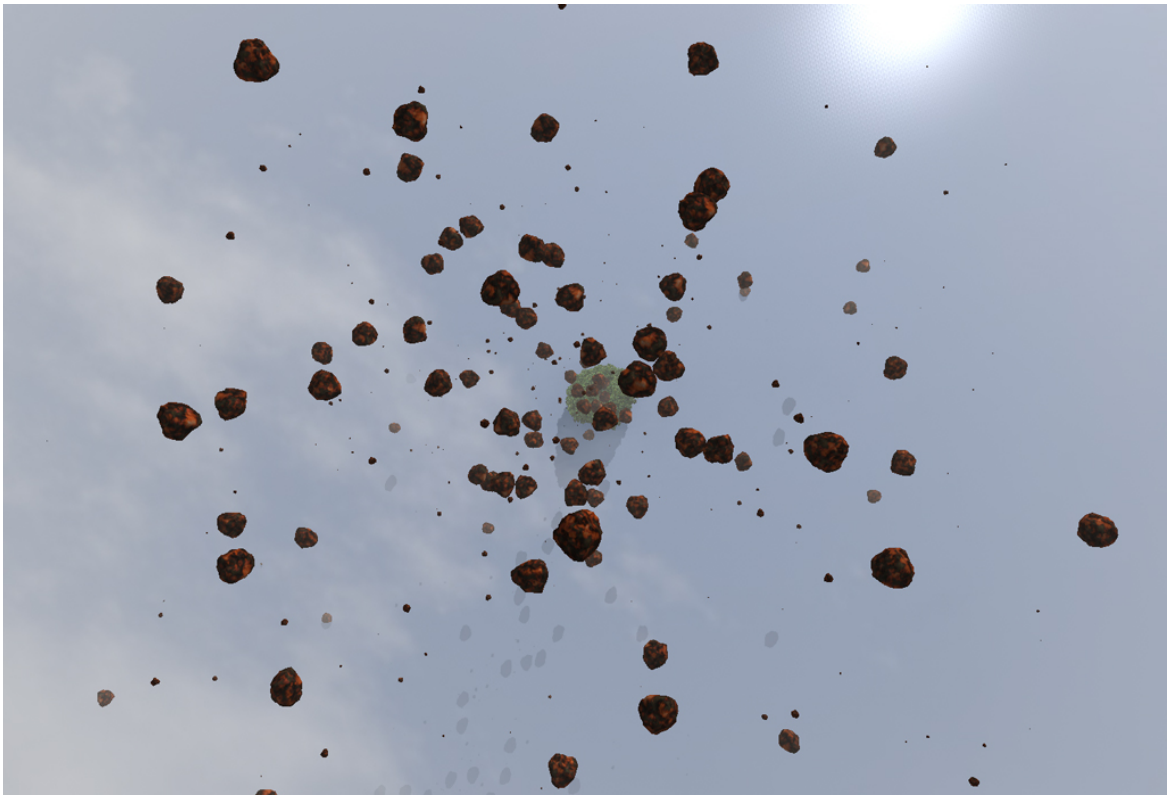


Figure 3.13: A (currently) dormant volcano and its surrounding.

A typical volcano and its surroundings is seen in figure 3.13. In figure 3.14 other views of the lava stone particle effect is shown.



(a) A Volcano eruption seen from inside the volcano)



(b) Volcano eruption on a small island seen from above.

Figure 3.14: *Volcano eruption, seen from different angles.*

4 Visual Effects

Various visual effects were used in this project and we will in this chapter go through some of them in more detail.

