Realistic Rendering Of Virtual Worlds

IN3405 BSc Project Report

Mattijs Driel - 1314440
Korijn van Golen - 1509152
Quintijn Hendrickx - 1509160
Zhi Kang Shao – 1509357

EEMCS Faculty

July, 2011

Exam committee

Initiator:	Ir. R.M. Smelik
Supervisor:	Dr.ir. A.R. Bidarra
Coordinator:	Drs. P.R. van Nieuwenhuizen

Preface

This document reports on the process and product of the BSc graduation project of four Computer Science students at the Delft University of Technology. The project is titled “Realistic Rendering of Virtual Worlds”, and was assigned to the authors by Ir. R.M. Smelik. The assignment was to create a reusable system that is able to render virtual worlds in a realistic manner, and to integrate this system into an existing software package called SketchaWorld.

At the start of the project our expectations were simple: a number of reusable shader programs had to be developed and integrated into SketchaWorld. The inputs for the shaders would have to be general, to make the shaders reusable. Things turned out to be quite different, as during the orientation phase we decided to apply a rendering approach called deferred rendering. In the end, the product has become a detachable deferred rendering system, specifically built for scene graphs.

We would like to thank Ir. R.M. Smelik for giving us the opportunity to take on this challenge and for the support he has offered us while integrating our system into SketchaWorld.

Mattijs Driel, Korijn van Golen, Quintijn Hendrickx and Zhi Kang Shao

Delft, July 2011

Table of Contents

1 Summary

The problem posed to the authors concerns SketchaWorld, a terrain modeling framework that allows users to easily construct virtual worlds by means of sketching. Until now, the developers of SketchaWorld have focused on procedural content generation and have left its visual presentation largely untouched. The authors have been tasked to develop a rendering system that improves the visual realism in SketchaWorld.

SketchaWorld organizes its world content in a scene graph, using the graphics toolkit OpenSceneGraph. A scene graph is a data structure in the form of an acyclic directed graph, for organizing objects in the world. The scene graph is both traversed to update the logic of all objects, as well as to render these objects. The rendering system would have to be compatible with this form of data structuring.

The authors had decided to implement a renderer that uses a deferred rendering approach. Contrary to forward rendering, which is the 'classic' approach, deferred rendering postpones many of the calculations that are required to determine the final output color for a pixel. Intermediate results are stored in texture buffers and combined in later phases. The deferred renderer is designed to make full use of the advantages of scene graphs; with the renderer itself is structured as a scene graph.

Many advanced techniques are implemented in the renderer, all aimed at enhancing visual realism. These techniques are spread out over the different phases in the deferred rendering pipeline. During the world render phase, primitive data about the world geometry is rendered. During the light render phase, each light is processed once to determine its influence on only the visible geometry. Afterwards, the results of both phases are combined. The atmosphere is rendered on top of that based on the time of day.

Transparency has always been an issue with deferred rendering. Transparency implies that the color value of one pixel depends on multiple objects, while most deferred renderers only store one set of data per pixel. This problem therefore needs to be solved by rendering transparent objects with forward rendering, adding it to the result of the deferred rendering steps. Afterwards, post processing effects are applied. The renderer supports an arbitrary number of post processing effects, which can be swapped in real time.

The authors have studied SketchaWorld and OpenSceneGraph thoroughly, in anticipation of integration at an early stage. As a result, integrating the renderer into SketchaWorld only cost minor effort. SketchaWorld's shaders have been converted to comply with deferred rendering requirements. All lighting calculations have been removed from the shader because the renderer performs these calculations in a later phase.

Most of the requirements that were formulated in the requirement analysis document have been implemented successfully. Some requirements were implemented somewhat differently than originally decided in the orientation report. These deviations were the result of new insights while developing the rendering system.

2 Introduction

The work presented in this document focuses on the design and development of a deferred rendering system, which combines advanced mapping and dynamic lighting techniques to achieve visual realism. The rendering system is built to be used with scene graphs, and is aimed at real-time applications.

The implemented rendering techniques are all based on existing work. Some techniques have minor setup modifications, to make them compatible with deferred rendering. Deferred rendering itself, despite being a relatively new rendering approach, has already shown successful application in mainstream games (Valient, 2007).

The main contribution of this work is to provide a detailed example of how deferred rendering can be implemented using scene graphs. The rendering system is built using an open source toolkit named OpenSceneGraph, but the discussed architecture and rendering techniques can be implemented in any scene graph environment.

This report is divided into ten sections:

- Section 3 (Problem definition and analysis) introduces the application SketchaWorld, and describes the previous rendering procedure's visual limitations.
- Section 4 (Environment) describes the environment in which the product was to be implemented. A summary of SketchaWorld's setup is given, as well as an introduction to the OpenSceneGraph open source toolkit.
- Section 5 (Design) describes the architecture behind the deferred renderer that the authors have built for SketchaWorld. The rendering process consists of multiple phases. Each of the phases is discussed in depth.
- Section 6 (Implementation) focuses on the implementation of the deferred renderer and the difficulties encountered therein. For each technique, the relevant part of the renderer's scene graph is presented, and the process of implementing the technique is described.
- Section 7 (Integration) describes the process of integrating the renderer into SketchaWorld, including the changes that have been made to SketchaWorld's existing shaders, and the steps that were taken to maintain support for all of SketchaWorld's existing content.
- Section 8 (Evaluation) presents an evaluation of the implemented renderer.
- Section 9 (Conclusions) summarizes the results gained during the development process of the deferred renderer.
- Section 10 (Recommendations) discusses improvements that the authors suggest for the techniques that have been implemented in the renderer, and suggestions for additional rendering techniques that can contribute to the visual quality in SketchaWorld.

3 Problem Analysis and Definition

The problem posed to the authors concerns SketchaWorld, a prototype implementation of a declarative terrain modeling framework. The main focus of the developers of SketchaWorld has been the procedural generation of a virtual world. Until now, *realistic* representation of the generated world has laid untouched.

As stated in the assignment document: “Improved visual quality would help user immersion and the level of realism that is perceived”. “Improved visual quality” is clearly the key to solving the problem.

The requirements of this assignment state that the system has to run at an acceptable frame rate on a DX10 GPU (e.g. NVidia GeForce 8800 GT). Therefore, a balance will have to be found between the visual quality enhancements and the resource consumption that the chosen enhancements entail.

According to the assignment document, the system should also be released as an open source component. This means that reusability, modularity and maintainability will be of high importance. Other programmers should be able to continue work on the project without much difficulty.

Finally, the product should be integrated into SketchaWorld, since this is the system being developed at the TU Delft that lacks the realistic rendering features we have to provide them with.

Incorporating all above statements, we define the problem statement as follows:

“The visual quality of SketchaWorld is lacking, and needs to be enhanced using an open source, modular, maintainable and reusable rendering component.”

4 Environment

4.1 SketchaWorld

One of the project requirements is that the rendering system must be compatible with SketchaWorld. SketchaWorld is a prototype implementation of a declarative terrain modeling framework. It allows users to easily sketch a world scene using a highly declarative approach (i.e. what do I want?) versus a constructive approach (i.e. how do I model it?). Additionally, the framework provides direct feedback in a 2D and 3D preview window. Figure 1 shows a screenshot of SketchaWorld's GUI.

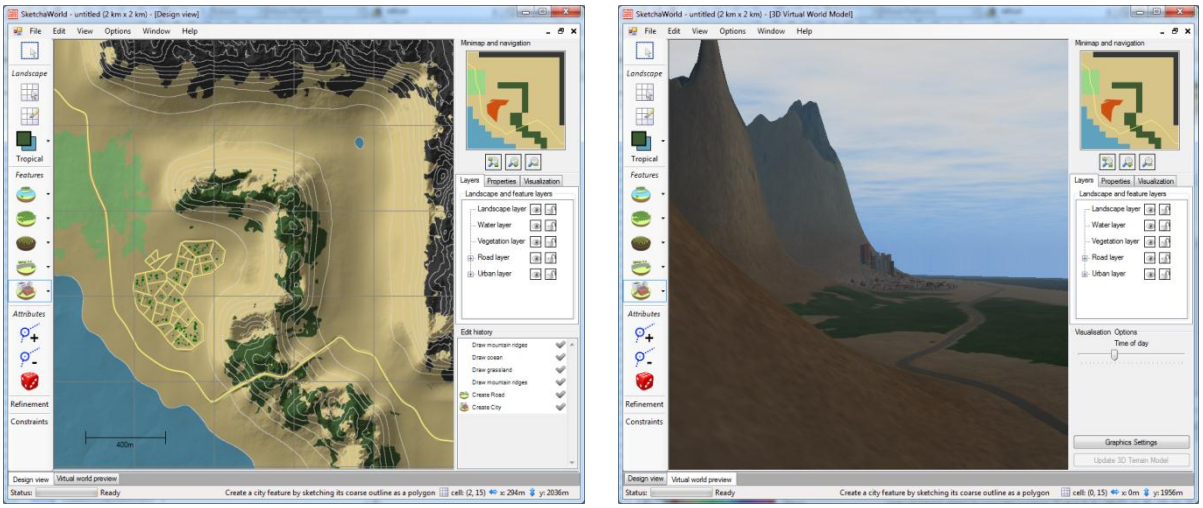


Figure 1. SketchaWorld's GUI.

Besides sketching the terrain outline it is possible to sketch roads, cities, rivers and many other terrain features. The content types are organized into different layers. In total SketchaWorld consists out of five layers: landscape, water, vegetation, road and urban. These layers are all able to interact and update each other. For example, a road that crosses a river will automatically generate a bridge structure.

The deferred rendering system will mostly interact with the 3D preview mode in SketchaWorld. Currently the 3D preview mode in SketchaWorld uses the OpenSceneGraph rendering framework. The following subsection discusses this framework in more detail.

4.2 OpenSceneGraph

In this subsection, the open source toolkit OpenSceneGraph is discussed. First, the general concept of scene graphs as a data structure is laid out and the primary node types in OSG are listed. Secondly, a global explanation is given of how OSG processes its scene graphs; by traversing the graph in specific ways. Finally, OSG's state set mechanism is discussed, which allows users to customize per node what the OpenGL rendering state should be when rendering that node. For the remainder of this section, 'user' refers to a programmer who makes use of an OSG scene graph to represent their graphical scene.

4.2.1 Scene Graphs

Scene graphs are data structures that represent virtual scenes. The definition of a scene graph is very general and many variations exist. In OSG, a scene graph is a collection of nodes, organized into an

acyclic directed graph. Nodes in the graph can fulfill a variety of functions. The main node types in OSG are:

- **Group nodes**: Group nodes are capable of having nodes attached to them as children. They do not by themselves serve additional functional tasks, save for organizing the scene graph.
- **Transformation nodes**: Transformation nodes are specializations of group nodes. They can define world transformations that are applied to all their children in the graph.
- **Geode**: Geode is short for geometry node. A geode represents a piece of 3D geometry, such as an imported 3D model. A transformation node is often placed above a geode, to define the 3D model's world position and orientation.
- **Camera nodes**: Cameras are nodes that render their attached sub graph. The camera node represents the position and perspective from where to render the sub graph, defined by the camera's view and projection matrix. The camera will render to a specified output buffer, which can be the screen or a render target.
- **LOD nodes**: Level of detail nodes can be used to select an appropriate level of detail model from a set of models representing the same entity. The selection is done based on the distance from the LOD node to the camera. For example, LOD nodes are employed in SketchaWorld for terrain: when the camera is farther away from a piece of terrain, a lower polygon model for that terrain is rendered.

Scene graphs allow for intuitive structuring of objects in the world. A simple example: consider a room, which has walls and contains a table. The room has a position and orientation in the world. To place the walls and the table without using a scene graph, one would have to take the room's position and orientation into account and add offsets to it.

Using a scene graph, one could define the same room as a hierarchy of transformations; this is demonstrated in Figure 2. The part of the graph that represents the room would have a transformation node as its root node. This transformation node defines the room's origin with respect to the world origin. Beneath this node, more transformation nodes can be placed to transform the walls and the table. The advantage is that the lower transformations are **relative** to the room's transformation. This allows for more intuitive placement of objects, and also makes it very easy to relocate the room as a whole.

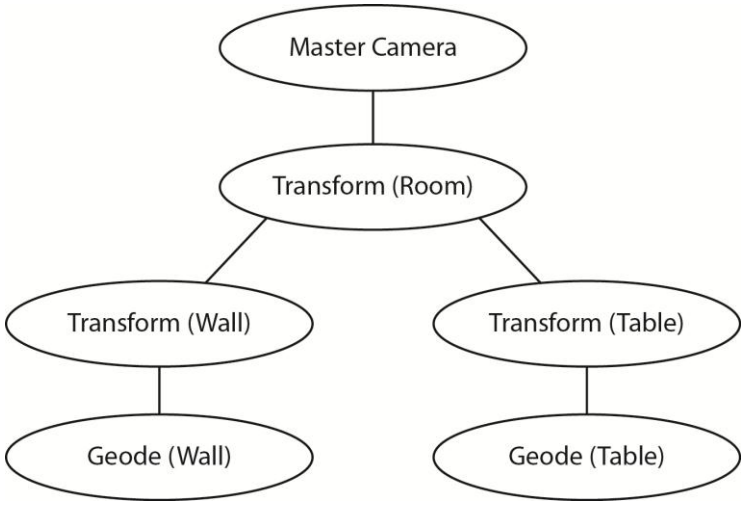


Figure 2. Scene graph representation of a simple room.

4.2.2 Scene Graph Traversal

Processing a scene graph, by manner of visiting the nodes, is called a traversal. In OSG, the scene graph is traversed multiple times each frame, each time with a different goal. The first two traversals are called the **event traversal** and **the update traversal.** The event traversal allows nodes to process events, such as user keyboard events. The update traversal allows nodes to update their state, based on the time that has elapsed since the last update. Both these traversals are application specific, and are less relevant to a renderer.

After the update traversal, the **rendering traversal** starts, this in itself consists of a **cull traversal** and a **draw traversal**. During the cull traversal, the **render graph** is constructed. This graph contains the nodes that should be rendered and determines the order in which they are rendered. It is during this traversal, for example, that a level of detail node decides which of its children to add to the render graph. Also, OSG includes functionality to filter nodes during this traversal, so that they are not visible to the rendering camera.

Cameras affect how the render graph is built. Often scene graphs will only contain one camera, that renders all geometry in one or more forward rendering passes. However, in scene graphs in OSG, the sub graph of a camera is allowed to contain other cameras. The child camera's **render order** then determines whether the child camera is drawn before or after its parent camera, depending on whether the render order is set to **pre render** or **post render**.

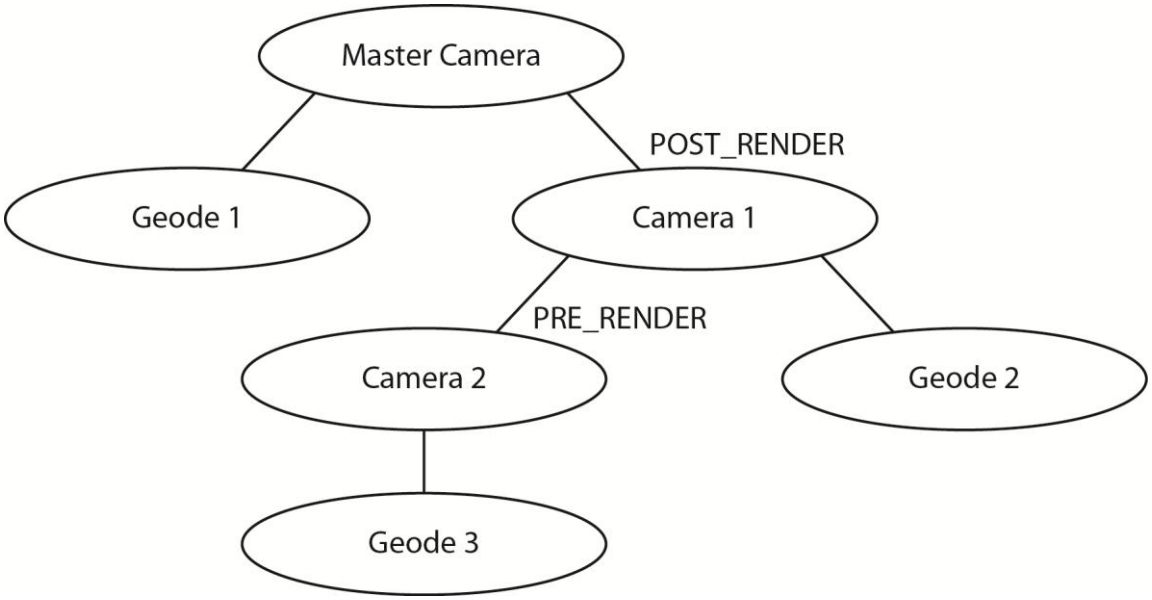


Figure 3. Example of a hierarchy of cameras in a scene graph.

It is important to realize that a camera's render order is always relative to its parent. Consider the scene graph displayed in Figure 3, which is a simple scene graph with multiple cameras. Camera 1 is a child of the master camera, and Camera 2 is a child of Camera 1. Camera 1 is set to post render, and Camera 2 is set to pre render. In what order will the cameras draw their geometry? In this case, the render graph would be constructed first, as follows:

- The master camera does not have any direct pre render cameras in the sub graph. Therefore all direct child geometry nodes are added to the master camera's render bin. Notice that only direct children are added (direct meaning that there is no other camera node separating

the geometry from the master camera). Afterwards, its post render child cameras are added to the render graph, in this case Camera 1.

- Camera 1's render bin is to be filled now. However, since Camera 2 is a direct pre render child camera, Camera 2 is added to the render graph first and its child nodes are processed first as well. Afterwards, since there are no more child nodes left, Camera 2's geometry is processed. The final render graph is displayed in Figure 4.

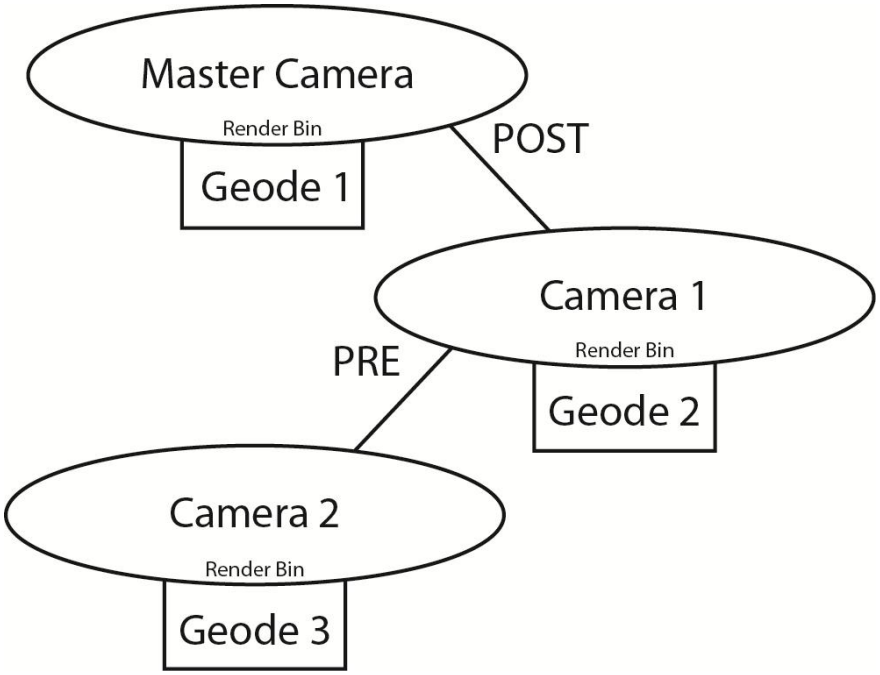


Figure 4. The resulting render graph associated with the scene graph displayed in Figure 3.

With the render graph now complete, the draw traversal can begin. This traversal simply performs a depth first search over the render graph, deciding what path to follow next using the following priorities: PRE > Render Bin > POST. This determines the final draw order, which would be as follows in our example:

- The draw traversal starts at the Master Camera. No PRE node is attached, so the Render Bin is processed and Geode 1 is drawn.
- The traversal then moves on to the node attached to the POST edge. Once there, it can tell there is a node attached to the PRE edge, so the traversal moves there first.
- Camera 2 has no PRE edge, so the Render Bin is processed and Geode 3 is drawn. There is no POST edge either, so the traversal returns to Camera 1.
- Since the PRE node has been processed, the Render Bin of Camera 1 is processed next and Geode 2 is drawn. No POST node is attached, so the traversal returns to the Master Camera, where the traversal ends.

Summarizing, cameras control when parts of the scene graph are to be rendered, relative to other parts. The final implementation of the deferred renderer exploits this at many locations in the scene graph. An example is the rendering of a shadow light, discussed in detail in Section 6.3.5.

4.2.3 State Sets

State sets are a mechanism provided by OpenSceneGraph to control how the OpenGL rendering state should change for different parts of the scene graph. In OSG, every node can have a custom state set. The state set defines which OpenGL calls should be made before rendering the node and its sub graph. Examples of such OpenGL calls are glEnable/glDisable, glBindTexture and glUseProgram.

A state set is only created for a node when the user decides to give the node a custom value for a certain state attribute. Even when a state set is created, any state attributes that do not get a custom value assigned, inherit their values from the parent of the node.

Examples of state attributes are:

- which OpenGL modes should be enabled, such as depth testing, blending and lighting;
- the parameters for these modes, such as the blend function;
- which texture should be bound;
- which shader program should be active.

4.2.3.1 Processing of State Sets

State sets are applied during the rendering traversal in the following manner:

1. When starting the rendering traversal of the scene graph, the root node of the scene defines the initial OpenGL rendering state. OpenGL calls are made to initialize that rendering state. Then, the rendering traversal continues through the graph.
2. When a node in the sub graph is to be drawn, if it has any custom state attribute values, corresponding OpenGL calls are made to make changes to the rendering state. The previous values are pushed onto a stack and the rendering traversal continues to the sub graph.
3. When the node's sub graph has finished drawing and control returns to the node itself, the previous values of the changed attributes are popped from the stack and used to restore the rendering state to what it was before rendering the node.

4.2.3.2 Optimization Using State Sets

One can use the knowledge that "a parent node's state attribute is active until specified otherwise", to organize scene graphs in such a way that overhead in OpenGL calls is minimized. For example, when adding multiple spot lights to the scene, all these lights are intended to be rendered with the same shader program. Rather than linking that shader program to the state set of each individual light, it is more efficient to link the shader program to a separate node, and add all lights as children to that node. This prevents unnecessary rebinding of shader programs, when spot lights are rendered directly after each other. This is an optimization that can also be applied without the use of scene graphs, but scene graphs allow for intuitive organizing and, most importantly, fast refactoring.

More examples of efficient use of state sets can be found in the next section, in which the design of the deferred renderer is discussed.

5 Design

This section focuses on the final design of the rendering system. Based on the problem analysis, one of the requirements that the system needs to take advantage of is an approach to rendering called deferred rendering.

This section will begin with a short introduction to deferred rendering, and then continue to specify the design of the implemented renderer in a scene graph.

5.1 Deferred Renderer

5.1.1 An Introduction to Deferred Rendering

In contrast with the 'usual' approach to rendering, called 'forward rendering', deferred rendering postpones a lot of the calculations that are required to determine the final output color for a pixel. Instead of applying all lighting calculations directly to a piece of geometry or a fragment defined in some other manner, all the data that is required for these lighting calculations is stored temporarily.

This means that for every pixel, all lighting calculations have to be applied only once (without requiring access to the geometry), rather than once for every piece of geometry inside a pixel (a disadvantage of forward rendering called 'overdraw'). The data storage mentioned is called the G-Buffer.

After all the lighting calculations have been performed, the results of these calculations are combined with the diffuse color of all on screen geometry (stored in the G-Buffer) and the result is the final image that is output to the screen.

A major disadvantage of Deferred Rendering is its inability to handle transparent objects. Since every pixel in the G-Buffers can only contain data for 1 piece of geometry, it cannot support multiple 'layers' of transparent objects. There are solutions to this problem such as Depth Peeling, but the hardware that is currently available is not nearly powerful enough to support this real-time. Considering the hardware requirements imposed upon this project, Depth Peeling is not an option. Another solution is to use multiple G-Buffers, one for each layer of geometry encountered. However, the memory requirement this imposes is not supported by today's hardware either.

5.1.2 Choice of G-Buffer

The G-Buffer in deferred rendering is a collection of screen-sized textures, to which information of the scene geometry is rendered in screen space. The choice of contents of the G-Buffer defines the capabilities of the deferred renderer, such as HDR and specularity. In the current implementation, the G-Buffer consists of a number of fullscreen render targets as shown below.

Texture name	Internal format	Contents			
		Red channel	Green channel	Blue channel	Alpha channel
DiffuseSpecular	8-bit Color per channel	Diffuse red	Diffuse green	Diffuse Blue	Specular reflectiveness
Normal	8-bit Color per channel	Normal X	Normal Y	Normal Z	
Light	16-bit Float per channel (HDR)	Light red	Light green	Light Blue	Specular intensity
Depth	32-bit Float single channel	Depth buffer			

Figure 5: G-Buffer content specification.

As with all deferred rendering implementations, a diffuse, normal, light and depth buffer is present. The normal and depth buffers are mainly used in lighting calculations. In the composition phase, the diffuse buffer is multiplied with the result of the light buffer to result in a lit environment.