4.1 Shadows

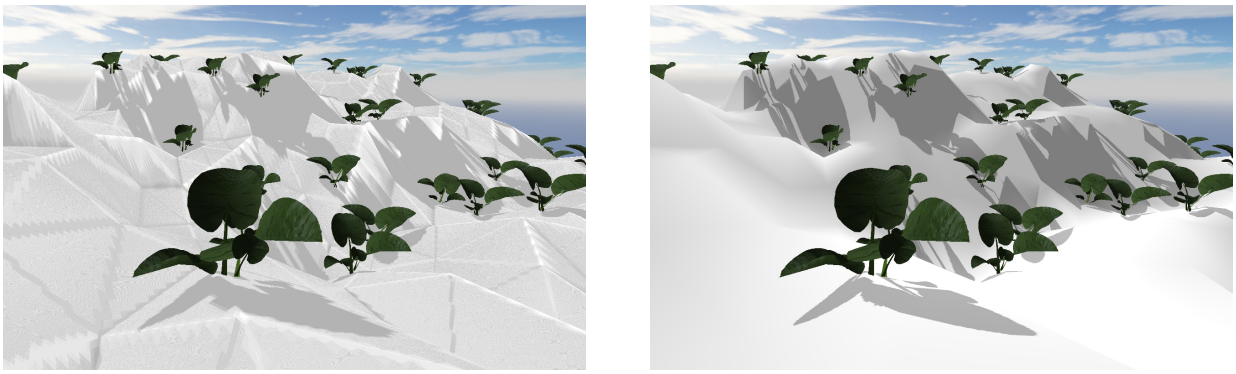
Shadows are one those things that can make a scene really come alive. It will help the viewer to understand the structure of the terrain and location of objects much better than with only shading. There are many techniques in which shadows can be achieved with different pros and cons. We have chosen 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.

First of, 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 since we only want to 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 4.1a. 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 4.1b.



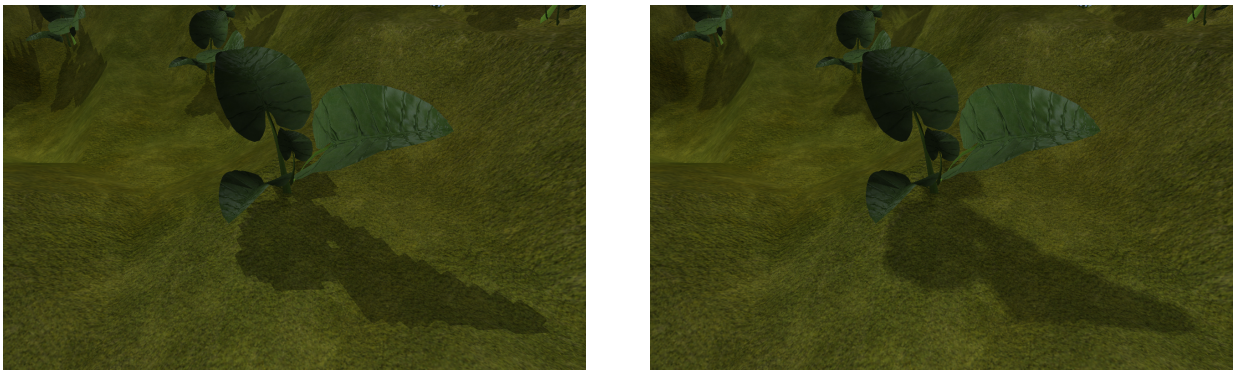
(a) Surface suffering from severe acne

(b) Surface healed by addition of small offset

Figure 4.1: *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 4.2a. 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 4.2b.



(a) Out-of-the-box shadows

(b) Percentage closer filtered shadows

Figure 4.2: *Comparison of shadow mapping with and without Percentage Closer Filtering*

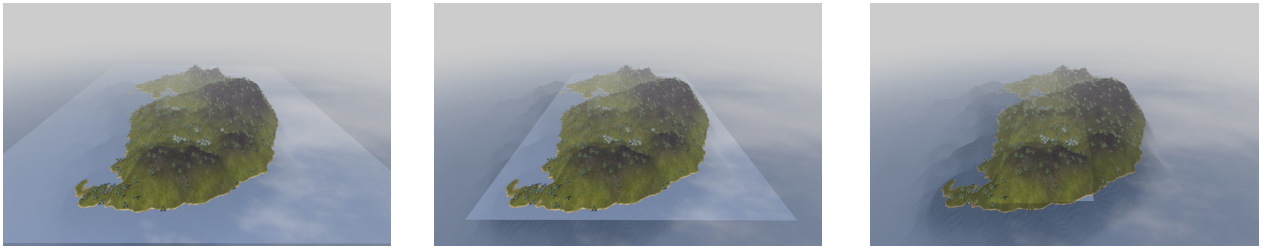
One can keep tweaking these filtering methods forever but they will by themselves never become more than smoothed low-resolution shadows. To reach the next level of shadow mapping, and current industry standards, we need to look for something better. The answer seems to be *Cascade Shadow Mapping* (CSM) [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 4.3 some different shadow map sizes can be seen and in figure 4.4 the corresponding increase in quality on low-level inspection.



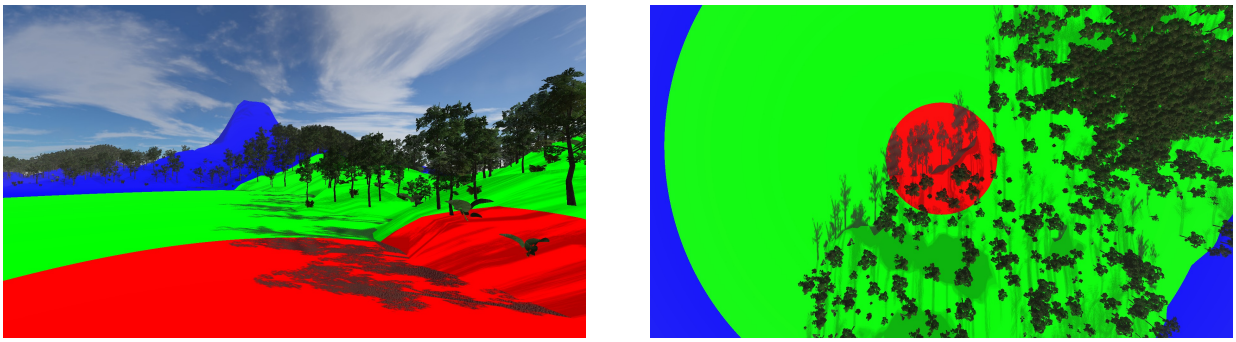
(a) Shadow mapping level 1 (b) Shadow mapping level 2 (c) Shadow mapping level 3

Figure 4.3: *Overview of different shadow mapping levels*

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

Figure 4.4: *Close-up of different shadow mapping levels*

The tricky part of CSM is to find a good way to divide the view frustum. This is discussed thoroughly in [7] which is where we found most of our inspiration on this subject. We thing that we do differently is that we let the light-source frustum focus in the extension of the look-at vector at ground level. This way, the shadows always have good quality where you look, instead of where you are. This is of great interest for us since we allow the user to fly high over the terrain. Our implementation still generate good shadows for this scenario. An example of the frustum division can be seen in figure 4.5a.



(a) Frustum division near ground. (b) Frustum division from high above.

Figure 4.5: *An example of the frustum division for cascade shadow mapping.*

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 4.6 below.

(a) BattleField 3 (2011)