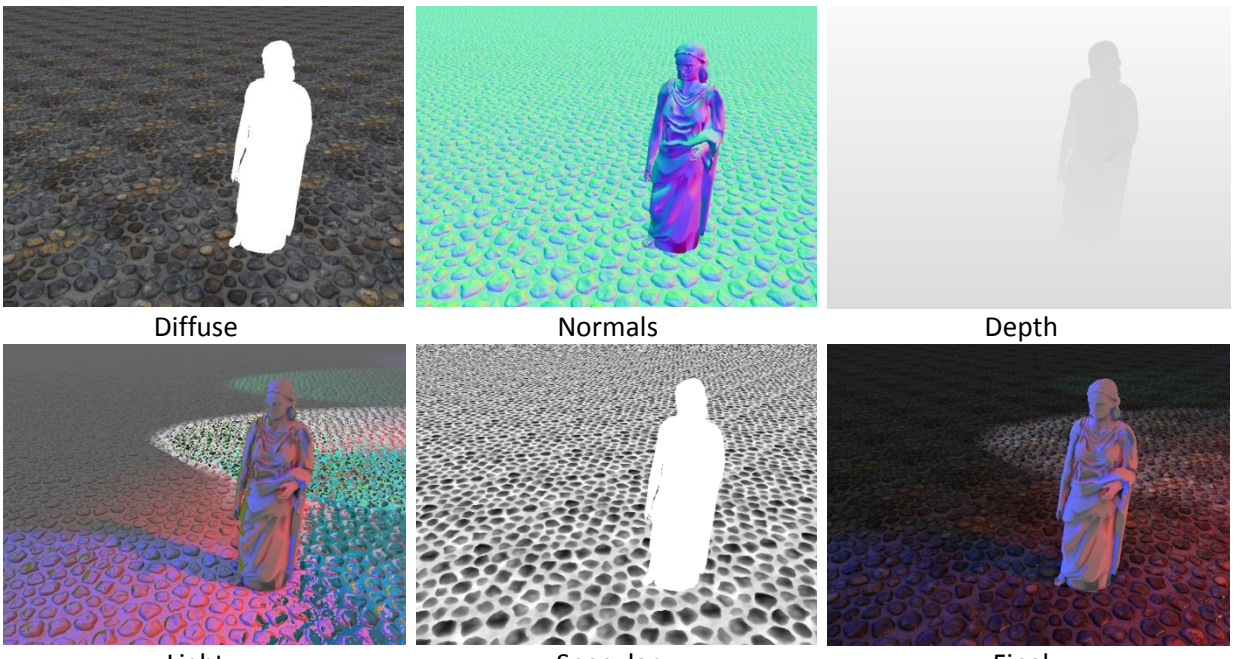


Figure 6. G-buffer contents.

A few choices made deviate slightly from common implementations of deferred renderers, namely the increased capacity of the light buffer and the handling of specular effects.

The reason for the increased capacity of the light textures originates from the intended capability of producing High Dynamic Range images (HDR). The 16-bit float storage allows for a higher range of light values. This results in the composited image being 16-bit as well, which later needs to be tone mapped to normal Low Dynamic Range (LDR) for proper display.

The alpha channel in the diffuse specular image stores the reflectivity of the surfaces in the scene. When rendering the lights, the specular value stored in the scene is used to calculate a level of reflection, the result of which is stored in the alpha channel of the light buffer. This does however come with a limitation. With the specular intensity currently being stored in a single channel, the color of the light reflecting of the surface is restricted to the same value for all lights. Regardless, the specular effect is used as the available alpha channels would remain unused otherwise.

5.1.3 Architecture

The deferred rendering solution presented here consists of several render phases. Figure 7 displays a high level scene graph, that reflects the current implementation of the deferred renderer. Every phase has a well-defined purpose within the deferred rendering process, and each phase is represented by a part of the scene graph.

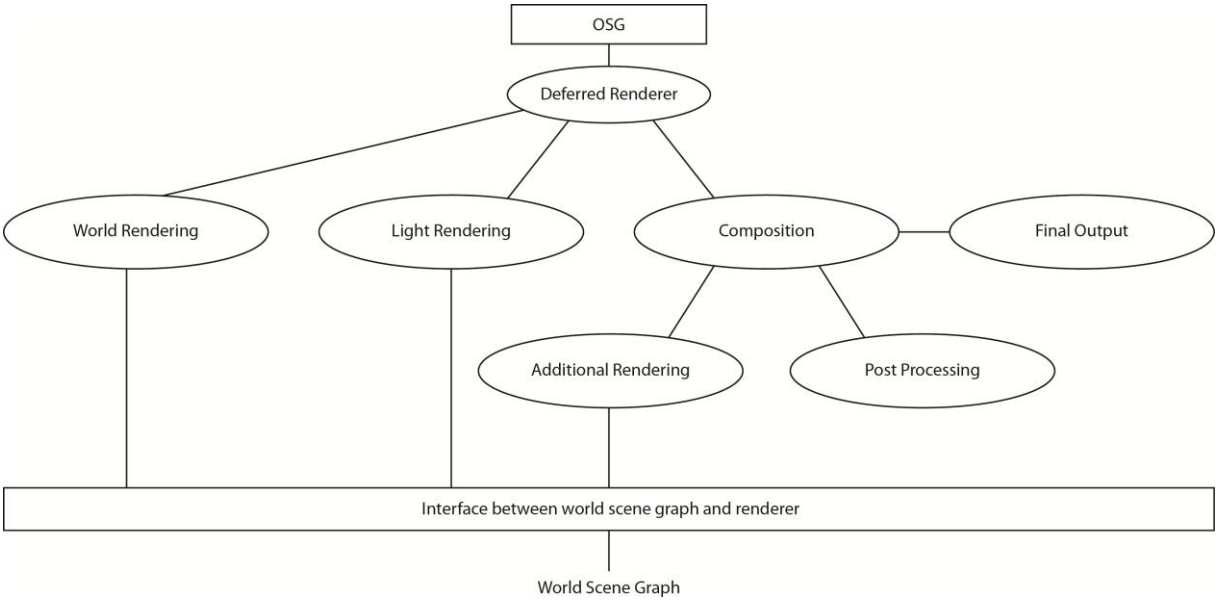


Figure 7. A simple view of the Deferred Renderer and its components.

The first phase renders the world scene and fills the G-buffer with information about the diffuse, specular, normals and glow components of objects in the world. Next, the light phase is executed and the light render target is filled with information about the effects that the lights have on the scene. The combine phase combines the G-buffer data with the light render target and constructs a resulting image of the scene. In the additional rendering phase, transparent objects and additional scene elements such as the sky and atmosphere are added by a forward renderer. Afterwards post processing effects are applied. Then, the last step is executed by the final camera; this node reads the output from the combine camera and writes it to the back buffer of the current graphics context.

5.1.3.1 World Render Phase

In the first phase of the deferred renderer, all opaque scene geometry is rendered to the G-Buffer (transparent geometry is handled later). The current implementation allows a total of 5 texture maps to be used by the scene geometry, which will affect how the G-Buffer is filled. The possible textures are explained as follows:

- Diffuse (or albedo) – This defines the color of the surface disregarding lights. The information for the diffuse would commonly be available as a 3-channel RGB texture.

- Normals – This defines offsets for the default vertex normals of the geometry, allowing more detail when lighting is present. The information for the normals is similarly stored as a 3-channel RGB texture, but with each color representing an axis by which the normal is offset. The G-Buffer stores this information in view space.
- Specular – This defines the amount of reflection from lights the surface allows. The information for the specular reflectivity is a single channel texture.
- Height – This defines a degree of parallax offsetting, offsetting the sampling coordinates at which to sample all other textures. This effect requires a single channel as a height map. In contrast to the other textures, this effect is not stored in the G-Buffer; instead it affects the way other information is stored in the G-Buffer.
- Glow – This defines the self-illumination of the surface. A single channel texture is used, that writes directly into the light texture of the G-Buffer. Rendering to the light texture makes the surface visible regardless of other lights in the scene.

After the rendering of the opaque geometry, the diffuse, specular and normal buffers of the G-Buffer are filled. The light buffer will contain glow information, to which actual light information will be added in the next phase of rendering.

5.1.3.2 Light Render Phase

The purpose of the second phase of rendering is filling the light buffer with all lighting information in the scene. The current implementation uses light classes other than those found in OSG, as deferred rendering allows a larger amount of lights in a scene than forward rendering would support. This decision does make it harder to integrate new lights in the existing OSG solution. This is especially true when optimizations concerning deferred rendering (explained further in this subchapter) wouldn't otherwise be possible. Because of this, the current implementation uses a custom deferred light class to represent lights that can influence the scene, all of which need to be manually added to the renderer.

5.1.3.2.1 Basic Lights

Depending on the type of light, a specific set of information is required that defines how the light will affect the scene. All available lights have a color and base intensity. In addition, a few extra inputs are needed for each type of light:

- Point light – This type of light requires a point in the world defining the origin of the light, and a radius for the maximum range of influence.
- Spot light – This type requires not only a radius and a point in the world defining its origin, it also requires a point in the world defining its target. Also, a radial drop-off value is used to imitate the gradient effect that is associated with spot lights.
- Directional light – This type of light simply has a direction stored as a unit vector.
- Ambient light – This light type uniformly lights the scene, and thus needs no additional information.

A degree of efficiency is gained by taking advantage of the limited area of influence some lights have. Point lights and spot lights do not always fill the entire screen when rendered, so it is beneficial to use geometry other than a full screen quad to render the light's influence. A world space sphere and cone can thus be used for point and spot lights respectively. This is especially useful for scenes with a lot of lights occupying just a small portion of the screen.

Each light the user adds to the renderer is rendered in sequence to the same render target, the light buffer. Additive blending is used to maintain information from previously rendered lights (this includes glow effects from the world render phase). Each light will add its contribution of color to the light buffer, which is calculated using the various light properties mentioned before.

5.1.3.2.2 Shadow Lights

All lights up to this point have the limitation of not taking objects from the scene into account, which might block a light from reaching some areas in the scene. For this purpose, the current implementation supports a spot light and a directional light that can cope with occluders. Both of these lights are externally similar to their basic counterparts (no extra inputs are needed for the user), but are noticeably more expensive because of the use of shadow mapping.

Shadow mapping is accomplished through rendering the scene from the viewpoint of the light. In this case, only the depth buffer is written to a shadow render target. With the shadow map available, the light can be rendered using a similar process as the basic lights. The only addition is that a shadow factor is calculated for each pixel. This factor is calculated by checking how visible the point is relative to the light.

As each light is rendered in sequence, all lights can use the same shadow render target which limits texture memory usage, improving efficiency.

5.1.3.2.3 Ambient Occlusion

If implemented correctly, ambient occlusion can significantly increase perception of depth and detail. Compared to previously discussed lights, ambient occlusion does not have a direct light source. Instead the light intensity is calculated based on occlusion of indirect light. To calculate this occlusion factor correctly, it is necessary to consider all geometry around the shaded pixel.

Screen space ambient occlusion is a specific implementation that uses information from the depth buffer to approximate an ambient occlusion factor. By reconstructing world space position from the depth buffer it is possible to sample multiple pixels around the shaded pixel and calculate at which angles the pixel is occluded.

Several problems and limitations exist for the screen space ambient occlusion method. To get accurate results a lot of samples are required. To reduce samples it is possible to use a randomly rotated kernel basis, however this can introduce noise in the final image. If needed, this noise can be reduced by blurring the occlusion factor.

Other problems include view dependency and frustum edges, these problems and their solutions are discussed extensively in Section 6.3.4.1.

5.1.3.3 Combine Phase

After the initial world render phase the G-Buffer is filled with information about the diffuse, specular, normal and light components. The combine phase uses these components to construct the final shaded image.

The first step is combining the diffuse, specular and light render targets and rendering the result into a HDR buffer.

After this step is done, a forward renderer is used to render additional elements such as transparent geometry and the atmosphere to this HDR buffer.

As a last step we apply all post processing effects added by the user using a buffer swapping technique called ping-ponging that minimizes memory usage. This technique is discussed in Section 6.5.2.

5.2 Interface with SketchaWorld

One of the design goals for the deferred rendering system was that it could be easily integrated into different rendering applications. In particular the rendering system had to be applied to the SketchaWorld modeling framework. To facilitate the integration of the deferred rendering system into existing applications the aim was to create a very simple and transparent interface.

The interface with external applications can be divided into several different categories: setting the world node, adding and removing of lights, configuration settings and post processing effects.

First there is a method that allows the external application to set the world node. Essentially this provides the renderer with all information about the scene geometry. A different set of classes facilitate the adding and removal of light sources to the renderer.

Although not required, it is possible to tweak the rendering system using several configuration settings. To do this the system provides a simple interface that allows external applications to set or get the configuration values. Settings are flagged to be dirty when they are changed so that the application can be notified of changes in the configuration.

Finally, there is an option to add or remove an arbitrary amount of post processing effect to the rendering system. It is possible to use any of the already provided effects or implement your own effect by inheriting from a base post processing camera. Setting the render order property on a post processing effect defines the order in which they are applied to the screen, regardless of the order in which they are added to the deferred renderer.

6 Implementation

The following chapter contains all implementation specific details on all components of the Deferred Renderer, presented in the order in which they are run during a scene graph traversal. The distinction was made between a number of phases in order to emphasize the borders between branches of the scene graph.

6.1 Initialization

Before the phases are laid out in depth, some vital background information is presented. The following components are parts of the deferred renderer that are set up during initialization of the system.

6.1.1 Synchronization Callbacks

There are many cameras in the deferred renderer that need to be synchronized in terms of view and projection matrices. To be exact, the world, light and transparency cameras all need to be in sync. User input is delegated to the deferred renderer through an OSG camera manipulator, which transforms the viewing perspective. The deferred renderer registers a callback on this manipulator to synchronize all relevant cameras.

The callback on the camera manipulator only synchronizes the viewing matrix though, so a different mechanic is needed to keep the projection matrix synchronized over all cameras. In each cull traversal, OSG calculates new near and far view planes for the projection matrix. It's important to keep this information synchronized, as it is needed for reconstructing a world space or view space position from the depth buffer. To solve this issue, a callback is added to the cull traversal of the world camera, being the first camera to calculate it's near and far view planes. In this callback, various cameras and uniforms are updated with the correct projection retrieved from the world camera.

6.1.2 Full Screen Quads

The deferred renderer almost constantly makes use of full screen quads, which is nothing more than a geometrical shape consisting out of two triangles forming a rectangular area. They are defined in screen space, generally having their min and max corners positioned at (-1, -1) and (1, 1).

They are especially useful in the deferred renderer since they enable a camera to pass over every fragment of a screen sized texture, using the least possible amount of vertices. For example, post processing effects such as a blur consist of nothing more than a camera and a full screen quad geode.

6.2 World Render Phase

The first phase the Deferred Renderer enters is the World Render Phase. The World Camera renders all visible opaque objects to the G-Buffer.

6.2.1 Filling the G-Buffer

When drawing the world scene that is provided by the external application to the deferred rendering the g-buffer is filled with the diffuse, specular, normals and glow data. Traditionally rendering output has always been in a 3 byte RGB color format. However, with current hardware it is possible to take advantage of multiple render targets and fill the entire g-buffer with a single draw call.

This means that if specialized shaders are provided for nodes in the world scene these shader have to write their output to the appropriate render targets. The integration guide provides more information about how to implement specialized shaders.

A special note should be addressed to the normal data in the g-buffer. The first implementation of the deferred renderer stored normal vectors in world space format. However, as it turned out this was not the most optimal solution when performing lighting calculations in the Light Render Phase (Section 6.3). Instead, to simplify operations for parallax mapping and lighting calculations the normals should be provided in view space format.

6.2.1.1 Applying Advanced Mapping Techniques

The deferred rendering system provides implementations for several known mapping techniques. In order of increasing complexity these are: normal mapping, parallax mapping (Kaneko & Takahei, 2001) (McGuire, 2005) and parallax occlusion mapping (Policarpo & Oliveira, 2006).

Parallax mapping is a mapping technique that distorts the texture coordinates based on the current view angle. Because this distortion is based on a single extra texture lookup in a height map it is a very efficient technique that gives convincing results. Because parallax mapping distorts the texture coordinates as a continuous function of the height map it isn't possible to create an occlusion effect. However, parallax occlusion mapping does allow objects to occlude one another, creating a significantly more realistic effect.

Contrary to the observations in the orientation report it was possible to create an efficient implementation of the parallax occlusion mapping algorithm. By applying a binary search it is possible to quickly and efficiently converge to the closest intersection point in the height map. The problem with this approach is that it might find a local minimum that might not necessarily be the global minimum. This problem can be solved by first executing a coarse linear search followed by several binary iterations.

Because the parallax occlusion effect is only visible when viewing objects up close and under shallow angles we have also provided a hybrid mapping mode. This mode automatically blends objects that are further away using simple parallax mapping while using high quality parallax occlusion mapping for objects that are near to the camera.

Figure 8. Simple parallax mapping (left) vs. parallax occlusion mapping (right).

6.3 Light Render Phase

After rendering all geometry data to the G-Buffer, the Deferred Renderer iterates over all the different lights that have been added to it, outputting their light contributions to a separate texture defined in the Deferred Renderer called the Light Texture.

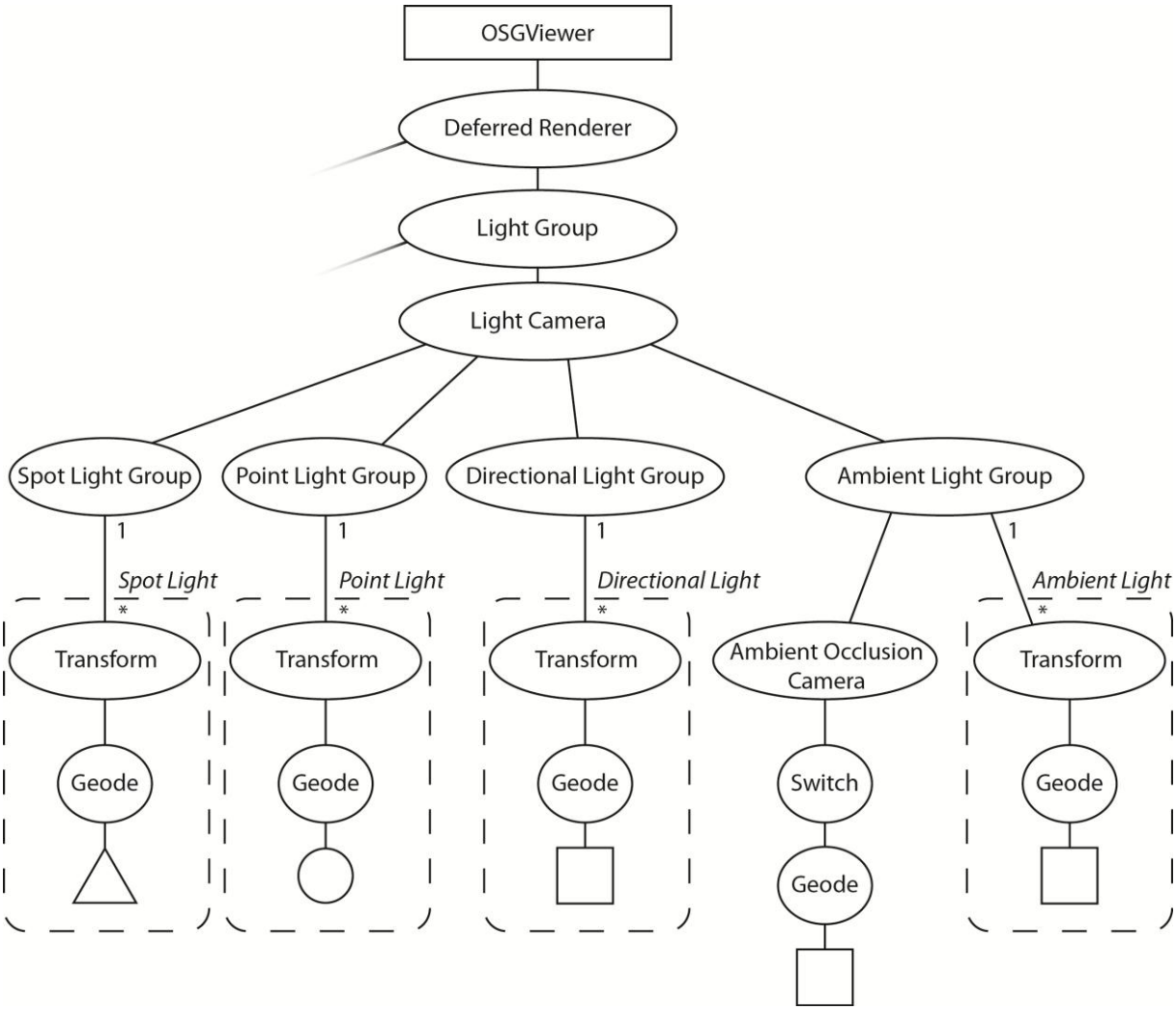


Figure 9. The scene graph for all non-shadow casting lights.

All lights are based on a generic type of light, called a Deferred Light. Some behavior of the lights rendered by the deferred renderer is shared. A summary of this shared behavior:

1. When a light is added to the deferred renderer, it is attached in the scene graph in the light group sub graph. Each light type has a group node that utilizes a specific shader program for that group. For example, all directional lights can use the same shader, it's the color and direction that will make the difference between them. This scene graph organization is beneficial as each individual light is kept compact.
2. The influence a light has on the scene can vary depending on the orientation of the source, in relation to the camera. For some types of lights, this property is exploited in the current implementation through rendering a world space geometry that limits the pixels that need rendering.

3. For each pixel a light needs to render, the view space coordinate of the pixel is required. Since the G-Buffer stores normals in view space and not in world space, the view space position of the pixel is desired for most basic light calculations. Sampling the depth buffer at the pixel coordinate restores the z-value of the pixel which can be used to transform from screen space to view space. An important detail is that, while each light's position and direction is initialized in world space, the shader program will need this location in view space. This transformation is done before the light is rendered, during the pre-draw callback on the light.
4. The specular factor for each light is calculated in the same way. A reflection vector is calculated by reflecting the light off the surface normal. The relative angle of this reflection with the camera viewing the scene defines a level of specularity to the surface.
5. Every light affects only the surfaces stored in the G-Buffer. This implies that all lights are themselves not visible to the viewer, only the surfaces the light illuminates. If a user would want to make the light source visible, other methods are necessary. For example a flashlight. The head of the flashlight could be given a glow texture, so that surface will always appear lit. Combined with a spot light positioned at the head, a convincing effect can be achieved.

Note that the 3rd and 4th point described above, do not apply to ambient lights as they light the scene uniformly. Ambient lights do have the optional capability of ambient occlusion, making its lighting contribution more appealing.

In the following subchapters, more specific behavior for each light type is explained.

6.3.1 Spot Lights

A spot light is initialized with a world space origin, a world space target, a radius, and a radial drop-off. These values are transformed into the uniforms needed by the spotlight shader program. All spotlights use a cone geometry transformed to the correct world space position based on origin, target and radius. This is done through building a matrix which can transform a unit cone to the desired dimensions (unit cone defined as base radius = 1 and height = 1).

In the shader program, an attenuation factor is calculated, a scalar by which the light's intensity is reduced. This is dependent on both the distance to the light's origin, and the radial drop-off (the closer to the edge of the light's cone, the lower the intensity). The attenuation is currently calculated linearly for simplicity's sake, but could be replaced with quadratic attenuation if desired.

6.3.2 Point Lights

A point light is initialized with a world space origin and a radius. These values are passed on to the uniforms for the pointlight shader program. Point lights use a sphere geometry, which is transformed similarly to how the spotlight's cone is transformed. Attenuation is calculated based solely from the distance to the point light's origin; it is otherwise the same as the spotlight's implementation.

6.3.3 Directional Lights

Directional lights are defined only by their direction. Because a directional light is usually visible throughout the entire scene, a full screen quad is used to get all pixels rendered. Furthermore, no attenuation is calculated, so the resulting intensity is simply based on the relative angle of the light's direction with the surface normal and the light's base intensity.

6.3.4 Ambient Light

As opposed to the previously discussed lights, simple ambient lighting does not apply shading and brightens all objects within a scene equally. This type of light can be used to approximate indirect lighting and ensures that all objects in the scene are at least somewhat visible.

Because ambient lighting is a global effect, it is rendered using a full screen quad. All pixels in the scene are influenced equally.

6.3.4.1 Screen-space Ambient Occlusion

Ambient light can be approximated more accurately by calculating an ambient occlusion factor. Ambient occlusion adds shading that is particularly helpful when perceiving depth differences in objects with fine details. If this effect is enabled, we calculate the ambient occlusion prior to processing any of the ambient lights. The ambient occlusion factor is calculated in screen-space (Mittring, 2007) (Bavoil & Sainz, 2008) and stored in a separate render target. Because we use a screen-space approximation for ambient occlusion, the added complexity is independent of the scene's geometry.

The effect of the current approximation of ambient occlusion in screen space is illustrated in Figure 10.

Figure 10. No ambient occlusion (left) vs. screen-space ambient occlusion (right).

During the implementation of SSAO, several problems were faced. One of the problems inherent with SSAO is that it is view dependent. This means that the total occlusion factor at a specific position depends on the camera position and angle. This is because geometry in the scene that occludes pixels might be occluded itself. Additionally ambient occlusion for pixels at the edge of view frustum can't be determined accurately because there is no information about the pixels that are not inside the view frustum. By setting the texture wrap mode to `BORDER` and choosing a suitable border color the artifacts around the edges became less noticeable. A possibility to decrease artifacts near the frustum edges even further is to decrease the sampling radius. However, decreasing the sampling radius will also reduce the quality of the result.