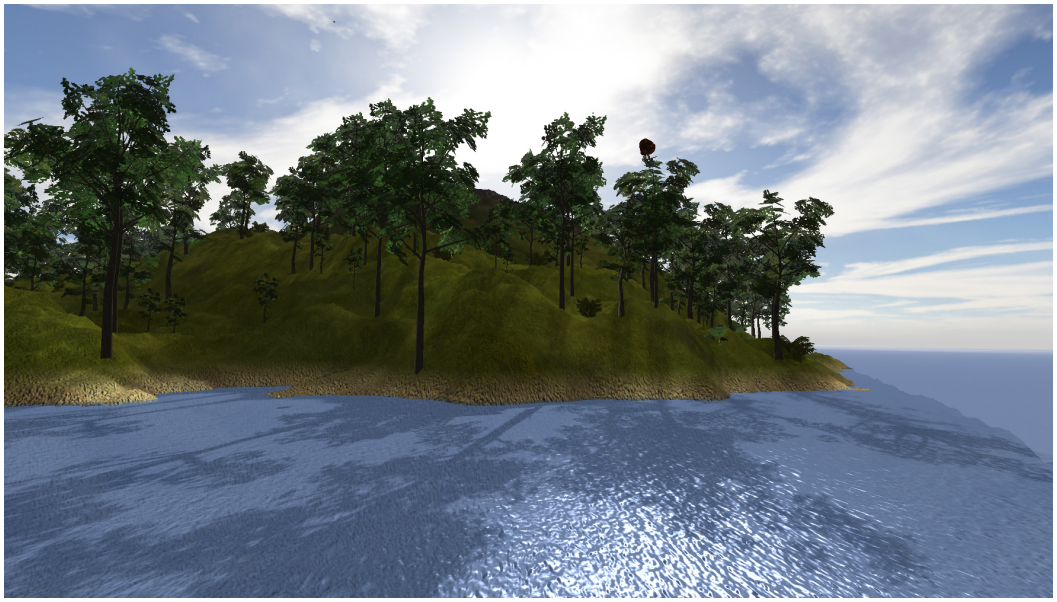
(b) Counter Strike: Global Offensive (2012)

(c) SimCity (2013)

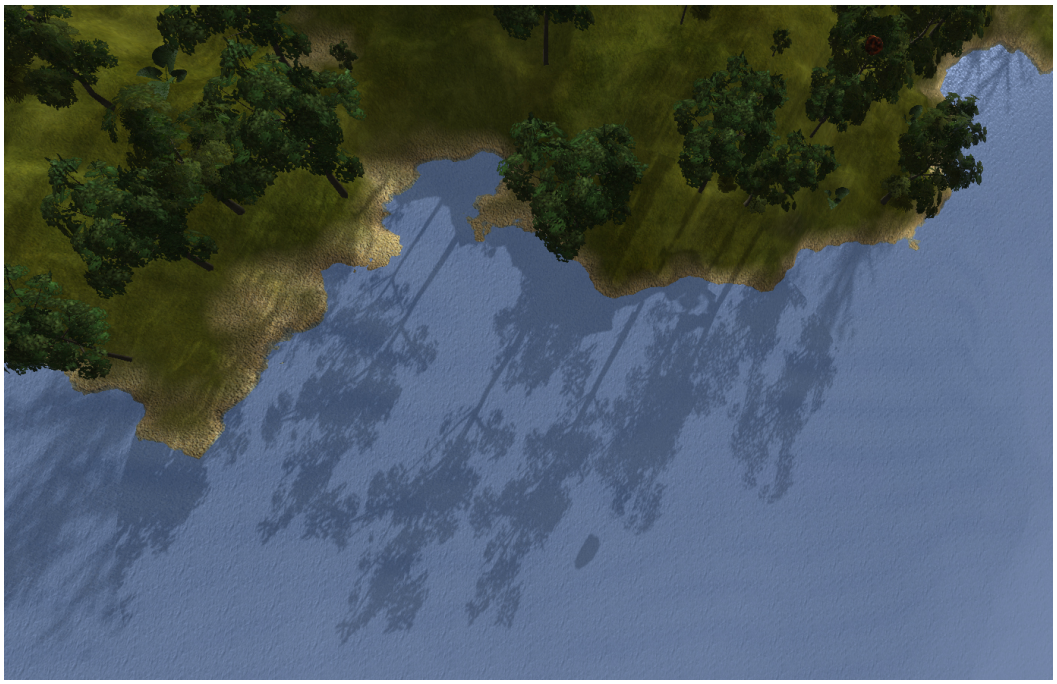
(d) League of Legends (2014)

Figure 4.6: *Comparison of current game industry standard shadow quality*

Our final in-game results can be seen in figure 4.7.



(a) From the side, looking into the sun.