A different problem apparent in most SSAO implementations is self-occlusion. This effect causes ambient occlusion to occur on flat surfaces due to precision issues. At screen-space boundaries between flat surfaces and other geometry this can cause haloing artifacts. The self-occlusion that is present on the flat surface disappears causing the area to appear brighter. By introducing a bias

parameter the self-occlusion effect can be eliminated almost entirely, this also considerably reduces haloing problems.

6.3.5 Shadow Spot Lights

Shadow spot lights are an extension of spot lights, that take into account that light might be blocked from reaching a point, because there might be an occluder in between. The occluder will cause a shadow to be cast on anything that is behind it from the light's perspective. Shadow spot lights have been implemented with the idea that they can be used for simulating street lights in SketchaWorld.

The light calculations for a shadow spot light are very similar to the calculations of simple spot lights. In the shaders, there is only one difference: in the fragment shader a shadow factor is calculated, that defines how much of the light actually reaches the pixel. The shadow factor is used to scale the light output of the original calculations. To calculate the shadow factor, an advanced shadow mapping technique called Percentage Closer Soft Shadows is applied.

6.3.5.1 Shadow Mapping

Shadow mapping is a method for determining whether light can reach a certain position in the world or not. The method is compatible with both forward rendering as well as deferred rendering.

For simplicity, assume that the light is a spotlight, and thus has a direction and a limited area of influence. First, the scene is rendered from the perspective of the light source. The depth buffer is filled and stored in a texture called the shadow map. This shadow map is sampled from during the rendering of geometry. While rendering geometry, in the fragment shader the pixel's position is transformed to the post-projection space of the light source.

Now, the transformed point can be compared to a corresponding value in the shadow map, since both are coordinates in the light source's post-projection space. If the distance between the pixel and the light source is larger than the distance stored in the shadow map, it means the pixel is not visible to the light source. As a result, the pixel does not receive the light's full intensity.

Using only one sample to determine whether a pixel is lit or not would result in hard shadows: pixels are fully lit or completely unlit. Also, due to the shadow map's limited resolution, borders of shadowed areas would appear to be blocky. More advanced sampling techniques have been developed to make shadows look more realistic, one of which is PCSS.

6.3.5.2 Percentage Closer Soft Shadows

PCSS takes multiple samples from the shadow map, to estimate the distance between the occluder and the shadowed surface. Also, the light source has a specified size. These values are used to compute the penumbra. Penumbra refers to the soft contour of a shadow, that we see in daily life because light sources have a surface, rather than being a single point. The technique is explained in detail in the orientation report and in (Kevin Myers, 2008).

6.3.5.3 Scene Graph Setup

The data in a shadow map is only necessary for the calculating of one light. It would be ideal if one texture could be reused between lights, to temporarily store the shadow map in. The implemented scene graph structure of one shadow spot light is based on this property. The scene graph is displayed in Figure 11.

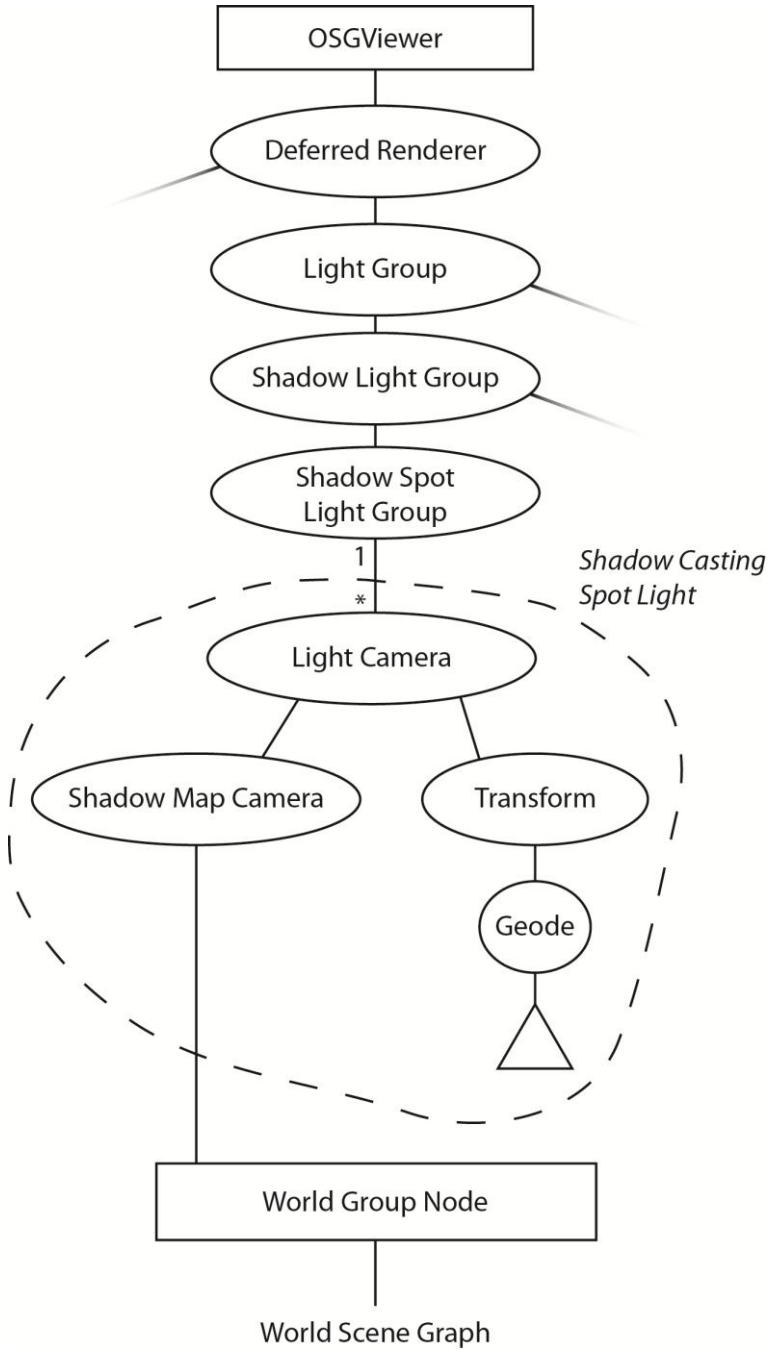


Figure 11. Scene graph representing one shadow spot light.

At the root of the graph is the light camera. The shadow map camera is added as a child to the light camera, and the render order is set to pre render. On the right side, the spotlight's cone shaped geometry is added as a child of the light camera. The world is also added to the shadow map camera, as this is its input for generating the shadow map. The shadow map is stored in the reusable shadow map texture.

This setup ensures, when the draw graph is built during the cull traversal, that the light cone is rendered directly after the shadow map is generated. Afterwards, the data in the shadow map may be overwritten by a potential next shadow spot light. The reusing of the shadow map texture is an option made available because of deferred rendering, since every light is rendered independently.

6.3.5.4 Process

The shadow's penumbra is realized by dynamically determining a filter radius in the shadow spotlight's fragment shader. The filter radius is finally used in a PCF filtering step. To find this radius, one of the steps includes a constant value named the **near plane** (not to be confused with the near plane of a projection)**.** (Fernando, 2005)

During the initial implementation of PCSS, it became apparent how much this assumed near plane affects the shadow calculations. The near plane and the light size are used to perform a similar triangles calculations. This calculation is used to find the penumbra size. The penumbra size is proportional to the final filter size.

Figure 12 shows how the near plane affects the calculation of the penumbra size. A lower value for the near plane, meaning closer to the light source (right), causes the penumbra size to increase. This results in softer shadows. However, since the filter radius is proportional to the penumbra size, when not careful the filter radius will become too large. PCF filtering from a shadow map with too large filters is known to show disastrous results, as averaging the samples will not give a proper estimate of the occluder's distance.

In the current implementation of PCSS in the deferred renderer, the near plane is tweaked specifically to street lights in SketchaWorld. That is, PCSS works best at distances of 1 to 4 world units, which in SketchaWorld corresponds to 1 to 4 metres.

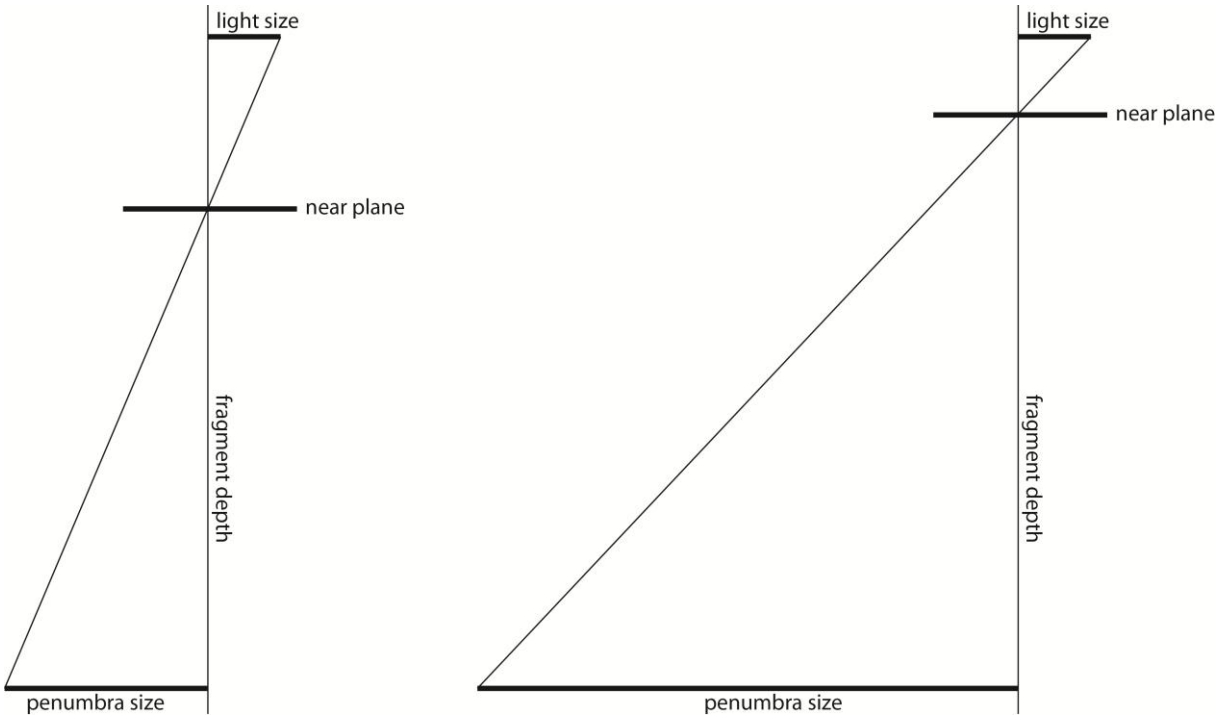


Figure 12. The effect of the near plane constant on estimating the penumbra size.

6.3.6 Shadow Directional Lights

In this subsection, the process of implementing cascaded shadow maps for the deferred renderer is discussed. First, the concept behind cascaded shadow maps is summarized from the Orientation Report. Secondly, the usage of OSG nodes to implement the technique is explained. Lastly, the challenges that were met when integrating the technique into SketchaWorld are discussed.

6.3.6.1 Cascaded Shadow Mapping

Cascaded shadow mapping is a scene shadowing technique, intended for displaying shadows in larger, outdoor areas. This technique is often used to make the objects cast shadows with the sun as light source. (Dimitrov, 2007)

For larger areas, using a single shadow map is problematic: a texture of enormous resolution is required to produce visually acceptable shadows. If that resolution is not met, the user will see blocky edges, even when applying a smoothing technique such as PCF filtering. This effect can be seen in Figure 13.

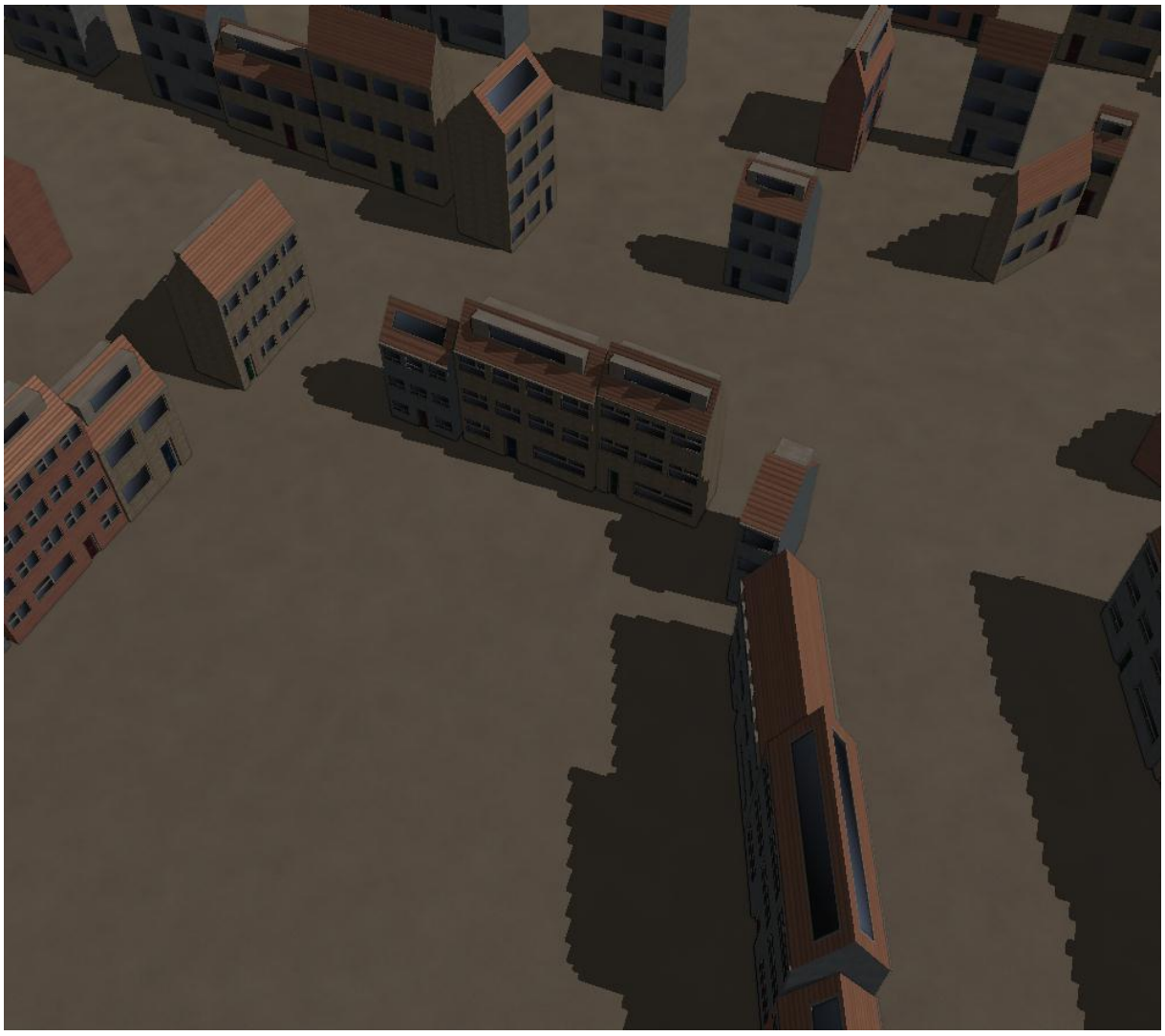


Figure 13. Low resolution shadow maps often produce blocky shadows.

Cascaded shadow mapping is based on the observation that an area farther away from the user requires less precise shadows than an area close to the user. It would be wasteful to generate one shadow map over the entire visible area, with the precision that is only needed in the area close to the user. Instead, multiple shadow maps are generated. The first shadow map focuses on the area closest to the user, while the following shadow maps focus on areas increasingly farther away from the user. The texture size of each shadow map is the same.

As a result, the shadow maps corresponding to farther areas are less precise, but video memory is saved and less time is needed per frame to generate the shadow maps. Figure 14 demonstrates the difference in video memory when the shadow map precision should be good enough for acceptable shadows at close range. According to the base assumption of cascaded shadow maps, the loss of precision in the later shadow maps is not very noticeable, because shadows in farther areas do not have to be as precise as in the nearest area.

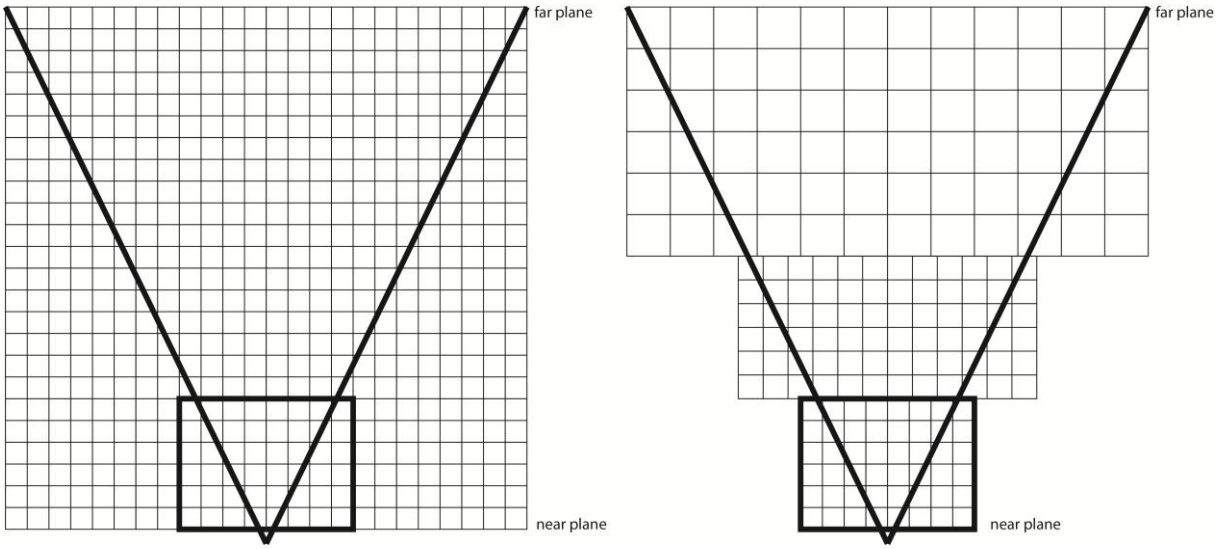


Figure 14. A single shadow map wastes a lot of memory and precision (left) while cascaded shadow maps scale precision and memory depending on distance from the viewer (right).

6.3.6.2 Scene Graph Setup

Scene graph wise, the implementation of shadow directional lights bears similarities to shadow spot lights. Instances of shadow directional lights are placed under the **shadow directional light group**. Figure 15 shows the internal graph of one shadow directional light.

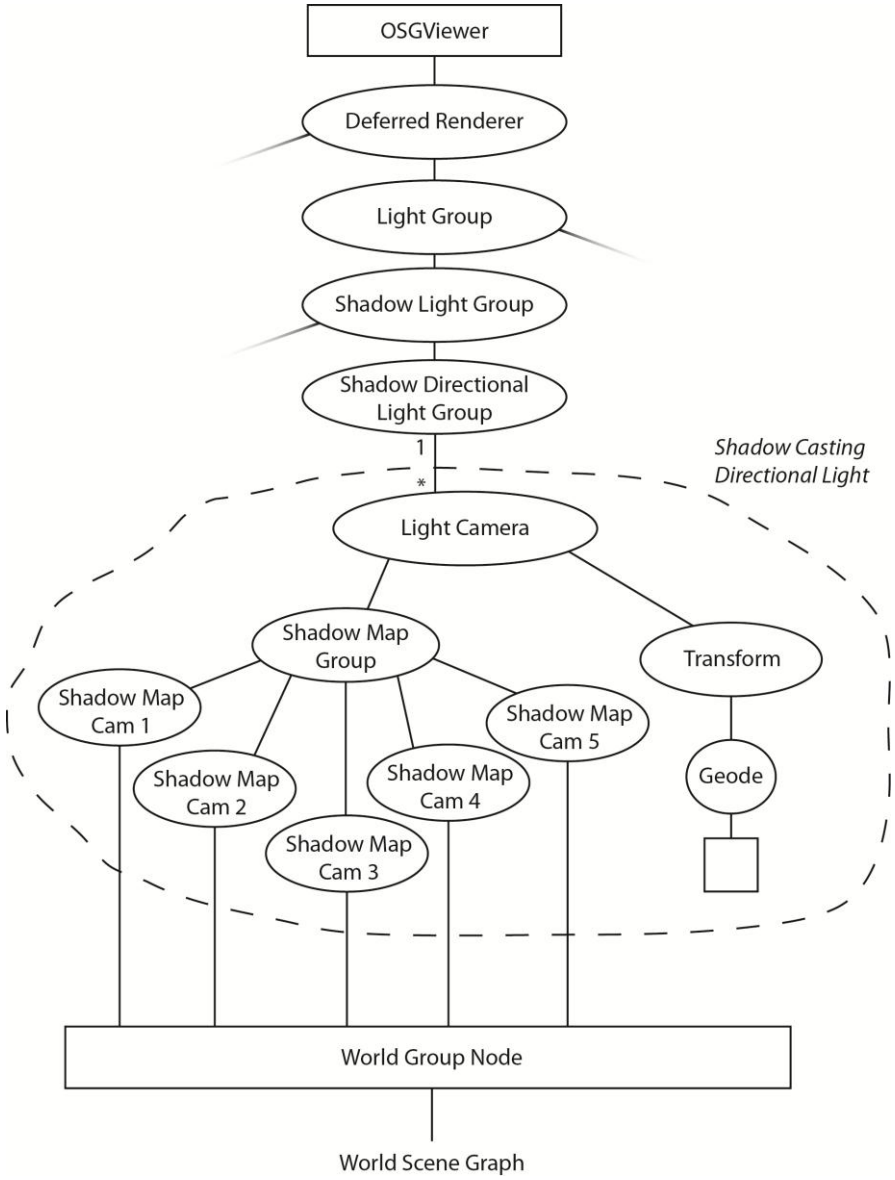


Figure 15. Scene graph representing one shadow directional light.

At the root of the internal graph is the **light camera**. The function of this camera, just as in the case of shadow spot lights, is to ensure that this directional light's full screen quad is drawn directly after generating the shadow maps.

The current implementation uses five shadow maps, stored in video memory as a 2D texture array. For each of the maps in the texture array, one **shadow map camera** is constructed and placed under the **shadow map group.** The shadow map group is a node used solely for optimization: the shadow mapping shader is bound to the state set of the shadow map group, rather than the shadow map cameras individually, to prevent unnecessary shader rebinding between the rendering of each shadow map camera (see Section 4.2.3).

6.3.6.3 Process

Initially, four cascaded shadow maps were used. The four shadow map cameras each looked over one part of the world camera's (i.e. the user's) viewing frustum, based on distance from the user. Since the user will often be in motion, the view- and projection matrices of the shadow map cameras are recalculated each frame.

The parallel split scheme discussed in (Fan Zhang, 2007) is used to calculate at which distances the viewing frustum should be split. This split scheme receives as input: the world camera's near and far plane, and the number of cascaded shadow maps to use.

6.3.6.3.1 Testing the Initial Implementation

A problem presented itself while testing the first implementation: shadows were often flickering. One of the reasons for this was because the world camera used automatic near and far plane adjustment: the near and far plane of the world camera's projection matrix are clamped every frame to the nearest and farthest geometry in view. This caused the scale of the area covered by each shadow map to fluctuate while, for example, looking to the ground and then turning to the sky. Shadows would look fine when viewing a still frame, but would change too sporadically when the user was moving.

This particular problem was worked around, by setting a minimum and maximum shadow distance. In effect, this could be seen as focusing the shadow map cameras on a viewing frustum of fixed size, **without** actually making the world camera's near- and far plane static.

6.3.6.3.2 Tweaking Maximum Shadow Distance

The problem then became to find the ideal maximum shadow distance for SketchaWorld. In most applications featuring a 3D virtual world the user's movement is constrained by gravity, such as in many first person games. In these cases, the maximum shadow distance can be set based on the scale of the world. A loose example, if a house is 10 world units, then a maximum shadow distance of 100 world units would be ideal: if the shadows fade out between 90 to 100 units away from the user, this would probably not be very noticeable. Yet, the distance limit would ensure that the shadow maps have enough precision to produce accurate shadows.

In SketchaWorld, however, the user's movement is not constrained: the user is allowed to fly through the world at arbitrary heights. To implement acceptable shadows in such a scenario proved to be quite challenging.

To recognize the problem, consider the following scenario: In SketchaWorld a tall building could have a base of 15 by 15 world units, and at a height of 80 world units. When standing close to the building, precise shadows are expected to be cast from the building. At a different location on the map, there could be a huge mountain with a ridge of 800 world units long and a height of 400 world units. When flying over the mountain, an impressive sight would be to see the mountain's huge shadow fall over the valley below.

To see any shadows from a large mountain at all, the maximum shadow distance would have to set to a high value, such as 1500. However, splitting a view frustum with far-plane 1500 into four parts would result in bad precision when standing near relatively small objects, causing erroneous self-shadowing. A screenshot displaying this can be seen in Figure 16.

Figure 16. Due to insufficient precision, the road casts shadows on itself.