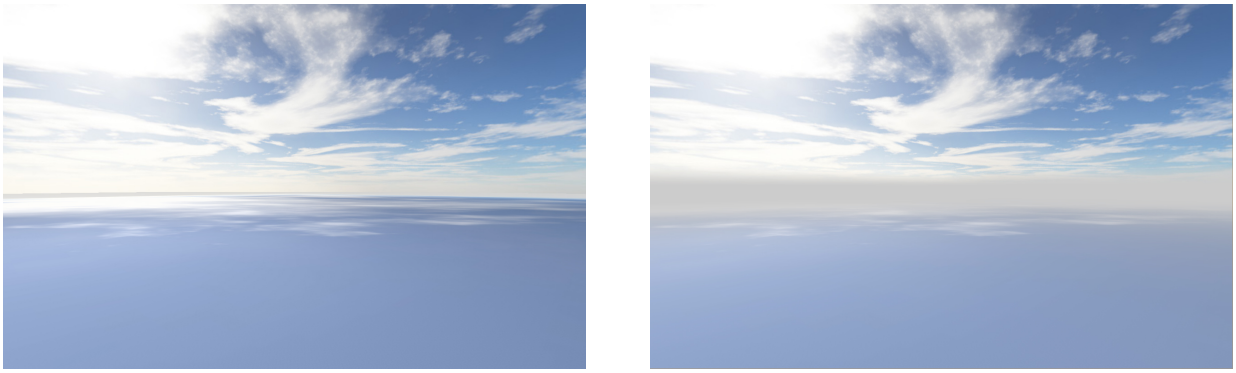
(b) From the top.

Figure 4.7: Two images of the same scene, showing our in-game shadow result. In (b) the cascade with the highest details do not cover the entire area, lower detail shadows can be seen to the left and to the top-right.

4.2 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 4.8a.

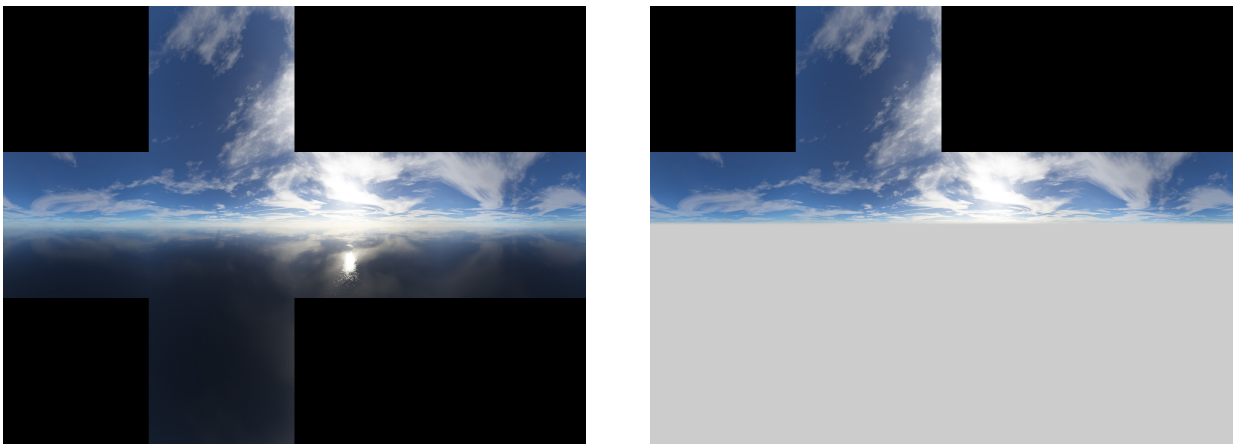
This can be remedied be adding some distance-fog to the ocean. If the color of the fog matches the color of the skybox at its horizon the transition will be seamless. Our skybox has been modify to fade into the color of the fog at its horizon, which can be seen in figure 4.9 below. Notice that we have 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 4.8: *Comparison of noise functions*



(a) Original skybox

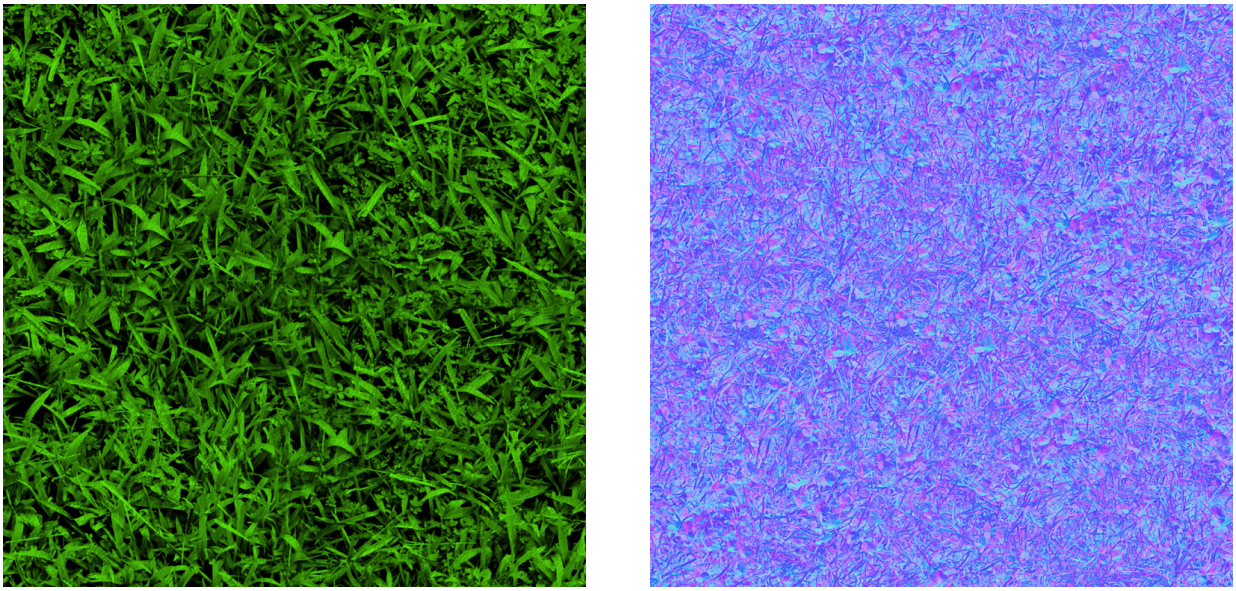
(b) Skybox modified to fade into fog

Figure 4.9: *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.

4.3 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 4.10.



(a) Grass texture.

(b) Grass normal map.

Figure 4.10: *Example of a normal map.*

5 Conclusions

We can generate a new world, with different terrain and placement of bushes, trees, forests and volcano. A world with 0.5 vertex density takes about 23 ms on a single 3.4 GHz Intel i5 core, which is fairly fast. The world looks very nice both close, however with tesselation even more so, and both far and very far away. An example of a world seen from far above is shown in figure 5.1. The forests and the nature look really good compared to many games. It is a very fine foundation to build a game on, either a FPS or even a RTS or RPG, since it looks good at all ranges and the world can be big enough. With further work such as LOD levels for the vegetation and tesselation for the land, even vaster and richer worlds could easily be made.