6.3.6.3.3 Extending the Practical Split Scheme

The final implementation uses a custom method of splitting the viewing frustum into parts. An extra shadow map is added, to make a total number of five shadow maps. The areas on which the first four shadow map cameras focus are decided using parallel split scheme. The maximum shadow distance when calculating these splits is set to 500.

An additional shadow map camera is created, that generates the last shadow map. This last shadow map is an exception. The area that is drawn into this shadow map is always the area that is between 500 and 5000 world units away from the user.

Finally, an extra step has to be taken to make sure that there are no visible artefacts in the scene at areas that are 500 meters away from the user. This is the border between the relatively precise fourth shadow map and the imprecise last shadow map. Any potential artefacts are masked by letting the shadows sampled from the fourth shadow map fade out, while letting the shadows sampled from the last shadow map fade in.

The result allows for precise shadows at close range, where the shadow maps have adequate precision due to setting a maximum range for precise shadows. The extra shadow map that covers a huge area allows for impressive effects from larger objects at far distances. When combined, they

present a fully shadowed scene in SketchaWorld, independent of where the user is standing or hovering in the world. Figure 17 shows the final result.

Figure 17. A mountain to the right casts its shadow on the city.

6.4 Combine Phase

The world geometry and lighting results are combined into a new texture; the Diffuse Texture in the G-Buffer and the Light Texture are added together. This result is called the Combined Texture.

6.5 Additional Rendering Phase

Even though all geometry and lighting tasks have been completed, uncolored pixels can still remain, for example in the sky. Forward rendering is applied to render the atmosphere. Transparent objects cannot be handled by the deferred rendering approach so this is also added to the scene using forward rendering (after rendering the sky).

After this forward rendering step, post processing effects added to the Deferred Renderer are executed.

6.5.1 Forward Rendering

Forward rendering is applied to draw the sky, and more importantly to render transparent geometry.

6.5.1.1 Sky

To keep all sky effects centered around the camera, the sky camera needs to do an additional operation each time the world's view updates. This is done by extracting the rotation from the view matrix coming from the manipulator callback, and using this to build a non-translated view matrix.

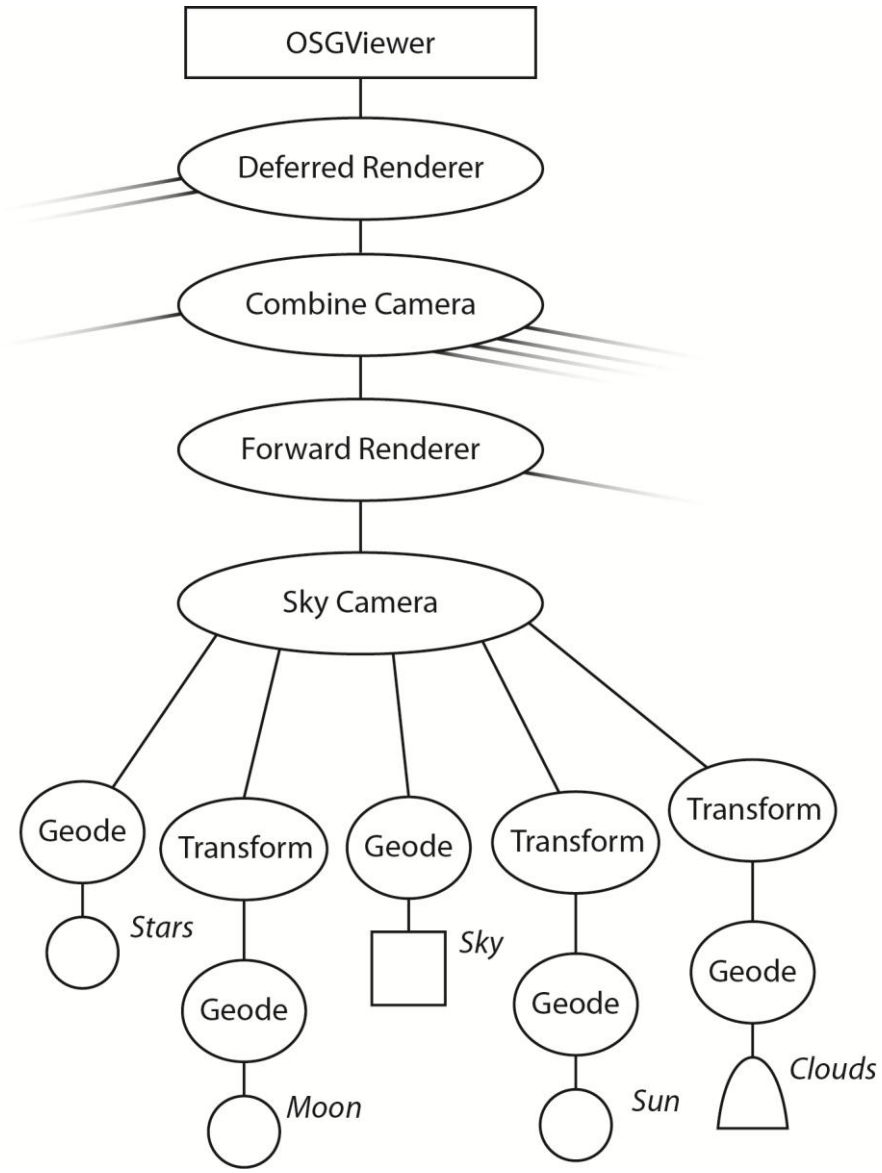


Figure 18. The scene graph of the sky camera.

In addition, each of the individual sky effects discussed in this subchapter are rendered in sequence. Stars are always first, followed by the moon, the atmosphere and finally the sun. It might seem illogical to render the sun last, but this choice allows for a simpler atmosphere calculation. It does make it impossible for eclipses to occur naturally.

6.5.1.1.1 Stars

Stars are rendered using a series of point sprites that are generated based on real world star data. A magnitude parameter is used to darken or lighten the points creating a significantly more realistic effect.

6.5.1.1.2 Moon

A textured sphere is used to visualize the moon. The moon is illuminated by the sun, taking its current direction into account to make waning and waxing moons appear correctly. As mentioned before, eclipses are not taken into account.

6.5.1.1.3 Atmosphere

The atmosphere is rendered using a simple quad geometry that samples from a 2d lookup texture. The result is a realistic looking sky that is very efficient and does not have to compute any of the scattering equations in real-time. Because we use a two dimensional lookup texture we are limited in using only two parameters for determining sky color.

The vertical coordinate is the zenith angle. Using the inverse view projection matrix of the sky camera, the screen space position from the quad is transformed into world space. These positions are used to construct a view ray from the camera into the atmosphere. The z value of this view ray defines the zenith.

The horizontal coordinate we use for the lookup texture is the current time of day. This parameter is calculated based on the current sun direction. This is done by scaling the z-component into a [0, 1] range.

Figure 19 shows an example of a lookup texture that could be used to determine sky color. The horizontal axis corresponds to the time parameter and the vertical axis corresponds to the zenith angle. Note that although the renderer uses this texture as a default implementation it is possible to exchange the lookup texture and achieve different atmospheric effects. Also note that in order to make the moon and stars visible, one simply needs to make parts of this lookup texture transparent.

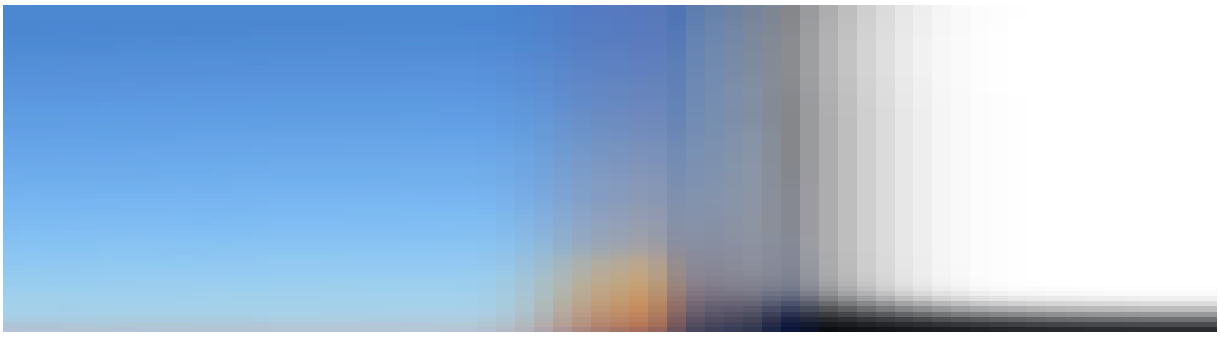


Figure 19. Lookup texture for sky gradient.

6.5.1.1.4 Sun

The sun is rendered using a textured quad that is moved across the sky based on the sun direction. An additional one-dimensional lookup texture is used to determine the sun color based on the time of day.

6.5.1.1.5 Clouds

Clouds have been implemented based on a multi-layer animated perlin noise function with density based lighting. This method results in realistic cloud coverage of the sky in overcast weather conditions, especially when viewed from low altitudes.

A pre-generated perlin noise texture is used as the basis for the cloud shapes. By resampling this texture at different scales it is possible to add fine details to these basic cloud shapes. This results in a single float value that can be used as an approximation of the cloud density.

Using this density value, a simple shading technique is applied. Typically, parts of the cloud with a lower density will have a whiter color and be more transparent. Parts with high density values will become darker and be less transparent. Finally, we multiply the cloud color with the current ambient light conditions. This effect is particularly visible during sunrise and sunset causing the clouds to turn red.

By rendering a series of cloud domes at different heights and with different texture scales it is possible to simulate several cloud layers. It is possible to set a different cloud movement vector for each layer. This increases the overall randomness of the cloud layer and removes any tiling issues that might otherwise still remain.

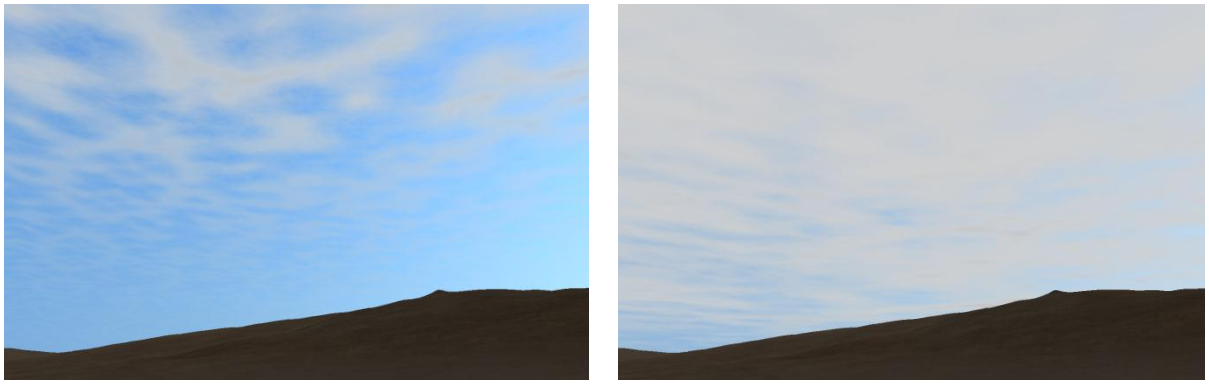


Figure 20. Single cloud layer (left) vs. four cloud layers (right).

Because we do not use expensive simulation models or three dimensional volumetric particles systems, our cloud rendering system is very efficient. However, this comes at the price of reduced rendering realism when viewing the clouds from higher altitudes.

6.5.1.2 Transparency

Rendering transparent object is not possible within the deferred rendering pipeline. To solve this problem we draw transparent objects in the forward rendering phase.

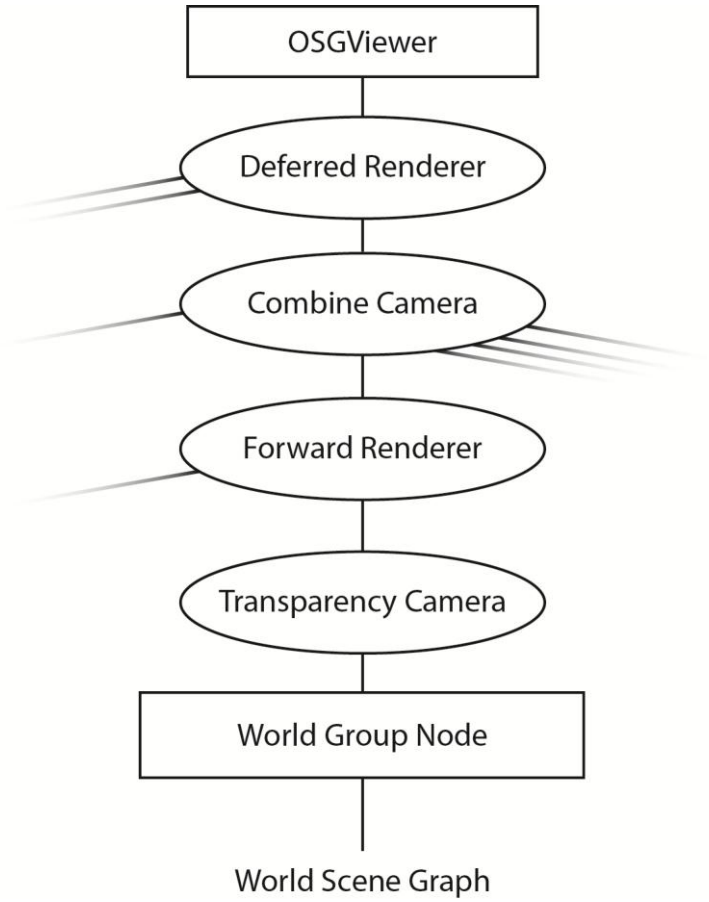


Figure 21. The scene graph of the transparency camera.

Using a transparency state mask we are able to filter all objects that are opaque and only render the transparent objects. These transparent objects do not render to any of the g-buffers as described in Section 6.2.1 but they render to a special transparency buffer.

When rendering to the transparency buffer it is important that the output is pre-lighted. The process of lighting transparent objects equivalent to the deferred lights is described in Section 7.2 as well as in the integration guide.

This transparency buffer is added to the output of the combine phase from the deferred renderer. Because this happens before any of the post processing effects are applied, effects such as bloom and tone mapping are still supported for transparent objects.

Note that when rendering transparent objects it is important that the objects are rendered from back to front. This problem was easily solved because OpenSceneGraph has built-in support for sorting transparent objects.

6.5.2 Post Processing Phase

In the last phase, post processing effects are executed on the result of the composition phase (including the forward rendering operations). To take multiple post processing effects into account,

the renderer contains a post processing framework. The framework maintains a pair of post processing buffers, which are subsequently linked to each effect (a technique called ping-ponging).

Post processing effects are added to this framework through the deferred renderer. This effect has to be derived from BasePostProcessingCamera. Currently, the rendering order of the camera is used to sort each effect. Sorting is mostly used to keep the framework simple, but it is not critical for the effects to work.

6.5.2.1 Night Vision

The night vision post processing effect serves as an example of implementing a custom post processing effect for the framework. It's a simple effect, consisting mostly of transforming pixels from an input.

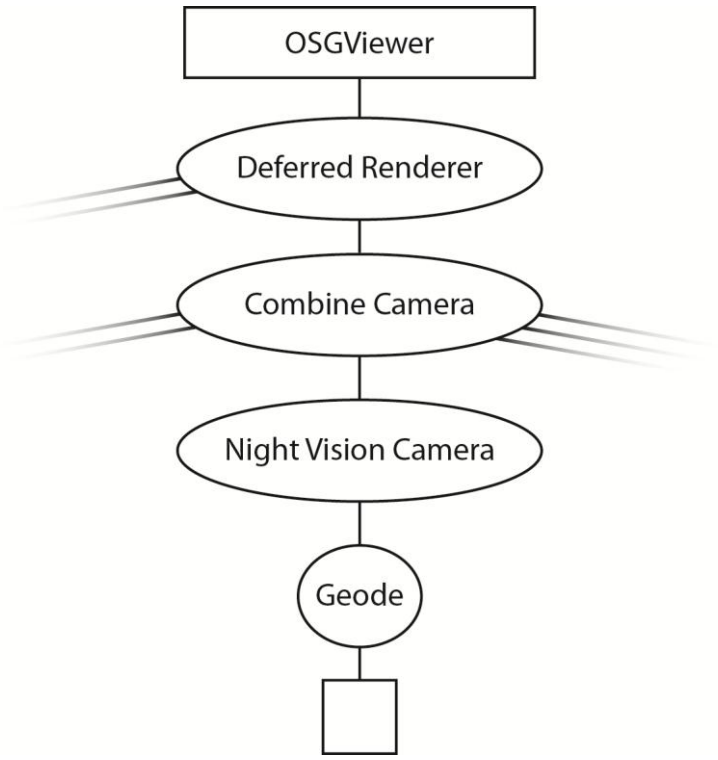


Figure 22. The scene graph of the night vision post processing effect.

This post processing effect is currently implemented mainly as a basic fragment shader. It assumes to receive the direct result of the composition phase. For each pixel on this input, an intensity factor is calculated simply by adding all channels into one float. The result of this calculation is then outputted with average intensity mapping to a green value, and brighter intensities mapping to full white and higher. Adding the related, but different bloom post processing effect would utilize the high ranges to complete the effect.

6.5.2.2 Blur

The blur post processing effect is used to generate a fullscreen gaussian blur. The desired blurring effect is achieved in the fragment shader by sampling nearby pixels and making a weighted average. A level of efficiency is achieved as instead of sampling all nearby pixels in one pass, two passes are used, one blurring horizontally and one vertically. This requires one extra buffer between these passes, but it reduces the number of samples from $O(n^2)$ to $O(2 * n)$. Writing to a temporary buffer

and then writing back to the original buffer also helps solve another issue: the inability to read and write from and to the same texture (Robert, 2008).

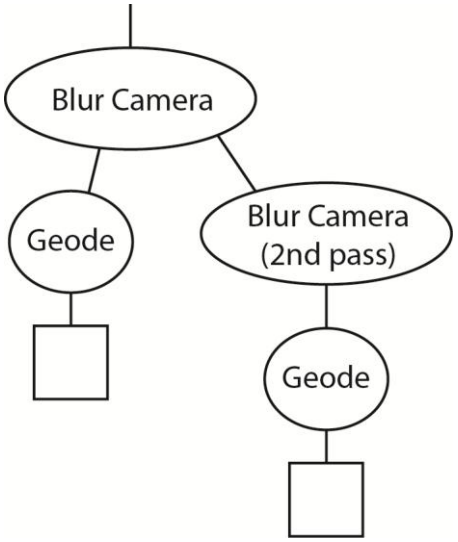


Figure 23. The scene graph of the blur post processing effect.

Although the Blur Camera can be added to the post processing effects, it will not contribute to a more realistic output. It is more commonly used within other post processing effects, such as bloom.

6.5.2.3 Bloom

The bloom post processing effect consists of a multitude of camera's and effects. The process is loosely based on (Houlmann & Metz) and is as follows:

1. The Brightness Map Camera takes the input texture and extracts only the bright parts of the input, storing them in a new temporary buffer (called the brightness map).
2. The first Down Sampling Camera then creates a copy of the brightness map, sized down by a factor of 2. The next Down Sampling Camera then takes that sized down texture as its input, and creates another copy of it, sized down by the same factor. The final Down Sampling Camera sizes it down another step.
3. Next the Bloom Map Camera takes the original brightness map and the three down sampled copies, sizing them back up to the original format and mixing them together. The result is a distorted, blurred texture containing square regions because of the up-sized textures. The down sampled maps that have been scaled up cause the increased area of effect that the bright areas have, creating the bloom effect.
4. In order to smooth out the square regions that originate from step 3, the bloom map is blurred by two Blur Camera's (the first blurring horizontally, the second blurring vertically).
5. The resulting bloom map is then combined with the original input texture by the Bloom Camera and is written to the post processing buffer designated as the output for this post processing effect, completing the procedure.

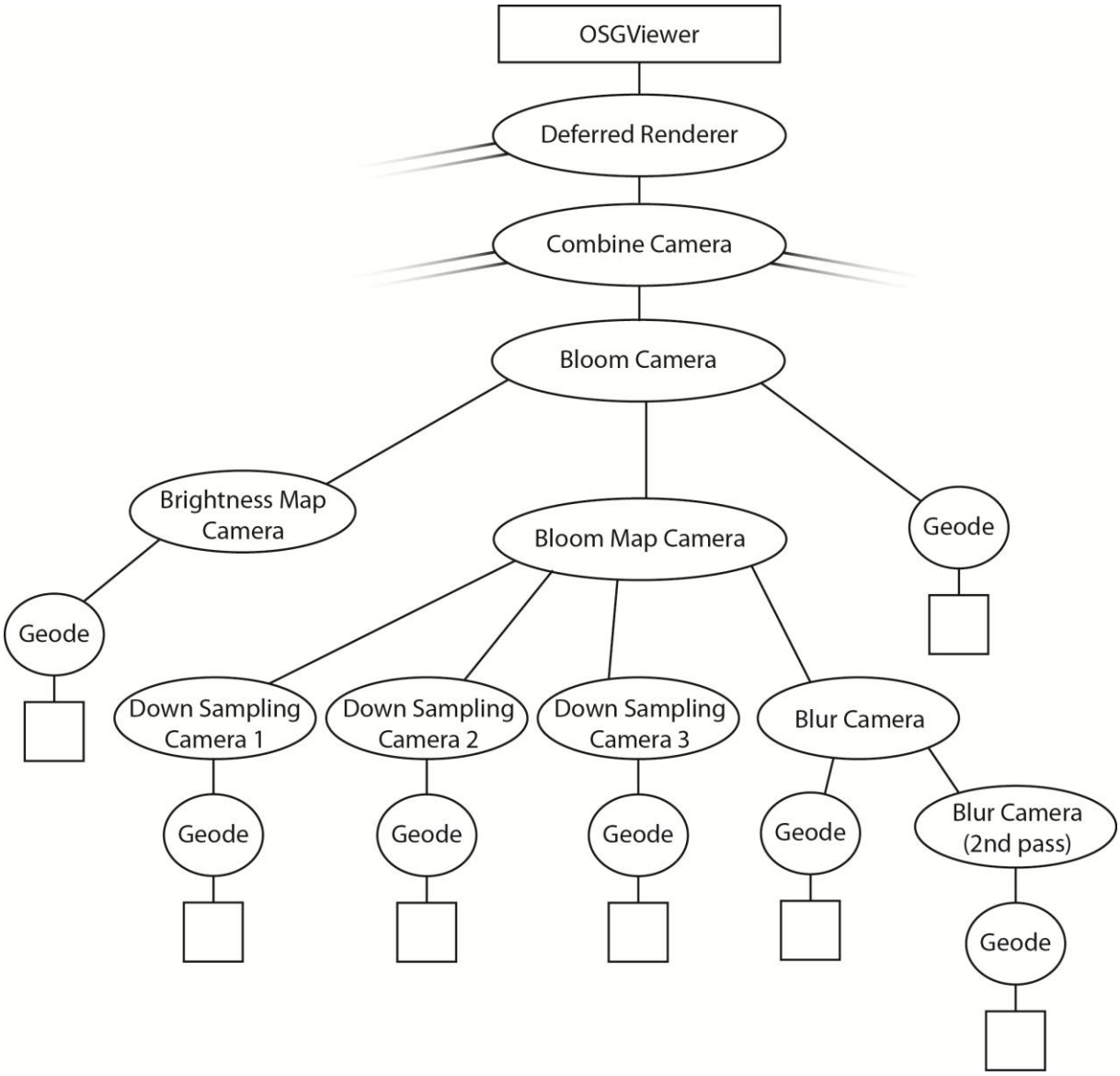


Figure 24. The scene graph of the bloom post processing effect.

6.5.2.4 Tone Mapping

Some post processing effects (such as bloom) cause the output image to contain color values that are outside the Low Dynamic Range (LDR) and inside the High Dynamic Range (HDR), meaning that the color values are greater than 1. A Tone Mapping effect will map those HDR colors back to LDR, preferably in some way that preserves the contrast of the image.

The implementation in the deferred renderer is based on (Z, 2008). The procedure is as follows:

1. First the Average Logarithmic Luminance Camera calculates the average luminance of the input texture. It does so by first converting all RGB values to CIE XYZ and then into CIE Yxy, where Y is then the luminance value of a pixel. The logarithm of Y is then written to a new texture, with hardware mipmapping enabled. The lowest level mipmap is then as close as possible to a 1x1 texture (depending on the screen's width/height ratio) containing the average logarithmic luminance.
2. The Adapt Average Luminance Camera then takes the texture produced in step 1 as its input and samples the exact center with linear filtering enabled, getting the absolute average logarithmic luminance. The exponent of that value, the real average luminance, is then stored in a 1x1 texture.

Finally, the Tone Mapping Camera takes the original input texture and the average luminance texture as its input, passing over every pixel on the screen, tone mapping the values to the correct output texture. The shader employs Reinhardt's Tonemapping Operator (Reinhard & al, 2002), which is generally seen as the industry standard.

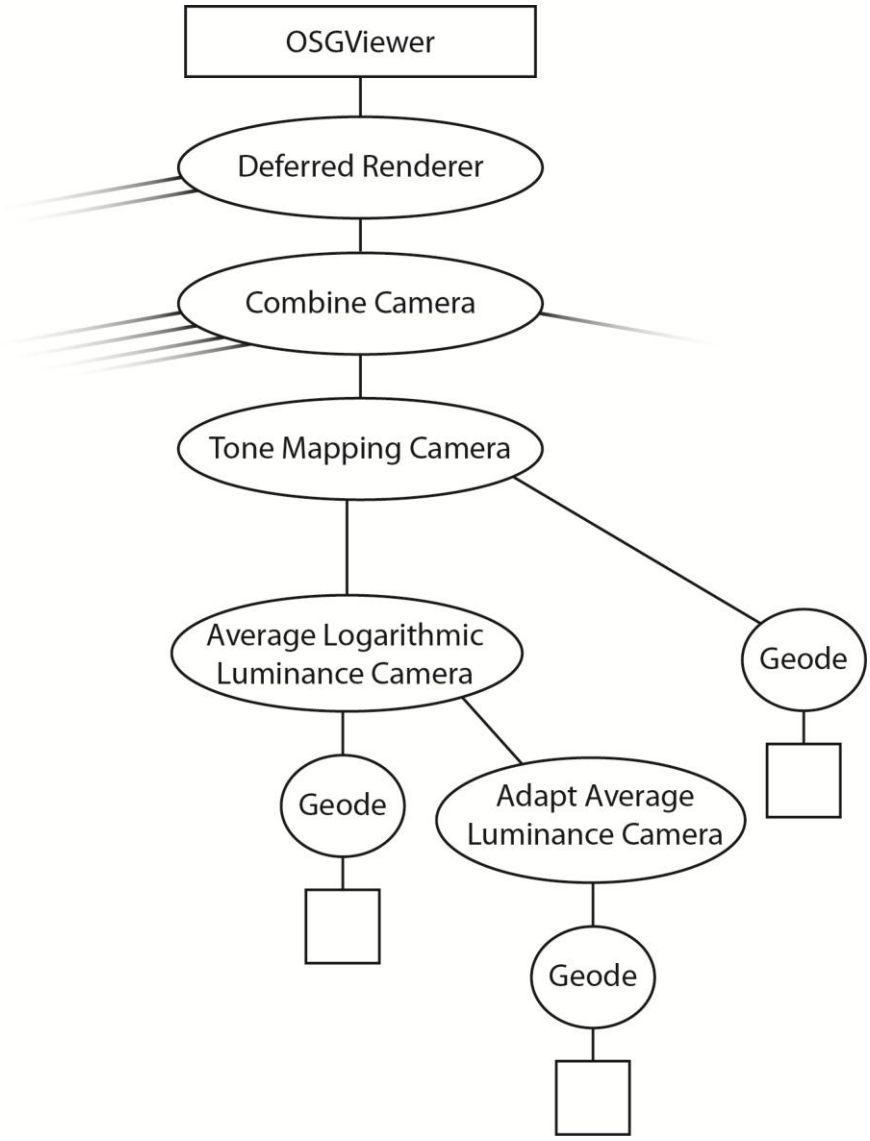


Figure 25. The scene graph of the tone mapping post processing effect.

6.6 Final Phase

This final, simple yet important step takes the output of the last post processing effect and outputs it to the screen.

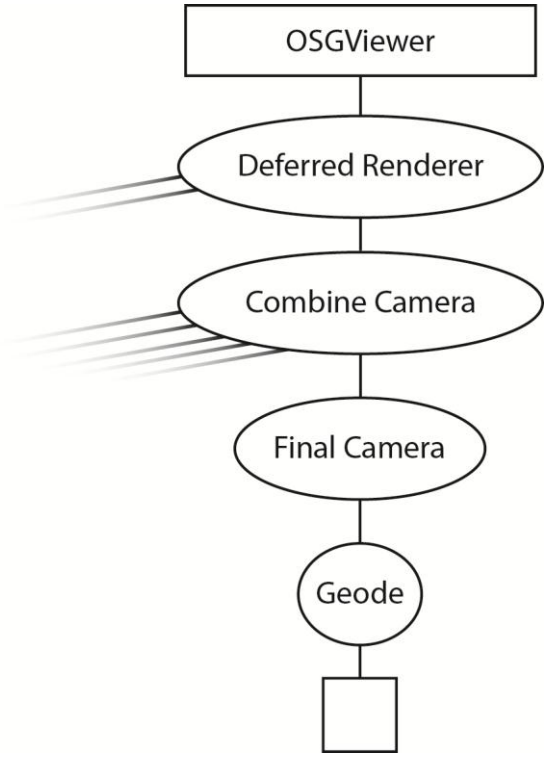


Figure 26. The scene graph of the final camera.

7 Integration

7.1 Converting SketchaWorld's Shaders to Write to the G-buffers

Before the deferred rendering system was introduced, SketchaWorld used a forward rendering system with several specialized shaders. These shaders integrated all lighting and shading operations and wrote their output directly to the back buffer. Our deferred rendering system expects custom shaders to output the diffuse, specular, normal and glow components to separate render targets. Because of this difference the shaders had to be converted before they were applicable to the deferred renderer.

These conversions were relatively straightforward, most of the components needed by the deferred renderer were already being calculated and stored as intermediate shader variables. In addition all lighting code in the shader had to be removed.

A small nuisance was the fact that we had to convert all shaders to comply with version 150 of the GLSL shading language. This required us to make a number of simple syntax changes to the shaders.

Listing 1 shows the shader used for bridges before and after the shader conversion. It illustrates the use of 3 output render targets en several version 150 syntax changes.

```
/*
 * Bridge Fragment Shader.
 */
#version 110

uniform sampler2D texture;

varying float fogAmount;

varying float distanceToEye;

varying vec3 vertex;

varying vec3 light;

varying vec3 normal;

uniform vec3 sunDir;

#include "common.fs"

void main () {

  vec3 tex = texture2D(texture,
    gl_TexCoord[0].st).rgb;

  vec3 norm = normalize(normal);
  float NdotL = max(0.0, dot(sunDir, norm));

  float sunFactor = diffuse*NdotL+ambient;

  tex.rgb *= mix(darkness, sunlight,
    sunFactor);

  tex = applyFog(tex, distanceToEye,
    normalize(vertex), normalize(light));

  gl_FragColor = vec4(tex, 1.0);
}
```

```
/*
 * Bridge Fragment Shader.
 */
#version 150

uniform sampler2D texture;

in vec2 texcoord;
in vec3 normalVS;

out vec4 rtt_diffuseSpecular;
out vec4 rtt_normal;
out vec4 rtt_light;

void main () {

  vec3 bridgeColor =
    texture2D(texture, texcoord).rgb;

  float specular = 0.0;

  // Renormalize the normal from the vertex
  // shader
  vec3 normal = normalize(normalVS);

  // Output the diffuse color and specular
  // factor
  rtt_diffuseSpecular =
    vec4(bridgeColor, specular);

  // Output the scaled normal vector into the
  // [0, 1] range
  rtt_normal = vec4(normal * 0.5 + 0.5, 0.0);

  // Do not directly output any light
  rtt_light = vec4(0);
}
```

Listing 1. Bridge fragment shader before (left) and after (right) conversion.

7.2 Handling transparent geometry

A deferred renderer does not support transparent geometry by default. However, as described in Section 6.5.1.2, transparency support has been added by supplementing the deferred rendering process by a forward renderer. Because transparent object are not drawn in the deferred rendering stage they require specialized shaders.

Shaders for transparent geometry write their output to a single transparency render target. Because this transparency render target is combined with the deferred rendering output after the lighting has already been processed, the transparency output is expected to be pre-lighted.

To facilitate lighting calculations in a transparency shader we automatically provide uniforms for up to 4 directional lights and 4 ambient lights. These contain all information needed to perform the lighting calculations. However, to make writing transparency shaders even easier we have introduced a special macro keyword that automatically inserts the shader code needed for the lighting calculations. This reduces special shader code for transparent geometry to a single macro definition and a function call to `computeLighting`.

There is no support for additional lights and/or shadowed lights. However, it is possible for users to perform these lighting calculations themselves. Shader code can be based on the deferred light shaders in 'Resource Files\Shaders\Lights'.

Listing 2 shows an example of a simple transparency shader used in SketchaWorld.

```
/*
 *  Lake Fragment Shader.
 */
#version 150
#rr_transparency

// Depth texture used for custom depth testing
uniform sampler2D depthTexture;

const vec3 lightBlue = vec3(0.725, 0.819, 0.913);

in vec3 normalVS;
in vec3 positionVS;
in vec4 positionSS;

out vec4 rtt_transparency;

void main () {

        // Test if pixel is occluded by opaque geometry
        vec3 screenPos = positionSS.xyz / positionSS.w;

    float depth = texture2D(depthTexture, screenPos.xy * 0.5 + 0.5).x * 2.0 - 1.0;

        if (depth < screenPos.z)
                discard;

        // Compute all necessary information for lighting calculations
    vec3 diffuseColor = lightBlue;
    float specularIntensity = 0.9;
    vec3 normal = normalize(normalVS);

        // Compute lighting with imported function
    vec3 result = computeLighting(diffuseColor, specularIntensity, normal, positionVS);

        // Output results
        rtt_transparency = vec4(result, 0.8);
}
```

Listing 2. A simple transparency shader used in SketchaWorld.

8 Evaluation

This section discusses how the final product compares to the requirements, as discussed in the **requirements analysis document** (Appendix E). The requirements are specified according to the MoSCoW model. The requirements will be evaluated in the same order as they are listed in the RAD.

8.1 Must Haves

The first *must have* requirement is to build a renderer that makes use of the **deferred rendering** approach. As discussed in Sections 5 and 6, the deferred renderer has successfully been implemented.

The shadowing techniques **cascaded shadow mapping** and **percentage-closer soft shadows** have been implemented for directional lights and spotlights respectively. A combination of both techniques has been considered, to further improve the shadow quality for directional lights. In the end this has not been implemented, because the necessary time to tweak this exceeded the project schedule.

The mapping techniques **normal mapping** and **parallax mapping** have all been implemented successfully and are fully supported. Additionally, **parallax occlusion mapping** has been implemented. Parallax occlusion mapping improves on the basic parallax mapping, but is also more computationally heavy. The user can configure which form of parallax mapping to use.

High dynamic range and **tone mapping** is fully supported in the rendering system. Currently Reinhard's tone mapping operator is implemented. Even though it is a valid HDR to LDR mapping, the effect of using HDR is currently not clearly exhibited in SketchaWorld. Using a different tone mapping operator, one that is more aimed towards SketchaWorld's scenes, might yield better results.

Analytic single scattering has not been implemented. During the implementation period, as the authors gained more insight into deferred rendering, it was decided that the quality of analytic single scattering would not be very noticeable in SketchaWorld's outdoor scenes. Instead, a custom global distance fog model has been included in the sky model.

Atmospheric scattering has been replaced by a custom sky model that uses pre-computed lookup textures. This choice has resulted in a system with better performance, but sacrifices configurability of atmospheric parameters that can be achieved with a full atmospheric scattering implementation.

Cloud density layers have been implemented as described in section 6.5.1.1.5. More realistic sky models that use volumetric simulations could be used to improve the sky but this was not part of any must have requirement.

8.2 Should haves

Procedural terrain shading has not been implemented. Procedural terrain shading concerns techniques that make use of the terrain's geometric data to influence the rendering process. During the implementation period, it became apparent that this subject lies outside of the scope of this project.

Raindrops using geometry shaders has also not been implemented. The time scheduled to implement this technique has been sacrificed to spend more time on tweaking the shadowing techniques.

8.3 Could Haves

Screen space ambient occlusion is listed in the RAD as a *could have* requirement. Despite being listed as a low priority feature, the technique has been implemented in the final product. This decision was made because the ambient occlusion effect worked well in conjunction with the deferred renderer and it improved rendering realism significantly.

9 Conclusions

The goal of this project was to create a realistic renderer for SketchaWorld. The level of perceived (visual) realism is a subjective criterion and people's expectations grow as hardware limitations diminish.

The authors have done research on many rendering techniques that enhance realism. Together with Ir. R.M. Smelik the authors have selected those techniques that would create a good balance between the level of realism and the support by the GPU's that are currently available, while also keeping an eye on the time available for this project.

The selected techniques have been formalized during the start-up phase of the project in terms of must haves and should haves (according to the MoSCoW model). All must haves have been implemented and the authors are satisfied with the results.

Implementing shadow lights cost more time than was anticipated. The Percentage Closer Soft Shadows algorithm and the Cascaded Shadow Maps algorithm both took up bigger parts of the schedule than was originally planned. The techniques rely on some assumptions that are violated by SketchaWorld's view on the generated worlds. For example, it is assumed that the designer can tweak the view of the virtual world so that the shadow maps fit properly. However, in SketchaWorld, the distance one can see can vary from very nearby to very far away, or even both at the same time in the same viewport. The shadow maps need to be adjusted automatically to look good in all possible cases, which posed a significant challenge.

10 Recommendations

In this section, the authors list suggestions for further improvement to the rendering procedure in SketchaWorld.

Cascaded shadow mapping is currently tweaked to display acceptable shadows with five 1024x1024 shadow textures. The need for this high resolution is to support shadows at both close and far distances, with minimal flickering near the edges of shadowed areas. Tweaking cascaded shadow maps is a much discussed subject. One potential improvement that the authors have found is to stabilize the shadow maps. This is done by snapping the view and projection of the shadow map generating cameras to points on a fixed 3D grid. The authors have not had the opportunity to experiment with this improvement themselves, due to time constraints.

Considering as there is no shadow variant of a point light, this might be worth looking into. It should be possible to implement a shadow point light. The authors have avoided implementing this light, because of the amount of time that would be needed to correctly implement cube mapping.

One of the lower priority requirements was to implement dynamic weather functionality. As mentioned in the evaluation, this task was dropped in favor of improving other techniques. It should be possible to use the existing transparency rendering phase to render effects such as rain, mist or dust as bill boarded particles. Additionally, the orientation report mentions Kawase's bloom filter, which is a post processing technique that can cheaply produce bloom over large areas.

As Reinhart's tone mapping operator has not worked as desired with the renderer, other solutions might result in better overall results. One possibility would be to sample luminosity values from more localized sections of the screen (as opposed to one global sample). Basically, this would be a histogram based approach. Based on the variation and values of the local averages, the tone mapping operator would be more able to judge what should be made visible.

Finally, SketchaWorld's generated cities can contain many buildings. The buildings already have multiple level of detail models, which enables entire cities to be viewed from a distance. However, when viewing the city from close by, the scene becomes very geometry heavy. Experimentation with replacing small geometric indentations with parallax height maps, normal maps, or both may lead to a considerable increase in frame rate.

11 References

Bavoil, L., & Sainz, M. (2008, September). NVidia Screen Space Ambient Occlusion.

Dimitrov, R. (2007, August 7). *Cascaded Shadow Maps.* Retrieved 6 14, 2011, from NVIDIA Developers: http://developer.download.nvidia.com/SDK/10.5/opengl/src/cascaded_shadow_maps/doc/cascaded_shadow_maps.pdf

Fan Zhang, H. S. (2007). Parallel-Split Shadow Maps. In H. Nguyen, *GPU Gems 3.* Addison-Wesley Professional.

Fernando, R. (2005). Percentage-Closer Soft Shadows. Boston: i3D.

Houlmann, F., & Metz, S. (n.d.). *High Dynamic Range Rendering in OpenGL.* Université de Technologie Belfort-Montbéliard.

Kaneko, T., & Takahei, T. (2001). Detailed Shape Representation with Parallax Mapping. *ICAT*, (pp. 205 - 208). Tokyo.

Kevin Myers, R. F. (2008, February). *Integrating Realistic Soft Shadows Into Your Game Engine.* Retrieved 6 14, 2011, from NVIDIA Developer: http://developer.download.nvidia.com/shaderlibrary/docs/shadow_PCSS.pdf

McGuire, M. (2005). Steep Parallax Mapping. I3D Posters Session.

Mittring, M. (2007). Finding Next Gen - CryEngine 2. *Advanced Real-Time Rendering in 3D Graphics and Games Course*, (pp. 97 - 121).

Policarpo, F., & Oliveira, M. M. (2006). Relief Mapping of Non-Height-Field Surface Details.

Reinhard, E., & al, e. (2002). Photographic Tone Reproduction for Digital Images. *SIGGRAPH 2002.* ACM Transactions on Graphics.

Robert. (2008, October 11). *Gaussian Blur Filter Shader*. Retrieved July 2, 2011, from gameRENDERING: http://www.gamerendering.com/2008/10/11/gaussian-blur-filter-shader/

Valient, M. (2007). Deferred Rendering in Killzone 2. *Develop Conference* (p. 55). Brighton: Develop Conference.

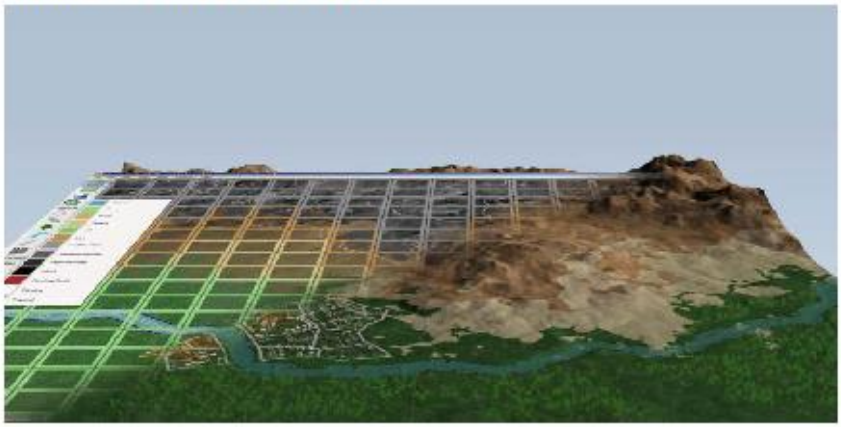
Z, J. (2008, November 8). *D3DBook:High-Dynamic Range Rendering*. Retrieved July 2, 2011, from GameDev.net: http://wiki.gamedev.net/index.php/D3DBook:High-Dynamic_Range_Rendering

Appendix A: Assignment Description

December 10, 2010

B.Sc. project: Realistic Rendering of Virtual Worlds

Manual creation of virtual worlds, such as game levels, is becoming more and more laborious and costly, as the size and level of detail of virtual worlds increase. The prototype system SketchaWorld provides a very accessible and efficient alternative for modeling virtual worlds. Using procedural sketching, it enables non-experts to create complete virtual worlds in minutes.



This process results in a 3D virtual world model that is consistent and functionally realistic. However, an improved visual quality would help user immersion and the level of realism that is perceived.

In this BSc assignment, you will study and implement interactive rendering techniques for improving the quality and perceived realism of SketchaWorld's 3D virtual worlds. We will focus mostly on lighting and shading quality. Potential techniques you could be working on include:

- Cascaded shadow maps and soft shadowing;
- Normal mapping and / or parallax mapping;
- Procedural terrain shading;
- Realistic fog;
- Screen space ambient occlusion.

Depending on your own graphics interests and requirements and priorities for SketchaWorld, we will compose a concrete project plan featuring several of these techniques.

Requirements

- Interest in and general knowledge of computer graphics and interactive techniques;
- Experience with shader programming in GLSL, HLSL or CG;

Some OpenGL experience is a plus.

Appendix B: Integration Guide

TU DELFT – COMPUTER GRAPHICS – B.SC. PROJECT

Realistic Rendering Of Virtual Worlds

Integration Guide

Zhi Kang Shao, Mattijs Driel, Quintijn Hendrickx, Korijn van Golen

31-5-2011

The Integration Guide belonging to the B.Sc. project “Realistic Rendering of Virtual Worlds”. It describes the interface provided by the Realistic Rendering project. Its purpose is to ease the integration of existing scene graphs into the developed rendering engine.

Table of Contents

1 Summary

This document describes how third parties can and should interact with the rendering engine we have developed for the BSc. project “Realistic Rendering of Virtual Worlds”. All important features of the engine are discussed in depth. If applicable, any limitations that are imposed by the rendering will be stated as well.

2 Introduction

We start by discussing how to set up your project environment so as to be able to use the rendering engine in your own project. Following this we show how to initialize and configure the deferred render to your preferences. Next, we will discuss how to add lights to the scene and allow for objects to be prepared to be properly lit by them. Finally we discuss how to write your own shaders that are compatible with the deferred renderer and how to handle transparent objects in your scene.

3 Setting up the project environment

The RealisticRendering project is compiled into a dynamic library (RealisticRendering.dll), additionally an import library (RealisticRendering.lib) is created that contains the symbols needed to interface with this library.

You should add the import library to the linker input dependencies of your compiler and place the dynamic library in a location accessible by the application. This will most likely be the startup directory of your application file.

Note that the RealisticRendering library is most likely compiled using Visual Studio 2008 and will not be compatible with other compilers.

4 Initializing the deferred renderer

To be able to initialize the deferred renderer we must first include the accompanying header file. After this creating a DeferredRenderer instance is very easy, it only requires the initial dimensions of the screen buffer.

```
#include "RealisticRendering/DeferredRenderer.h"

...

int screenWidth = 1024;
int screenHeight = 768;

dr::DeferredRenderer* deferredRenderer = new
dr::DeferredRenderer(screenWidth, screenHeight);
```

Note that altough screen dimension have to be known when creating the deferred renderer they can easily be changed afterwards by calling dr::DeferredRenderer::setBufferSize(int, int).

Next we have to add the deferred renderer to the osgViewer.

```
// Set the deferred render as the root scene data of the viewer
osgViewer->setSceneData(deferredRenderer);
```

```
// Set the world root of the deferred render
deferredRenderer ->setWorld(worldRoot);
```

Finally we have to create a camera manipulator and add it to the osgViewer as well as the deferred renderer. It is possible to use any osgManipulator but in this example we will use the FirstPersonManipulator included in the RealisticRendering project.

```
// Setup a manipulator and bind it to the osgViewer and deferredRenderer
osg::ref_ptr<dr::FirstPersonManipulator> firstPersonManipulator = new
dr::FirstPersonManipulator();

firstPersonManipulator->setAcceleration(10);
firstPersonManipulator->setFriction(0.97);
firstPersonManipulator->setTerrainOffset(5.0);
firstPersonManipulator->setHomePosition(Vec3(0,0,100), Vec3(100,-100,0),
Vec3(0,0,1));
firstPersonManipulator->setTerrainHeightProvider(new HeightProvider());

osgViewer->setCameraManipulator(firstPersonManipulator);
deferredRenderer->setCameraManipulator(firstPersonManipulator);
```

5 Configuring the deferred renderer

Many options in the deferred renderer can be enabled, disabled or configured with various parameters. All configuration settings are applied through the static AppConfig class. A list of available settings and their effect on the deferred renderer is available in the AppConfig header file.

Note that some settings can be changed real-time while others require the deferred renderer to be reinitialized.

```
// Set the base path used to load resources from RealisticRendering
dr::AppConfig::set(dr::BASEPATH, "RealisticRendering/RealisticRendering/");

// Configure SSAO parameters
dr::AppConfig::set(dr::AMBIENT_OCCLUSION, true);
dr::AppConfig::set(dr::AMBIENT_OCCLUSION_INTENSITY, 3.0);
dr::AppConfig::set(dr::AMBIENT_OCCLUSION_SCALE, 1.5);
dr::AppConfig::set(dr::AMBIENT_OCCLUSION_BIAS, 0.2);
dr::AppConfig::set(dr::AMBIENT_OCCLUSION_SAMPLERADIUS 0.4);
```

6 Adding lights to the scene

Without any lights the deferred renderer will not produce any meaningful output. All lights that are supported by the deferred renderer inherit from the dr::DeferredLight base class. All lights inherit the color and intensity parameter but most of them will also have additional parameters available.

Shadowed versions of lights will also produce shadows in the scene, these might not be available or applicable for all types of light.

Class	**Additional parameters**	**Shadowed version**
dr::DirectionalLight	direction	dr::ShadowDirectionalLight
dr::SpotLight	origin, target, radius, radiusDropoff	dr::ShadowSpotLight
dr::PointLight	origin, radius	Not supported

dr::AmbientLight	Not applicable	Not applicable

```
shadowDirLight = new ShadowDirectionalLight(Vec3(0, -1, -1), Vec3(1, 1, 1),
0.8);

ambientLight = new AmbientLight(Vec3(1, 1, 1), 0.2);

shadowSpotLight = new ShadowSpotLight(Vec3(4, 1, 20), Vec3(4, 1, -10), 10,
0.5, Vec3(1, 1, 1), 2.0);

...

renderer->addLight(shadowDirectionalLight);
renderer->addLight(ambientLight);
renderer->addLight(shadowSpotLight);
```

7 Preparing objects for rendering

Objects that are added to the world root will be shown in the deferred renderer. When no custom shader is assigned to the object, the deferred renderer will provide a default shader. This shader has support for diffuse maps, normal maps, specular maps, height maps and glow maps.

Supported mapping techniques include basic normal mapping, parallax mapping and parallax occlusion mapping. Depending on the chosen technique a normal and height map might be required.

Rendering technique	**Required maps**
Flat (No mapping)	
Normal mapping	Normal map
Parallax mapping	Normal map, Height map
Parallax occlusion mapping	Normal map, Height map

Specular or glow maps can be omitted and the deferred render will automatically disable these effects. Not assigning a diffuse map will cause the object to be rendered with a magenta diffuse color.

For a scene graph to be rendered using the default provided shader it is also necessary that it will be pre-processed by the deferred renderer. This step will ensure that all vertex attributes will be available and bound to the correct slots. This processing can be done with a simple function call that accepts an arbitrary scene graph.

```
#include "RealisticRendering/Util.h"

...

hebeStatue = prepareSceneGraph(osgDB::readNodeFile("HebeStatue2.3DS"));
worldRoot->addChild(hebeStatue);
```

Note that the prepareSceneGraph function does not support osg::Drawable objects that are not of the type osg::Geometry. There is also no support for multiple texture coordinates. In any case a warning will be generated to the osg notification log.