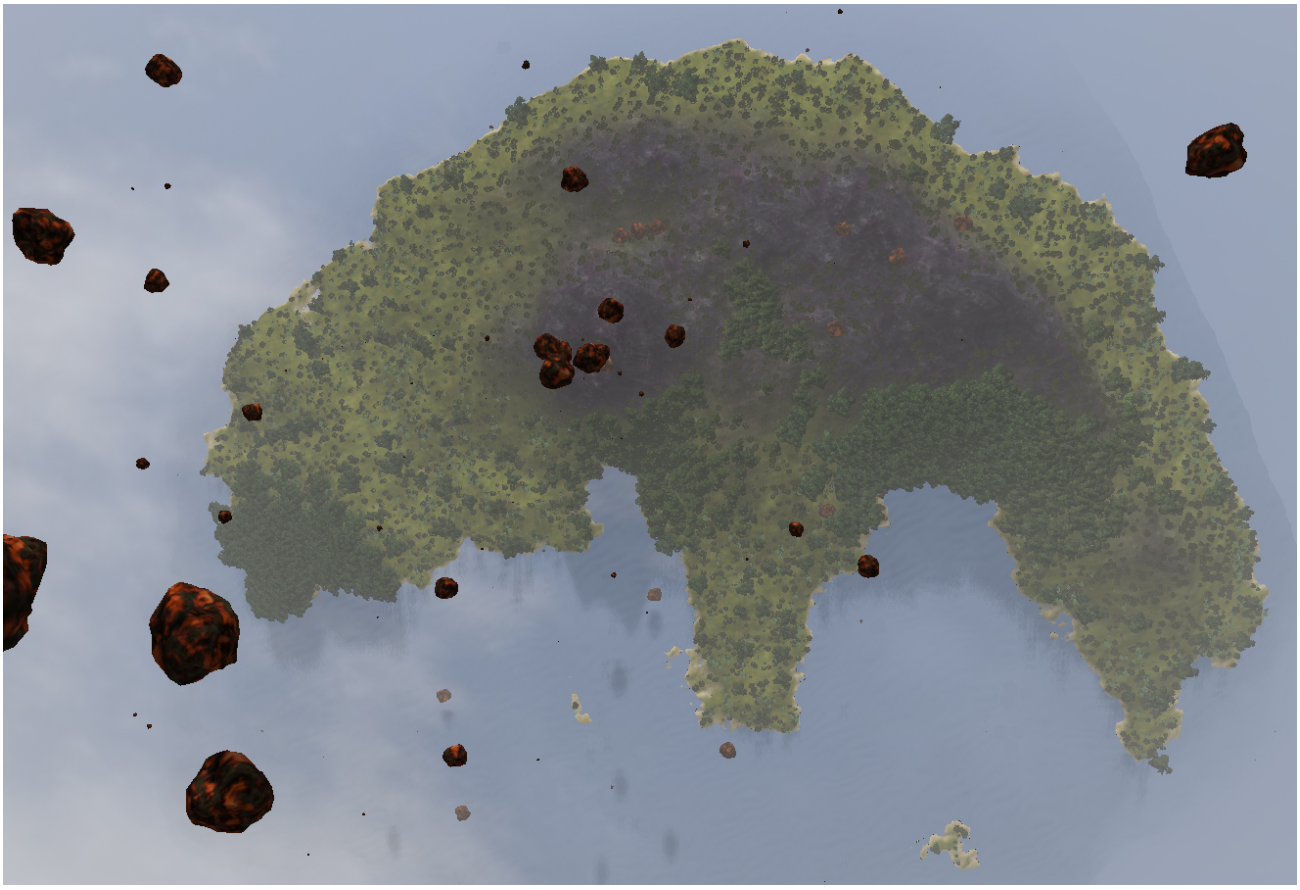


Figure 5.1: The final world from far above.

Possible improvements and further work are Level Of Detail (LOD) objects for the vegetation, so that we can have even more vegetation, especially on low-end machines. Tesselation for the terrain is another thing that would make our world look much better and still work on slower machines. Living water with real waves could be accomplished if we use tesselation on a mesh grid, something we cannot afford right now without losing too much performance. The collision detection would allow for many more objects if we use an occree to segment the world. This would also be useful for an AI in any game/simulation that want to sense the close environment. Finally we lack some inhabitants to make use of our beautiful world and to bring it to live.

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] Alec McEachran (2013)
"*Why Use Entity Systems for Game Engineering?*"
http://alecmce.com/library/why-use-entity-systems-for-game-engineering

[9] Boreal Games at gamedev.net (2013)
"*Understanding Component-Entity-Systems*"
http://www.reddit.com/r/gamedev/comments/1f83c5/

[10] Mike West (2007)
"*Evolve Your Hierarchy* "
http://cowboyprogramming.com/2007/01/05/evolve-your-heirachy/

[11] Adam Martin (2007)
"*Evolve Your Hierarchy* "
http://t-machine.org/index.php/2007/09/03/entity-systems-are-the-future-of-mmog-development-part-1/