An important final step in preparing objects for rendering is setting a node mask. This mask should be set to OPAQUE_BIN for most objects in the scene. When rendering transparent objects it should be set to TRANSPARENT_BIN (see section 9). If no node mask is setup then this will cause the node to be rendered twice.

```
meshGroup->setNodeMask(osg::StateSet::OPAQUE_BIN);
```

8 Writing custom shaders

It might be necessary to apply a custom shader to objects in the scene. If this is the case some points require attention.

The deferred renderer expects shaders that operate in the worldRoot to write their output to the diffuseSpecular and normal render target. Glow effects are supported by writing directly to the light target. These shader outputs have to be declared at the top of the fragment shader.

```
#version 150 core

out vec4 rtt_diffuseSpecular;
out vec4 rtt_normal;
out vec4 rtt_light;
```

When creating the shader program these render targets have to be bound to the correct slots.

```
program->addBindFragDataLocation("rtt_diffuseSpecular", 0);
program->addBindFragDataLocation("rtt_normal", 1);
program->addBindFragDataLocation("rtt_light", 2);
```

Note that the diffuseSpecular target expects the RGB components to contain the diffuse color and the alpha component to contain the specular factor.

The normal target expects the xyz components to contain the normal of the object in view space. The w component is unused. If there are no normal available for the object you should output a vec4(0, 0, 0, 0) value. This will disable automatically disable all lighting for this object.

The light target is in HDR format and allows you to specify a glow component. This will cause the object to appear as if it emits light, although no radiosity effects are applied. Most likely this will be used in conjunction with a Bloom post processing effect.

9 Adding transparent objects

Deferred rendering pipelines do not support transparent objects natively. However, to allow for transparent objects in our rendering engine we have added a separate rendering stage for transparent objects. This stage is executed after the main rendering is finished and is based on a forward rendering approach. This means that all lighting calculations will have to be performed manually. However, the rendering engine provides default functionality for ambient light (without ambient occlusion) and directional light (without shadows).

Every node that should be rendered in the transparent rendering stage needs to have the correct node mask set. This ensures that the object is skipped in the deferred stage and not rendered twice.

```
meshGroup->getOrCreateStateSet()->setRenderingHint(osg::StateSet::TRANSPARENT_BIN);
meshGroup->setNodeMask(osg::StateSet::TRANSPARENT_BIN);
```

Because we are not using the deferred pipeline for transparent objects we only have to define a single output target.

```
#version 150 core

out vec4 rtt_transparency;
```

When creating the shader program this corresponds to the following code.

```
program->addBindFragDataLocation("rtt_transparency ", 0);
```

Because we bypass the deferred renderer we have to do all lighting calculations ourselves. However, default functionality is provided for up to 4 ambient and 4 directional lights. To enable these lights for use in the transparent render stage the affectTransparentObjects properties needs to be set.

```
light = new DirectionalLight(osg::Vec3(0, 1, -0.1), osg::Vec3(1, 1, 1), 1);
light->setAffectTransparentObjects(true);
```

In the shader program the code for the default functionality can be included by adding a transparency tag at the top of the shader file. This is a custom tag that is automatically replaced by the realistic rendering engine.

```
/*
 * Water Fragment Shader.
 */
#version 150
#rr_transparency

...
```

In your main shader body you now have access to the function `computeLighting()`, which will automatically compute lighting by the ambient and directional lights. The following code shows an example how it can be used. Note that normals and position should be provided in view space.

```
vec3 diffuseColor = vec3(1, 0, 1);
float specularFactor = 0.9;
vec3 normalVS = osg_NormalMatrix * vec3(0, 0, 1);
vec3 positionVS = osg_ModelViewMatrix * vec4(in_vertex, 1.0);

vec3 litResult = computeLighting(diffuseColor, specularFactor, normalVS,
positionVS);
```

Appendix C: Orientation Report (Dutch)

TU DELFT – COMPUTER GRAPHICS – B.SC. PROJECT

Realistic Rendering Of Virtual Worlds

Oriëntatieverslag

Zhi Kang Shao, Mattijs Driel, Quintijn Hendrickx, Korijn van Golen

13-4-2011

Een bespreking van het onderzoek dat voorafgegaan is aan het B.Sc. project “Realistic Rendering of Virtual Worlds”. Eerst wordt het onderzoeksdomein vastgesteld. Vervolgens worden verscheidene technieken binnen dat domein besproken en beoordeeld op basis van vooraf opgestelde criteria. Uiteindelijk wordt op basis van multi-criteria analyses een subset te implementeren technieken geselecteerd.

Inhoudsopgave

1 Inleiding

Het project “Realistic rendering of virtual worlds” is ingesteld teneinde de waargenomen werkelijkheid van SketchaWorld te verbeteren. Het doel van een realistischere grafische weergave zal behaald worden door verschillende shading technieken toe te passen. Dit oriëntatieverslag beschrijft de verkenning van de mogelijkheden.

Eerst wordt een onderzoeksdomein vastgesteld, opgebouwd uit een aantal onderzoeksgebieden. Het onderzoek divergeert vervolgens; per gebied zal een aantal technieken worden besproken. Vervolgens convergeert het onderzoek; op basis van vooraf vastgestelde criteria wordt gekozen voor een subset van de gepresenteerde technieken. Deze technieken zullen vervolgens worden geïmplementeerd tijdens de implementatiefase van het onderzoek.

2 Onderzoeksdomein

De opdrachtomschrijving van het project is erg breed opgezet. Op basis van de interesses van de auteurs en de prioriteiten die zij stellen bij SketchaWorld is er de mogelijkheid om de technieken die moeten worden geïmplementeerd bij te stellen, in overleg met de werkgever. Om deze reden is ervoor gekozen om allereerst het onderzoeksdomein te bepalen.

Eerst worden de verschillende opties genoemd die mogelijk bijdragen aan het doel van het project. Vervolgens wordt er op basis van de potentie van elk onderzoeksgebied een onderzoeksdomein vastgesteld, waarbinnen de rest van dit vooronderzoek plaats zal vinden.

2.1 Onderzoeksgebieden

In een divergerend proces zijn de volgende onderzoeksgebieden geïdentificeerd. Tijdens de brainstorm is nog niet gekeken naar eventueel nut voor het project.

1. Shading Method
2. Dynamic Soft Shadows
3. Advanced Mapping Techniques
4. Fog
5. Clouds
6. Skies
7. Procedural Terrain Shading
8. Rain
9. Depth of Field
10. High Dynamic Range en Tone Mapping
11. Bloom
12. Lens Flare
13. Indirect Global Illumination
14. Water

2.2 Selectie

De volgende gebieden zijn in samenwerking met de werkgever geïdentificeerd als potentieel waardevol voor het onderzoek. De gebieden worden gepresenteerd in volgorde van belang en zijn

gecategoriseerd in de stijl van MoSCoW requirements. In het hoofdstuk 3 zullen ze uitgebreid toegelicht worden.

2.2.1 Must Have

1. Shading Method
2. Dynamic Soft Shadows
3. Advanced Mapping Techniques
4. High Dynamic Range en Tone Mapping
5. Fog
6. Skies
7. Clouds

2.2.2 Should Have

8. Procedural Terrain Shading
9. Rain

2.2.3 Could Have

10. Indirect Global Illumination

Nadat de planning is opgesteld zal deze lijst nog worden bijgesteld, deze is dan te vinden in het Requirements Analysis Document.

3 Technieken

3.1 Inleiding

Binnen de verschillende onderzoeksgebieden is een grote variëteit aan technieken beschikbaar. Om een weloverwogen keuze te kunnen maken worden eerst criteria vastgesteld waarop de technieken kunnen worden beoordeeld. Vervolgens wordt per onderzoeksgebied een aantal technieken besproken en beoordeeld op de eerder genoemde criteria. Afsluitend wordt per onderzoeksgebied een conclusie getrokken op basis van een multi-criteria analyse.

3.1.1 Criteria

- Moeilijkheidsgraad implementatie
- Performance
    - Tijd
    - Ruimte
- Externe benodigdheden
- Doeltreffendheid

3.2 Shading methods

Waar voorheen altijd gebruik gemaakt werd van de conventionele Forward Shading techniek, is de nieuwe techniek Deferred Shading in opkomst sinds de beschikbaarheid van meerdere Render Targets in GPU's.

Dit onderdeel van hoofdstuk 3 zal verschillen van de volgende onderdelen van dit hoofdstuk in de zin dat er niet per techniek een discussie wordt aangeboden, maar een vergelijking van de twee opties aan het eind van het onderdeel.

3.2.1 Forward Shading

De traditionele gang van zaken bij realtime rendering heet Forward Shading. Er wordt geometrie gerendered en belicht totdat alle onderdelen van de scene getekend zijn. Daarna vind nog eventuele post processing plaats.

3.2.1.1 Beschrijving

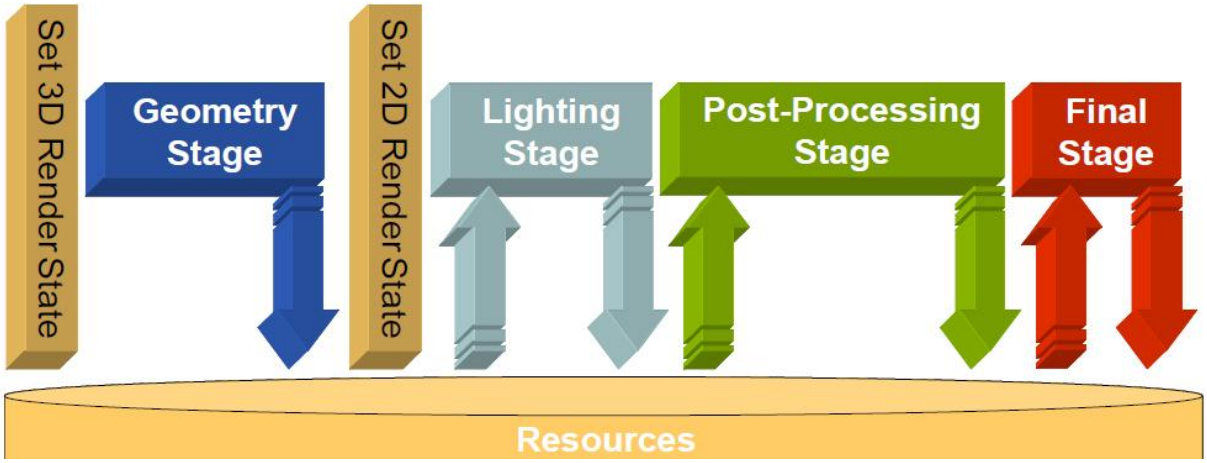
Bij de traditionele methode wordt geometrie naar een vertex shader gestuurd, getransformeerd naar screen coordinates, en de belichting wordt dan ofwel per-vertex of per-pixel berekend. Wanneer er slechts één licht in het proces wordt betrokken is de complexiteit van de vertex shader $O(1)$ aangezien elke vertex éénmalig gerendered wordt. Bij meerdere lichten stijgt de complexiteit naar $O(n)$ aangezien er nu elke vertex n keer gerendered moet worden. Een vergelijkbare argumentatie gaat op wanneer we per pixel renderen in plaats van per vertex. Merk op dat bij Forward rendering dus elke keer de geometrie ook opnieuw wordt gerendered.

3.2.2 Deferred Shading

Deze nieuwe methode maakt gebruik van het feit dat MRTs (Multiple Render Targets) beschikbaar zijn geworden op GPU's. Informatie over geometrie kan van tevoren per pixel worden opgeslagen in verschillende Render Targets en daarna gebruikt worden bij andere calculaties. Op deze manier staat het renderen van geometrie los van belichting en andere effecten.

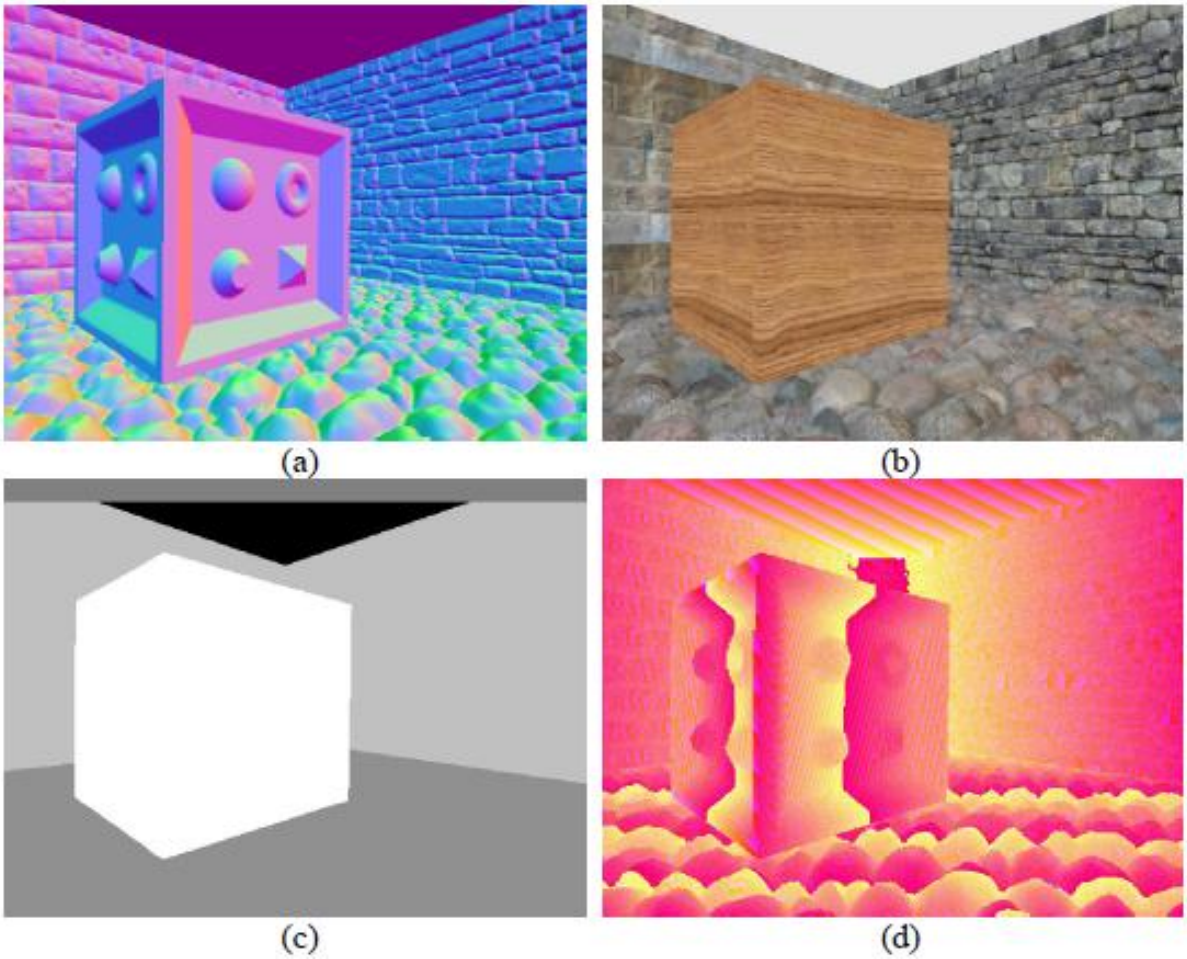
3.2.2.1 Beschrijving

Het render proces wordt opgebroken in een aantal fasen. Zie Figuur 1. Het renderen van geometry en alle andere effecten worden effectief losgekoppeld.



Figuur 1. Een overzicht van de fasering van Deferred Shading (Policarpo & Fonseca, 2005).

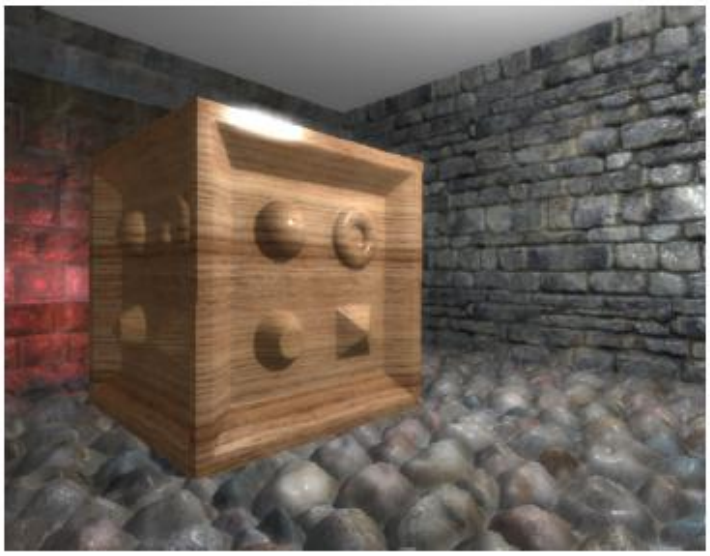
In de Geometry fase wordt de geometrie in de scene gerendered. Door gebruik te maken van MRTs kan er veel informatie per pixel worden opgeslagen, zoals de diffuse en specular components van het materiaal, de depth van de surface in de betreffende pixel, shininess, normals en eventuele andere waarden. Zie Figuur 2.



Figuur 2. Vier texture maps gegenereerd tijdens de Geometry fase: (a) normal, (b) diffuse, (c) specular en (d) encoded depth als color (Policarpo & Fonseca, 2005).

Voor de volgende fase worden de zojuist gevulde MRTs geleegd en opgeslagen in screen-space texture maps. Deze worden vervolgens gebruikt door de shaders in de volgende fasen.

Tijdens de Lighting fase wordt voor elk licht voor elke pixel de lichtberekening uitgevoerd aan de hand van de opgeslagen data. Zo ook voor de Post-Processing fase. In de Final fase worden vervolgens de verschillende resulterende screen-space texture maps gecombineerd om tot het uiteindelijke plaatje te komen. Zie Figuur 3.



Figuur 3. De uiteindelijke output van de Final fase (Policarpo & Fonseca, 2005).

Op deze manier blijft het renderen van geometry $O(1)$ en staat het berekenen van lichteffecten daar los van, m.a.w., geometrie hoeft niet nogmaals worden gerendered omdat de informatie al opgeslagen is.

3.2.3 Discussie

De meest evidente voordelen van Deferred Shading ten opzichte van Forward Shading zijn duidelijk; logica is losgekoppeld, dus shaders zijn conceptueel eenvoudiger. Dat houdt bijvoorbeeld in dat combinaties van materials en lights niet allemaal uitgewerkt hoeven te worden; één shader per type licht kan al voldoende zijn, aangezien alle informatie die bij de berekening betrokken moet worden al voor handen is. Bij Forward Shading is dat niet het geval aangezien dat per pixel, terwijl de geometrie wordt gerendered, op de juiste manier moet gebeuren afhankelijk van het type licht en het type material. De moeilijkheidsgraad van implementatie van Deferred Shading is daarom lager dan die van Forward Shading.

Een ander groot voordeel van Deferred Shading is dat de upper bound van de complixiteit vele malen voorspelbaarder is. Immers; bijna alle berekeningen vinden plaats op een per-pixel basis. Bij Forward Shading zijn er ook veel gevallen die per-vertex plaats vinden. Beide technieken kunnen geoptimaliseerd worden door middel van o.a. depth-sorting, culling en screen-space scissor rectangles.

Misschien wel het meest belangrijke voordeel van Deferred Shading is dat er een grote hoeveelheid lichten kan worden toegepast op de scene aangezien alle geometrie maar één keer hoeft worden gerendered, waar dat bij Forward Shading al gauw een grote stijging in complexiteit betekent. Vooral voor straatverlichting in steden tijdens de nacht zal dit een groot voordeel zijn in SketchaWorld.

Het grootste nadeel van Deferred Shading is het onvermogen om met transparante objecten om te gaan. Aangezien er per pixel maar over één surface point informatie kan worden opgeslagen is het onmogelijk om om te gaan met transparante objecten. Een oplossing hiervoor zou zijn om eerst alle opaque objecten te renderen, en daarna volgens het Forward Rendering principe de transparante objecten eroverheen te renderen. Bij veel transparante objecten is dit ongewenst vanwege de hoge

complexiteit. In SketchaWorld zullen weinig transparante objecten worden geplaatst, dus we achtten dit geen groot probleem.

Een ander nadeel van Deferred Shading is de grote hoeveelheid geheugen die nodig is. Stel dat er een spel wordt gespeeld op een resolutie van 1024x768. Stel ook dat we in de texture-maps een 4-byte float opslaan voor depth, vier 4-byte floats voor normal data en vier 2-byte floats voor kleur. Voor elke pixel is dan $4 + 4 * 4 + 4 * 2 = 28$ bytes benodigd. Gevolg; per frame zijn er $28 * 1024 * 768 = 22020096$ bytes oftewel 22 Mbs benodigd. Stel dat de speler op een high quality resolutie gaat spelen van bijvoorbeeld 1920x1200, dan loopt dat getal al op tot 65 Mbs! Om deze reden scoort Deferred Shading slecht op externe benodigdheden.

Voor SketchaWorld zullen er voornamelijk in de steden veel lichten worden toegepast (bijv. nachtelijke verlichting van de straten). Deferred Shading zal dan vele malen efficienter om kunnen gaan met de hoeveelheid lichten. Anderzijds kun je je afvragen of het ook efficienter is bij een grote terrain map met slechts de zon. Forward Shading zal hier efficienter te werk kunnen gaan omdat er minder werk hoeft worden verzet; aan de andere kant zal Deferred Shading niet (veel) trager gaan werken dan bij de nachtelijke stadverlichting, die Forward Shading niet aan zou kunnen. Om deze reden geven we Deferred Shading een hogere beoordeling voor doeltreffendheid.

3.2.4 Conclusie

Onderstaande tabel geeft een overzicht van de verschillende methoden en in hoeverre deze aan de opgestelde criteria voldoen.

Techniek	**Moeilijkheidsgraad implementatie**	**Performance**	**Externe benodigdheden**	**Doeltreffendheid**
Forward Shading	-/+	-/+	+	-/+
Deferred Shading	+	+	--	++

Deferred Shading scoort uiteindelijk het beste op de verschillende criteria, vooral vanwege het feit dat de voordelen van de techniek goed passen bij de doelen die gesteld zijn voor SketchaWorld.

3.3 Dynamic Soft Shadows

Bij aanvang van het project is aangegeven dat schaduwen van hoge prioriteit zijn. De voordelen van schaduwen in een virtuele werkelijkheid zijn dan ook groot (Hasenfratz, Lapierre, N., & Sillion, 2003):

- schaduwen helpen om de relatieve grootte en positie van objecten in een scene te herkennen;
- schaduwen helpen om de geometrie van een complexe occluder te begrijpen;
- en, tenslotte, schaduwen helpen om de geometrie van een complexe receiver te begrijpen.

Er zijn vele manieren om schaduwen te genereren, en veel manieren om schaduwen ‘soft’ te maken. Bij dit project is het belangrijk dat de schaduwen soft zijn, en dat ze realtime kunnen worden gerendered. Daarom hebben we direct de keuze gemaakt voor Shadow Mapping als basis methode voor het genereren van shadows, in de eerste plaats omdat dit de industriestandaard is en ten tweede omdat de meest efficiente soft shadow technieken zijn gebaseerd op Shadow Mapping. We vergelijken daarom slechts de technieken die voortborduren op Shadow Mapping om ze soft te maken.

3.3.1 Type Shadow Map

Een probleem van Shadow Maps is het niveau van detail van de informatie die zij bevatten over een scene. Wanneer een scene met een groot terrein wordt gerendered is de resolutie van de shadow map vaak niet afdoende om overal in het beeld een mooie schaduw te kunnen genereren (Dimitrov, 2007).

Figuur 4. Shadow Map artifacts bij grote terreinen in de scene (Dimitrov, 2007).

Er zijn verscheidene oplossingen voor dit probleem, welke in ShaderX7 (Engel, 2009) worden verdeeld over twee categorieën: split shadow maps en warped shadow maps. Uit beide categorieën hebben we een methode geselecteerd.

3.3.1.1 Cascaded Shadow Maps

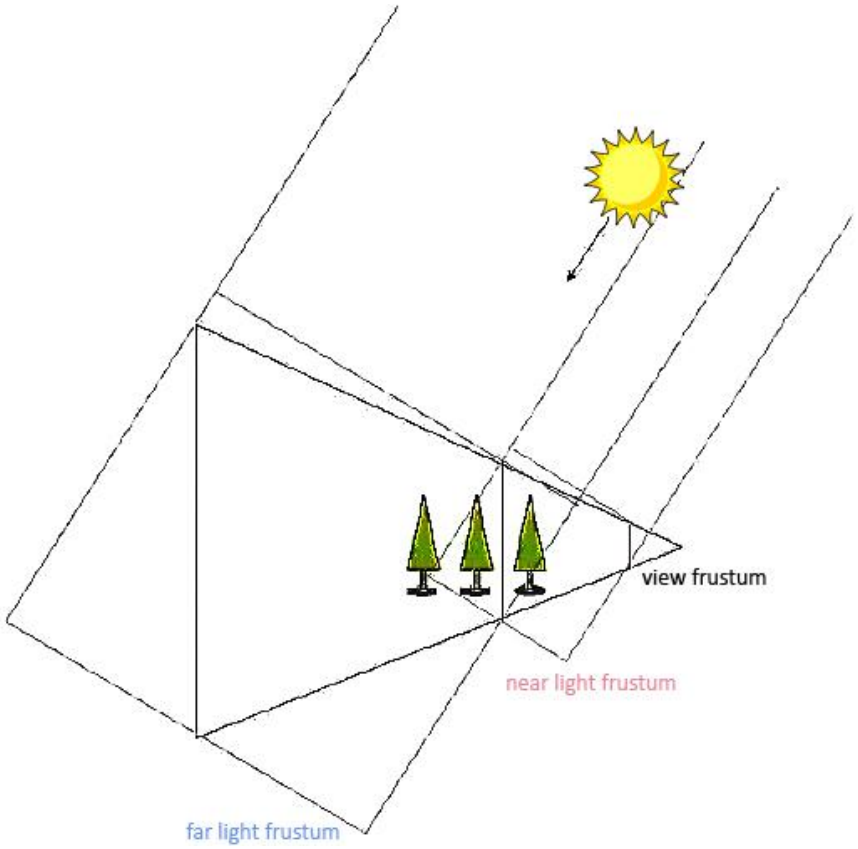
Grote dynamische omgevingen vereisen vaak meer of minder detail in de shadow map afhankelijk van de afstand van de camera tot de schaduw. CSM is een techniek uit de categorie split shadow maps.

3.3.1.1.1 Beschrijving

CSM word meestal toegepast wanneer de zon schaduw werpt over een groot gebied. Omdat het onpraktisch is om één grote shadow map te maken wordt de shadow map opgesplitst in meerdere delen. Een shadow map die alleen nabije objecten bevat, zodat deze schaduwen van hoge kwaliteit kunnen werpen, een shadow map die alleen objecten op grote afstand bevat met lagere resolutie en eventueel nog meerdere maps ertussen. Deze verdeling is redelijk omdat objecten op grotere afstand ook minder pixels in het scherm innemen en dus ook minder informatie in de shadow map nodig hebben (Dimitrov, 2007).

Bij het genereren van de shadow maps wordt rekening gehouden met een goede plaatsing van de splits. Het is de bedoeling dat deze zo worden gepositioneerd dat de overgang van de ene shadow map naar de andere zo min mogelijk opvalt. Hiervoor kunnen verschillende correcties worden toegepast. Nadat de splits zijn bepaald wordt er per view frustum een light frustum parallel aan de

view frustum gegenereerd, waarvoor dan de cascaded shadow maps worden gegenereerd. Zie Figuur 5.



Figuur 5. De meest rechtse boom wordt in de near-shadow map opgeslagen en de andere twee in de far-shadow map (Dimitrov, 2007).

3.3.1.1.2 Discussie

Ondanks dat het idee van CSM goed klinkt zijn er nog een behoorlijk aantal moeilijkheden waar de techniek mee worstelt. Dit zijn onder andere (Engel, 2009):

- Wisseling schaduw kwaliteit bij splits;
- Opslag;
- Niet-optimale split selectie;
- Filteren over meerdere splits.

ShaderX7 presenteert voor alle bovenstaande problemen een oplossing, en volgens resultaten die ook gepubliceerd zijn in dat boek blijkt CSM veel betere resultaten op te leveren dan een techniek uit de categorie warped shadow maps.

Omdat Cascaded Shadow Mapping veel problemen effectief oplost krijgt het een hoge score voor doeltreffendheid.

De moeilijkheidsgraad is niet bijzonder hoog; er is voldoende stof om een stabiele implementatie te maken van CSM.

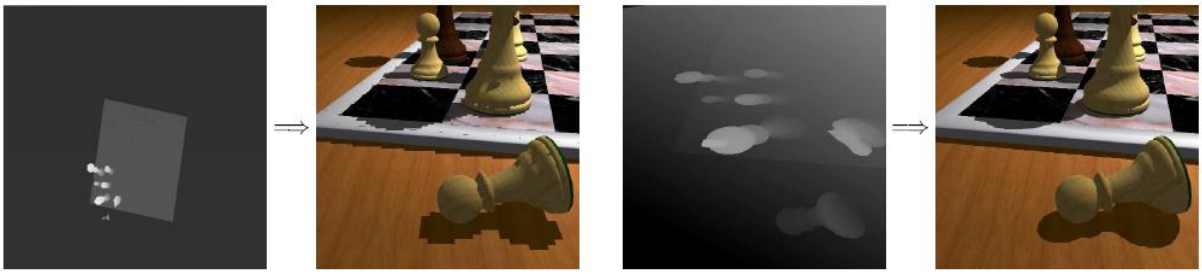
Wat de performance aangaat zijn er wel iets meer moeilijkheden. Er moeten meerdere maps worden gegenereerd en in het geheugen worden gehouden. De lookups zijn ook iets complexer dan bij enkele shadow maps. Daarom ook een slecht cijfer voor externe benodigdheden.

3.3.1.2 Perspective Shadow Maps

Een heel andere manier om met het probleem van slecht verdeeld niveau van kwaliteit bij shadow maps om te gaan is door de shadow map te warpen.

3.3.1.2.1 Beschrijving

PSM's worden pas gegenereerd in perspective-space, in plaats van world-space. Op deze manier zijn ze automatisch beter aangepast aan de eye-space en krijgen objecten die dichterbij zijn meer detail omdat ze groter in de shadow map worden gerendered.



Figuur 6. (Links) Uniform 512x512 shadow map en de resulterende afbeelding. (Rechts) Dezelfde scene met een perspective shadow map van dezelfde afmetingen (Stamminger & Drettakis, 2002).

3.3.1.2.2 Discussie

Wanneer de scene erg groot en/of dynamisch wordt kan het detail van de shadow map alsnog problemen opleveren. Daarnaast neemt de kwaliteit van een PSM ook af wanneer de kijkrichting parallel is aan de lichtrichting (Stamminger & Drettakis, 2002). Andere developers claimen (uit ervaring) dat de kwaliteit van PSM's uiteindelijk inferieur is aan die van CSM's (Stone, 2010).

Omdat de wiskunde achter deze warping techniek best wel complex is, geven we deze methode een slechte score voor de moeilijkheidsgraad van de implementatie.

Qua performance is het positief; er komt slechts 1 shadow map uit en er hoeft ook maar 1 stappenplan worden doorlopen.

De externe benodigdheden zijn niet bijzonder afwijkend van traditioneel shadow mapping.

De doeltreffendheid is positief; maar niet in alle gevallen. Dit vanwege het probleem met de parallele kijkrichting en de lichtrichting.

3.3.1.3 Conclusie

Techniek	Moeilijkheidsgraad implementatie	Performance	Externe benodigdheden	Doeltreffendheid
Cascaded Shadow Maps	+	+/-	-	++
Perspective Shadow Maps	-	+	+\-	+

Uit de tabel kan worden geconcludeerd dat CSM een efficientere oplossing zou zijn. Bovendien worstelt het niet met de (onoplosbare) problemen die PSM heeft. CSM is gemakkelijk te gebruiken in combinatie met andere technieken ook omdat de maps niet gewarped zijn.

3.3.2 Soft Shadows

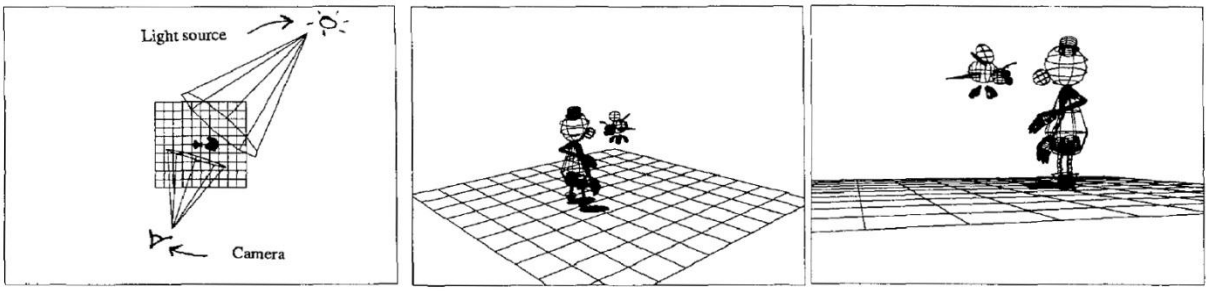
Omdat we ons hebben gericht op Shadow Maps in de voorgaande onderdelen, zullen wij ons nu richten op uitbreidingen op Shadow Map algoritmes die Soft Shadows genereren. Op deze manier minimaliseren we de overhead die ontstaat door de combinatie van het Type Shadow Map en de Soft Shadows.

3.3.2.1 Percentage Closer Filtering

Standaard shadow mapping maakt gebruik van een depth map gerendered vanuit het licht. De schaduwen die geproduceerd worden zijn hard omdat de vergelijking van dieptes in de depth map en die gezien vanuit de camera altijd 0 of 1 oplevert. Percentage Closer Filtering (PCF) maakt gebruik van een sampling technique waarmee aliasing (harde shadows) wordt geelimineerd, en soft shadows worden gegenereerd die lijken op penumbrae.

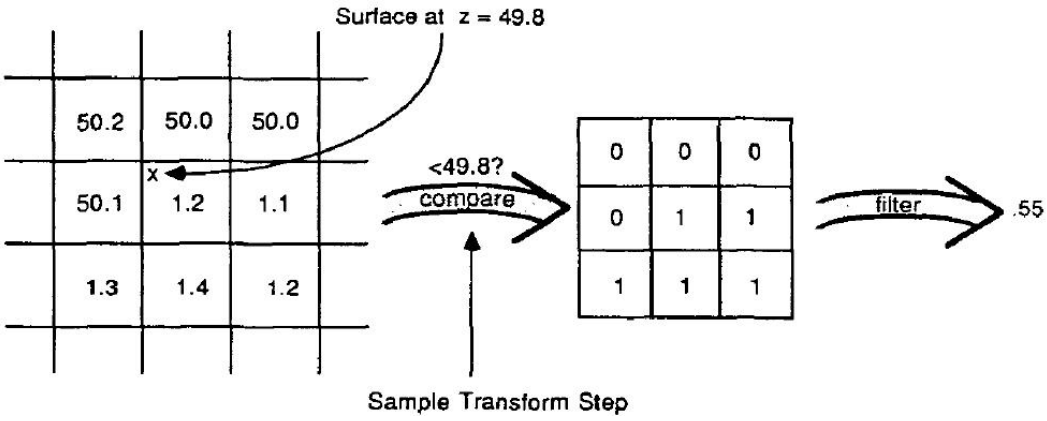
3.3.2.1.1 Beschrijving

Het algoritme bestaat uit twee stappen. Eerst wordt op de traditionele manier een depth map gerendered, gezien vanuit de lichtbron. In de tweede stap wordt de scene gerendered vanuit de camera. Zie Figuur 7.



Figuur 7. (1) De scene. (2) View vanuit de lichtbron (depth map). (3) View vanuit de camera (eye space) (Reeves, Salesin, & Cook, 1987).

Voor elke pixel wordt berekend waar de depth map gesampled moet worden om de diepte op te halen. Er wordt dan een blok met waardes, om de gevonden coordinaten heen, uitgelezen uit de depth map, en vergeleken met de diepte die vanuit eye space gezien wordt. Het resultaat van de vergelijking $depth_{depth\ map} < depth_{eye\ space}$ wordt in een tijdelijke map opgeslagen. Daarvan wordt vervolgens het gemiddelde genomen om te bepalen hoe transparant de schaduw moet zijn. Zie Figuur 8.



Figuur 8. Links de depth map, rechts de percentage closer filtering (Reeves, Salesin, & Cook, 1987). Te zien is dat de 9 waardes binnen de filter region worden vergeleken met 49.8. Van de resulterende waarden wordt vervolgens het gemiddelde genomen.

3.3.2.1.2 Discussie

Het is duidelijk dat PCF eenvoudig toe te passen is na implementatie van traditioneel shadow mappen. Er moet slechts een filter stap worden toegevoegd aan het proces.

Het proces voegt ook veel toe aan de kwaliteit van de schaduwen. Het is eenvoudig en effectief. Ook al zijn de schaduwranden geen echte penumbrae (ze zijn niet afhankelijk van de relatieve afstanden tussen objecten en de lichtbron), ze lijken er wel op (Reeves, Salesin, & Cook, 1987).

Wat betreft externe benodigdheden is er dus weinig te melden; het principe werkt zoals gewone shadow mapping werkt. Er is een klein stukje extra geheugen nodig om steeds de filterwaarden in te bewaren, maar dat is niet significant.

De performance echter is niet optimaal. De kwaliteit en de performance zijn sterk afhankelijk van elkaar. De grootte van het filter bepaalt grotendeels de kwaliteit van de schaduwranden, en ook de hoeveelheid berekeningen die gedaan moeten worden. Vanwege deze directe afhankelijkheid scoort PCF slecht op performance. Zie Figuur 9.

(a) (b)

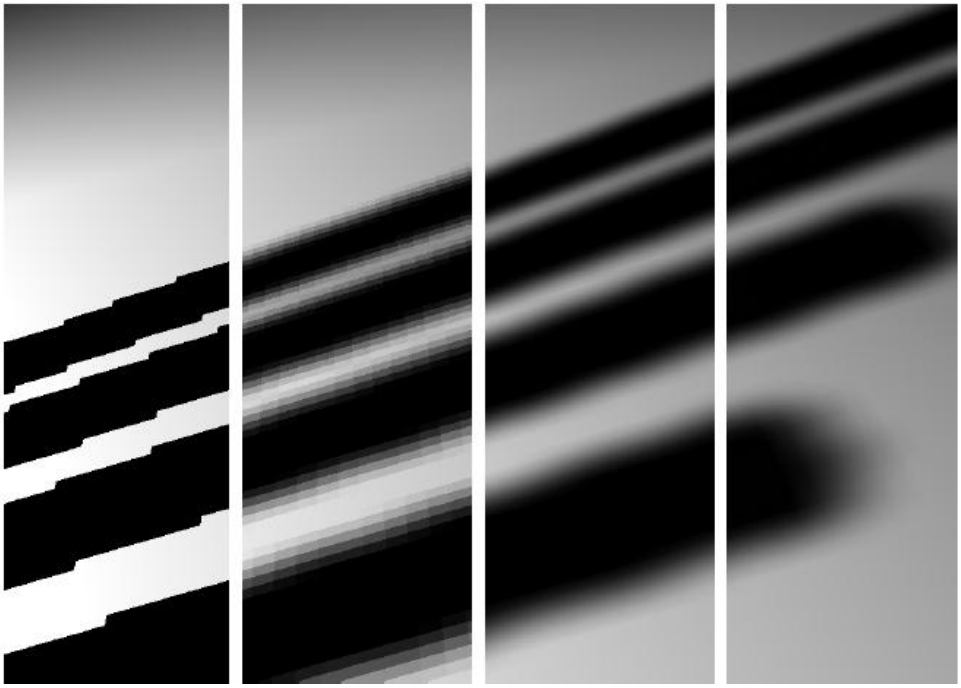
Figuur 9. Percentage closer filtering met een (a) 1x1 sample filter per pixel en met een (b) 4x4 sample filter per pixel (Bunnell, 2004).

3.3.2.2 Variance Shadow Maps

Variance Shadow Maps is net als PCF een variatie op standaard shadow mapping.

3.3.2.2.1 Beschrijving

Waar bij Shadow Mapping de diepte voor een pixel wordt opgeslagen, wordt bij VSM per pixel het gemiddelde en het gemiddelde in het kwadraat opgeslagen van een distributie van dieptes. Daaruit kan vervolgens efficient een bovengrens worden berekend door middel van Chebychev's ongelijkheid. Die bovengrens is dan een goede benadering van de fractie van de pixel die in schaduw is (Donnelly & Lauritzen, 2006).

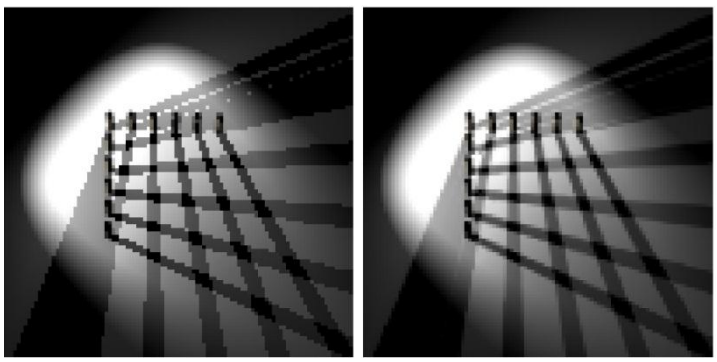


Figuur 10. Van links naar rechts: 1) traditioneel shadow mapping, 2) 5x5 percentage closer filtering, 3) 5x5 bilineair percentage closer filtering, 4) variance shadow maps met een 5x5 ontkoppelbare gaussian blur (Donnelly & Lauritzen, 2006).

3.3.2.2.2 Discussie

Variance Shadow Mapping is overtuigend sneller dan Percentage Closer Filtering omdat er geen blok filter over de depth map gaat. De berekening is ook veel eenvoudiger. Wat betreft geheugengebruik is er ook weinig verschil met gewoon Shadow Mapping.

Echter is het ook niet zonder problemen. VSM heeft zonder pre- en post-filters zoals mipmapping en anisotropic filtering ook nog redelijk veel aliasing. Zie Figuur 11.



Figuur 11. Variance Shadow Mapping zonder (links) en met (rechts) mipmapping (Donnelly & Lauritzen, 2006).

Daarnaast zijn er ook light bleeding problemen. Zie Figuur 12.

Figuur 12. Light bleeding treedt op wanneer de variantie over een gefilterd gebied hoog blijkt te zijn (Donnelly & Lauritzen, 2006).

Het lijkt erop dat Variance Shadow Mapping efficienter is dan PCF, maar pas echt tot zijn recht komt wanneer er nog meer technieken aan worden toegevoegd zoals mipmapping. Ookal is de moeilijkheidsgraad niet erg hoog, we beoordelen VSM er toch slecht voor omdat er veel extra werk bij komt.

3.3.2.3 Percentage Closer Soft Shadows

De laatste techniek van dit onderdeel die we bespreken is Percentage Closer Soft Shadows (PCSS). Deze techniek is een uitbreiding op PCF, die ook te combineren is met VSM.

3.3.2.3.1 Beschrijving

PCSS is een nieuwe techniek uit 2005 (Fernando, 2005), gebaseerd op Shadow Mapping en PCF. PCSS heeft de volgende karakteristieke eigenschappen (citaat uit (Fernando, 2005)):

- Genereert perceptueel accurate soft shadows;
- Gebruikt een enkele lichtbron sample (één shadow map);
- Vereist geen pre-processing, post-processing, of additionele geometrie;
- Vervangt naadloos een traditionele shadow map lookup (heeft dezelfde voordelen als traditioneel shadow mapping – onafhankelijk van scene complexity, werkt met alpha testing, displacement mapping, enz.);
- Draait in real-time op huidige consumer graphics hardware.

Ten opzichte van PCF valt vooral op dat de schaduwen niet overal even soft zijn. Dit geeft een veel realistischer effect. Zie Figuur 13.

Figuur 13. Percentage Closer Soft Shadows in actie (Fernando, 2005).

Er hoeven alleen wat wijzigingen aan een standaard shadow mapping shader worden aangebracht om PCSS te implementeren.

Net als PCF worden er floating-point waardes gegenereerd per pixel in eye view die bepalen in hoeverre die pixel in schaduw is. Daarnaast gebruikt PCSS ook een lichtgrootte. Het grote verschil met PCF is dat PCSS geen vaste PCF kernel grootte gebruikt, maar deze varieert op een intelligente manier om de juiste graad van softness te bereiken. Om dat doel te bereiken gebruikt PCSS het volgende stappenplan (Fernando, 2005):

1. **Blocker search**: De Shadow Map wordt doorzocht en de dieptes die dichterbij de lichtbron zijn dan het punt dat wordt geshade (de receiver) wordt gemiddeld over een bepaalde regio. De grootte van die regio is afhankelijk van de grootte van de lichtbron en de receiver's afstand van de lichtbron.
2. **Penumbra schatting**:Door middel van parallel planes approximation worden de breedte van het penumbra geschat, gebruik makend van de lichtgrootte en de afstanden tot het licht van de blocker en de receiver: $w_{Penumbra} = (d_{Receiver} - d_{Blocker}) \cdot w_{Light} / d_{Blocker}$
3. **Filtering**: Als laatste stop wordt een standaard PCF stap toegepast op de shadow map gebruik makend van een filter size proportioneel aan de penumbra schatting van stap 2.

3.3.2.3.2 Discussie

Aangezien de functie net iets uitgebreider is dan PCF geven we PCSS een positieve score voor de moeilijkheidsgraad van de implementatie. De stappen zijn vrijwel hetzelfde, op een paar extra stapjes na.

Qua performance is PCSS waarschijnlijk een stuk beter; de PCF kernel size wordt gevarieerd afhankelijk van de omvang van de penumbra, dus wanneer er geen schaduw is zal de kernel size minimaal (1x1) zijn. Zo wordt veel onnodig samplen voorkomen.

Externe benodigdheden zijn net als bij de andere methoden minimaal.

Qua doeltreffendheid is deze methode waarschijnlijk het meest exact van de drie, dus daarvoor krijgt PCSS de maximale score.

3.3.2.4 Conclusie

Techniek	Moeilijkheidsgraad implementatie	Performance	Externe benodigdheden	Doeltreffendheid
Percentage Closer Filtering	++	-	++	+
Variance Shadow Maps	-	+	++	+
Percentage Closer Soft Shadows	+	+	++	++

Uit de tabel valt snel te concluderen dat PCSS de beste techniek zou zijn. Wat ook logisch is aangezien het de mooiste schaduwen genereert en ook vrij optimaal te werk gaat. Daarnaast is het ook nog eens vrij eenvoudig te implementeren (ook in combinatie met CSM).

3.4 Advanced mapping techniques

Veel oppervlakken in de werkelijke wereld bevatten kleine oneffenheden en hobbels. Het is niet efficient om deze details te verwerken in het drie dimensionale model van het oppervlak omdat hierdoor het aantal benodigde vertices drastisch zou toenemen. Om oppervlakken die in het drie dimensionale model vlak zijn toch een oneffen structuur te laten hebben zijn meerdere technieken bedacht.

Eerst wordt de standaard normal mapping techniek besproken en daarna twee uitbreidingen die ten koste van performance het resultaat nog verder verbeteren.

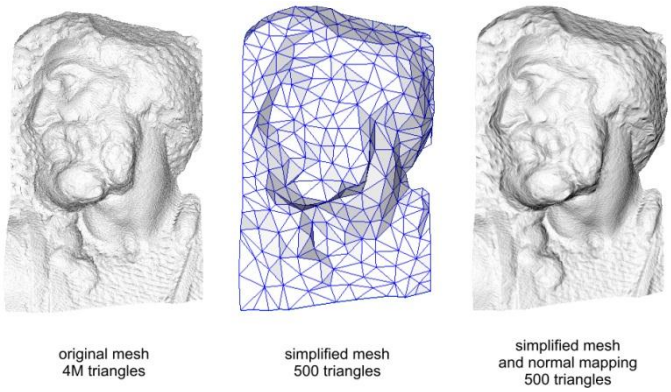
3.4.1 Normal mapping

3.4.1.1 Beschrijving

Bij het gebruik van normal mapping moet er een normal map texture aanwezig zijn. Deze texture wordt voor elke pixel op het model gesampled om de bijbehorende normal vector te vinden. Dit in tegenstelling tot het renderen zonder normal mapping waar de normal bepaald wordt door de triangle waarin de pixel zich bevindt.

Als de normal is gesampled dan wordt deze vermenigvuldigd met de light direction om de licht intensiteit te bepalen.

De onderstaande afbeelding illustreert hoe het gebruik van normal mapping het aantal benodigde triangles drastisch kan verminderen zonder merkbaar kwaliteits verlies in het eindresultaat. Het model wordt gereduceerd van 4 miljoen naar 500 triangles, dit is een factor 8000.



Figuur 14. Illustratie polygoon reductie zonder kwaliteitsverlies met normal mapping.

3.4.1.2 Discussie

Normal mapping is een techniek die veel extra realism toevoegd en een relatief kleine performance impact heeft. Het is een techniek die tegenwoordig erg veel gebruikt wordt en het valt niet te verwachten dat de implementatie ervan voor grote problemen zal zorgen.

Het belangrijkste nadeel is dat voor het toepassen van normal mapping er bijpassende normal maps beschikbaar moeten zijn. Deze normal maps moeten ook goed aansluiten op de al bestaande color maps.

3.4.2 Parallax mapping

3.4.2.1 Beschrijving

Een toevoeging op normal mapping is de parallax mapping techniek. Hierbij wordt extra realisme toegevoegd door de schijn van diepte te creeeren. Dit wordt bereikt door het verschuiven van de texture coordinaten. Hoeveel en in welke richting de texture coordinaten verschoven moeten worden is afhankelijk van de camera positie en de diepte die wordt gesampled uit de heightmap.

Figuur 15. Schijn van diepte gecreëerd door het toepassen van parallax mapping.

3.4.2.2 Discussie

Parallax mapping vereist meer rekenkracht dan normal mapping maar resulteert ook in een hoger mate van realiteit. De afgelopen jaren is de beschikbare rekenkracht sterk toegenomen en wordt er steeds vaker gekozen parallax mapping toe te passen.

Voor parallax mapping is geen normal map nodig maar een heightmap, het is echter vrij eenvoudig een normal map naar heightmap te converteren en vice versa. Wat betreft benodigde resources is deze techniek dus niet significant veeleisender dan normal mapping.

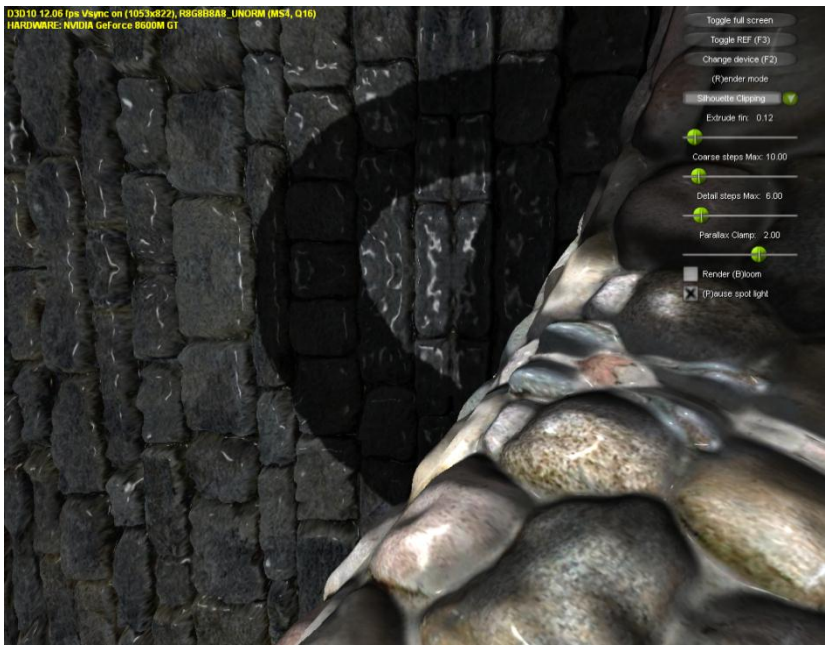
3.4.3 Relief and parallax occlusion mapping

3.4.3.1 Beschrijving

Zowel normalmapping als parallax mapping hebben als nadeel dat er geen rekening wordt gehouden met effecten zoals lokale verhulling en lokale schaduw. Ook het sillouhette van het object kan niet worden aangepast aan de informatie uit de heightmap.

Relief en parallax occlusion mapping lossen dit probleem op door via een iteratief ray tracing process voor elke pixel het intersection punt te bepalen. Dit resulteert in een erg realistische weergave waarin afhankelijk van de kijkhoek bepaalde oneffenheden elkaar kunnen verhullen en schaduwen.

Een extra toevoeging hierop is ook in staat het silouhet van het object aan te passen. Hierdoor wordt een extra mate van realiteit bereikt waardoor het resultaat in feite niet meer te onderscheiden is van een sterk getrianguleerd model.



Figuur 16. Voorbeeld van een silouhet clipping shader.

3.4.3.2 Discussie

Relief en parallax occlusion mapping zijn in staat dezelfde kwaliteit te bereiken als een hooggedetailleerd getrianguleerd object. Dit gaat echter gepaard met een hoge performance cost. Het is echter mogelijk een afweging te makken tussen kwaliteit en performance door het aantal stappen in het ray tracing process te verminderen.

Wat betreft benodigde resources is deze techniek vergelijkbaar met parallax mapping. Het opstellen van een efficiente implementatie zal echter lastiger zijn.

3.4.4 Conclusie

Er kan worden geconcludeerd dat er veel verschillende methoden zijn voor het visualiseren van oneffenheden in vlakke geometrie. De onderstaande tabel geeft een overzicht van de verschillende methoden en in hoeverre deze aan de opgestelde criteria voldoen.

Techniek	**Moeilijkheidsgraad implementatie**	**Performance**	**Externe benodigdheden**	**Doeltreffendheid**
Normal mapping	++	++	-/+	-/+
Parallax mapping	-/+	-/+	-/+	+
Relief en parallax occlusion mapping	-	--	-/+	++

Parallax mapping lijkt een goede middeweg tussen de mindere kwaliteit van normal mapping en de lage performance van relief en parallax occlusion mapping. Het feit dat de meeste rendering engines van de huidige generatie ook parallax mapping implementeren is een indicatie dat dit inderdaad een goede keuze zou kunnen zijn.

3.5 Sky and Atmosphere

In elke openluchtomgeving op aarde is er vaak een deel van de lucht te zien. Gegeven een tijd, positie op aarde en weercondities kan er nagegaan worden hoe de zichtbare lucht eruit moet zien. Om in een real-time renderer dit te kunnen simuleren wordt er vaak gebruik gemaakt van skyboxes of skyspheres (skymodel), die bekleed worden met een texture. De intentie van een skymodel is om een weergave van objecten mogelijk te maken die effectief oneindig ver weg zijn. Voor een texture van een skymodel is dan een map nodig van de omgeving die weergegeven moet worden, dat kan in de vorm van fotos van een real life omgeving of het werk van een artist.

Op het moment dat de omgeving dynamisch wordt, zoals variabele weercondities, tijd en planeetcoördinaten is een statische texture al snel geen optie meer. De real-time renderer moet dan apart rekening houden met wolkendek, dag-nacht transities, een sterrenhemel en de positie van andere hemellichamen, afhankelijk van de hoeveelheid realisme en detail er vereist wordt van de renderer.

In dit hoofdstuk zullen verscheidene technieken en ideeën aan bod komen, die een bijdrage kunnen hebben aan het renderen van skymodels en atmosferische effecten in een real-time renderer.

3.5.1 Layered Sky

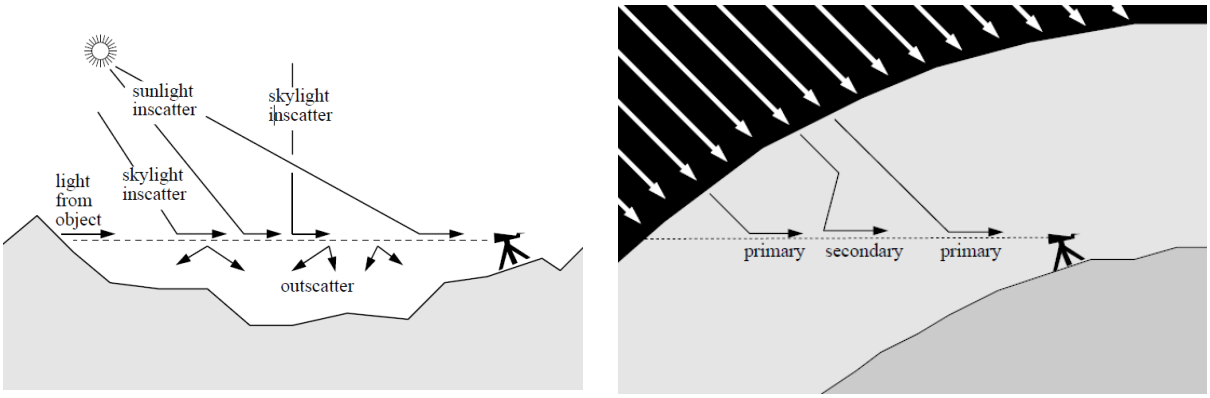
Een mogelijke techniek die de dynamiek van een bewegende sky kan simuleren is door gebruik te maken van layering. Een eenvoudige representatie is door blauwe lucht, wolken en sterren in aparte textures te hebben. Met een layering van deze textures, door ofwel verschillende skymodels of texture blending is het mogelijk om een eenvoudige weergave te maken van hemellichamen. Deze kunnen dan gedraaid of displaced worden afhankelijk van de invoer van een tijd van een dag.

Wolken kunnen benaderd worden als een texture met alpha waardes die voor een skymodel kan zitten.

3.5.2 Atmospheric Scattering

Een realistische weergave van de lucht zal op een moment rekening moeten houden met atmospheric scattering. Dit is het verschijnsel dat licht van de zon door de atmosfeer moet komen en

onderweg reageert met deeltjes in de lucht. Vaak is dit te zien als een fog-achtig effect voor objecten die beduidend ver weg staan van de camera, en ook voor de kleurveranderingen van de lucht die het duidelijkst merkbaar zijn tijdens de schemering. Veel technieken zijn voorgesteld om scattering in real time te simuleren (Preetham, 1999), waarvan de onderstaande de meest haalbaar lijkt.



Figuur 17: Scattering principe (Preetham, 1999).

Om zichtbare lucht realistisch te renderen op een methode die atmospheric scattering moet simuleren, kan er gebruik gemaakt worden van metingen of formules die gebaseerd zijn op waarnemingen in de wereld. Per pixel kan dit dan berekend worden op basis van de hoek van de pixel met de horizon en de positie van de zon.

Om ditzelfde toe te passen op objecten die ver weg staan is er een stap nodig die de gerenderde meshes per pixel gaat blenden met de berekende atmosfeerwaarde op die pixel.

3.5.3 Discussie

Gegeven de bovengenoemde technieken zijn er de volgende mogelijkheden. Het gebruiken van scattering is een heel effectieve manier om de zon, de lucht en verre objecten realistisch te laten lijken. Daarbij moet wel gezegd worden dat op zichzelf scattering heel kaal is, waarbij in een realistische situatie er minstens ook wolken, sterren en een maan zijn. Sterren zijn in principe, uitgaande van een aardse setting, makkelijk te tonen door middel van een texture gebaseerd op de coordinaten van sterren. De texture hoeft dan verder niet dynamisch te zijn, zelfs als de simulatie de lengte- en breedtegraad kan varieren, de texture hoeft slechts te draaien om een voorbepaalde as.

De maan is niet per se veel moeilijker om realistisch weer te geven. Er zal rekening gehouden moeten worden met de belichting van de maan, die afhangt van de stand van de zon. Een mogelijke implementatie om dit voor elkaar te krijgen is om het zichtbare maanoppervlak in een texture te hebben, met ook een normal map om zo de belichting accuraat te krijgen. Dit is dan wel afhankelijk van dat de maan kan bewegen over de hemel. Voor groter realisme kunnen er statistieken raadgepleegd worden voor de goede positie gegeven een tijdstip. Een realistische eclips mogelijk maken valt hier buiten de scope, omdat dit een zeldzaam voorkomen is en mogelijk de formules voor scattering beïnvloeden.

Renderen van wolken is een apart probleem vanwege de variatie, zoals mist, hoge wolken en lage wolken. Meerdere lagen van gegenereerde wolk- textures is een mogelijkheid die voor hoge wolken waarschijnlijk realistisch genoeg over komt. Het wordt lastiger als laaghangende wolken gesimuleerd moeten worden. Mist achtige effecten kunnen beter apart behandeld worden, dit komt terug in het hoofdstuk Rain effects. Voor alle wolken is het vaak wel wenselijk om zonlicht goed te laten reageren

afhankelijk van de dikte van het wolkendek. Creatief gebruik van texture mapping, waarbij de dichtheid van wolken voor een pixel wordt aangegeven is hier een mogelijkheid.

3.5.4 Conclusie

Effecten va een layered sky zijn relatief eenvoudig te realiseren. Dit omvat dan een sterrenhemel, eenvoudige wolken en de maan. Er moet wel rekening gehouden worden met de textures of statistische informatie die nodig is. Atmospheric scattering houdt vooral rekening met de effecten die de zon heeft op de lucht, en is dus nodig voor een overtuigende schemering en voor overdag. Er kan gebruik gemaakt worden van bekende formules voor de kleuring van de lucht gegeven bekende invoer.

Techniek	**Moeilijkheidsgraad implementatie**	**Performance**	**Externe benodigdheden**	**Doeltreffendheid**
Layered Sky	++	+	-/+	-/+
Atmospheric scattering	-	-/+	+	++

Beide technieken representeren een ander doel en kunnen elkaar niet echt vervangen in een dynamische wereld. Een realistische weergave zal dan beide moeten implementeren. Mits goed uitgevoerd zal het wel een sterk gevoel van realisme kunnen weergeven.

3.6 High Dynamic Range Rendering (HDRR)

Een image is beperkt in de informatie die het kan weergeven. Dit stamt van dat de data die per pixel opgeslagen wordt niet een groot bereik heeft, met tegenwoordig normaal gesproken 8 bits per kleur. Vergeleken met een persoon die de echte wereld waarneemt is dit niet realistisch. Het menselijk oog is in staat om zowel heel belichte en heel donkere omgevingen waar te nemen. De techniek om dit vermogen te simuleren voor realtime renderers wordt High dynamic range rendering genoemd. Met HDRR wordt t mogelijk om een grotere range van waardes op te slaan per pixel.

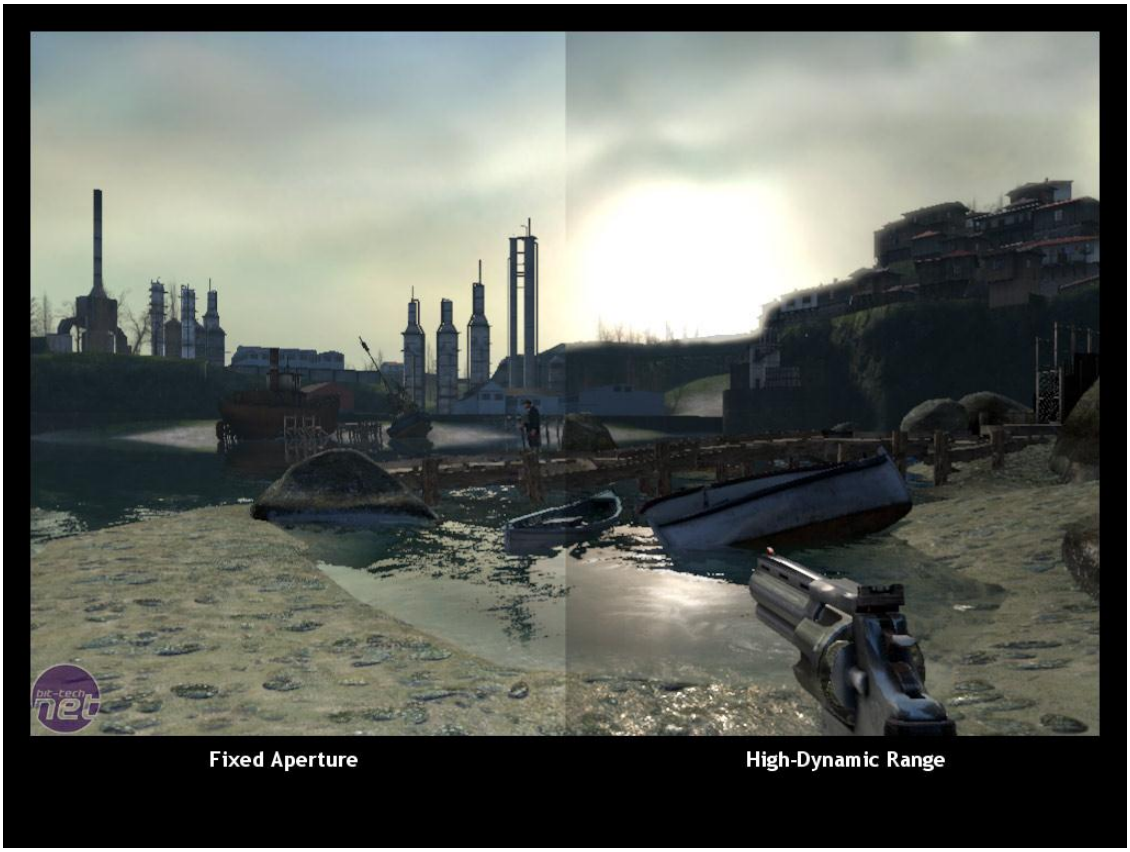
HDRR is afhankelijk van een paar stappen om het gewenste effect te krijgen. De weer te geven informatie moet eerst renderen naar een HDR-capable imageformaat. Vervolgens moeten de gerenderde waardes op het beeldscherm komen op een dusdanige manier dat de range niet verloren gaat. Dit gebeurt met tone mapping. Naast deze nodige stappen kunnen er ook nog filters gebruikt worden die niet strikt nodig zijn, maar wel het realisme van de image verbeteren, zoals HDR-bloom.

Het gebruik van HDRR is in het algemeen duur om te doen op oudere hardware, maar gezien de hardware waarmee we rekening moeten houden niet hieronder valt is het gebruik niet uit te sluiten in sketchaworld. Daarnaast is HDRR een nodige techniek voor de realistische weergave van een serie van andere effecten, zoals atmospheric scattering. In het hoofstuk Rain Effects word een directe toepassing van bloom uiteengezet, dat op zijn beurt ook HDRR nodig heeft voor de uiteindelijke weergave. De core HDRR technieken zullen in dit hoofdstuk in groter detail besproken worden.

3.6.1 Image format

Bij het gebruik van HDRR is het nodig om aan te geven wat voor informatie aangegeven kan worden per pixel. De laagste requirement is om meer dan 8 bits per kleur per pixel op te slaan voordat het toepassen van HDRR relevant wordt. Als transparante objecten ook gerenderd moeten worden in dezelfde pass, vereist dit ook dat de alpha channel bits een deel moeten uitmaken van de pixel. Er

bestaat een mogelijkheid om toch gebruik te maken van van een 32bit per pixel format, door 10 bits per kleur te hebben, en de resterende 2 voor de alpha. Het is wel meer gebruikelijk om 64bit per pixel op te slaan, met 16 bits per component. Het verschil hiertussen is vrij eenduidig: hoe meer bits per pixel er berekend moeten worden, hoe realistischer het uiteindelijke beeld potentieel is, maar dit gaat ten koste van de snelheid. In het ideale geval zijn beide opties beschikbaar.



Figuur 18: Vergelijking van een Half-Life 2 scene gerendered zonder HDR en met HDR (Richards, 2005).

3.6.2 Tone mapping

Tone mapping is nodig om de HDR informatie weer te geven op het scherm. Hoe dit precies gebeurt kan varieren, door de vele effecten die gerealiseerd kunnen worden als dit op een specifieke manier benaderd wordt. Een heel envoudige manier om tone mapping te implementeren is om alle pixels lineair te mappen. Het maximum en minimum van de HDR image zullen mappen naar maximum en minimum van het scherm, met tussenliggende waardes lineair geinterpoleerd. Zit zou als effect hebben dat veel informatie verloren gaat voornamelijk in beelden die voornamelijk donker of licht zijn.

Een betere manier is om visual adaption te simuleren, aanpassen aan het weergegeven licht. Dit komt dichter overeen met wat realistisch zou zijn, want het menselijk oog gedraagt zich zo ook. Dit vereist wel dat de waardes van het beeld gesampled moeten worden om te kijken welke range rerenderd moet worden. Er kan uitgegaan worden van de gemiddelde lichtintensiteit van de scene om aan de hand daarvan de tone mapping uit te voeren.

Tone mapping kan ook gebruikt worden om een specifiek effect weer te geven, zoals night vision. Dit kan gezien worden als een kleurfilter te gebruiken gevolgd door een tone mapping op een lage range. Dit is uiteraard beperkt tot specifieke situaties.

3.6.3 Conclusie

Het gebruik van HDR is een eenvoudige keuze, het is nodig om veel licht effecten te realiseren. Een basisvorm van HDR is in ieder geval eenvoudig te implementeren. Voor een effect waarbij het minste informatie verloren gaat, of waarbij het zo realistisch mogelijk is ligt het iets minder voor de hand.

Techniek	**Moeilijkheidsgraad implementatie**	**Performance**	**Externe benodigdheden**	**Doeltreffendheid**
Image format R10G10B10A2	++	++	++	-/+
Image format 16b per channel	++	+	++	++
Linear tone mapping	++	+	++	-
Adaptive tone mapping	+	+	++	++
Situational tone mapping	-/+	-/+	-/+	+

Het gebruik van een 16 bit per pixel image format heeft een beter effect op de kwaliteit van het uiteindelijke beeld, en is dit dan ook de eerste keuze. Het gebruik van een compacter image format levert misschien wel een performance voordeel, dus kan het handig zijn om dit te kunnen kiezen in de applicatie.

Voor de doeltreffendheid van adaptive tone mapping, het zoeken naar de light range die weergegeven moet worden, is het de beste keuze in algemene situaties. Het gebruik van linear tone mapping geeft gewoon niet het gewenste effect, door voorgenoemde redenen. Situational tone mapping zoals dat bij night vision denkbaar is, moet meer gezien worden als een mindere prioriteit vanwege de beperkte toepassing.

3.7 Rain effects

Spellen die regen als visuele optie bevatten benaderen dit vaak op een versimpelde manier. Veel spellen beperken zich bij de weergave van regen tot het tonen van vallende druppels en uiteenspat effecten op de grond. De mate van geloofwaardigheid van deze aanpak wordt in dit project niet genoeg geacht. Voor regen om geloofwaardig over te komen is het nodig dat de invloeden van de regen op de gehele omgeving duidelijk zichtbaar zijn.

Teneinde dit resultaat te bereiken wordt het onderzoeksgebied van realistische regenweergave onderverdeeld in twee gebieden. Als eerste worden technieken besproken die regendruppels tonen. Vervolgens worden technieken besproken die de visuele indruk geven van vochtige oppervlaktes en hoge luchtvochtigheid als gevolg van de regen.

3.7.1 Rain drops

Traditioneel worden regendruppels in virtuele werelden vaak gesimuleerd met primitieve particle systems (Tariq, 2007). In deze subcategorie wordt als eerste een variatie op traditionele particle systemen besproken. Vervolgens wordt een techniek besproken die gebruik maakt van een geometry shader, wat mogelijk is sinds DirectX 10. Ten slotte wordt een post-processing aanpak besproken die regendruppels over de scene heen rendered.

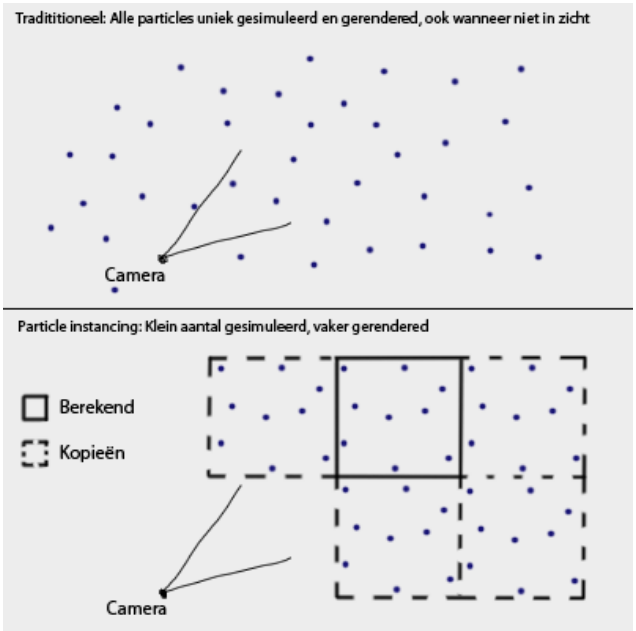
3.7.1.1 Particle instancing

Particle instancing is een verbetering op primitieve particle systems. In deze paragraaf wordt besproken hoe particle instancing kan worden gebruikt om efficiënt vele regendruppels weer te geven.

3.7.1.1.1 Beschrijving

In een virtuele omgeving waarin gebruik wordt gemaakt van traditionele particle systems zou elke regendruppel worden gerepresenteerd door een uniek particle in de wereld. Elk frame zou dan voor elke particle de nieuwe positie van die particle moeten worden bepaald, ook als deze niet zichtbaar is. Het resultaat is dat er meer particles worden gesimuleerd dan er gerendered worden. (Engel, 2009)

Particle instancing beperkt het aantal nodige particles dat gesimuleerd hoeft te worden. Bij particle instancing worden slechts particles in een compact kubusvorming gebied vòòr de camera gesimuleerd. De particles in dit gebied worden gekopieerd en dan in naastliggende kubusvormige gebieden nogmaals gerendered, maar alleen als die gebieden voor de camera zichtbaar zijn. Als in het oorspronkelijke gebied een particle de kubus verlaat wordt deze aan de andere zijde van het gebied opnieuw geplaatst. Omdat de kubus direct meerdere malen naast zichzelf wordt gerendered lijkt zo´n particle vloeiend van één gebied naar de ander te gaan. Figuur 11 demonstreert deze techniek.



Figuur 19. Het verschil tussen tradionele particle systems en particle instancing.

3.7.1.1.2 Discussie

Het voornaamste voordeel van deze techniek is dat het aantal nodige particle simulaties sterk is verminderd. Een ander voordeel is dat het meermaals renderen van een particle volledig op de GPU kan gebeuren. Dynamische omgevingsinvloeden zoals wind en zwaartekracht kunnen nog steeds worden toegepast, door deze als parameters aan de GPU te geven.

De nadelen van deze techniek hebben direct te maken met het feit dat particles niet meer uniek zijn. Het particle systeem is nu een globaal effect geworden, het is lastig om het in een bepaald gedeelte van het zicht te laten regenen en in een ander gedeelte niet. Om aan te geven waar het wel en niet zou moeten regenen, bijvoorbeeld dat het niet moet regenen onder een afdak of in een gebouw, zou er bijvoorbeeld gebruik moeten worden gemaakt van occlusion height maps die per vlakte aangeven op welke hoogte regendruppels moeten stoppen. Deze kunnen voorbereid of eenmalig gegenereerd worden, maar nemen wel veel geheugen in beslag.

3.7.1.2 Rain using geometry shader

Het geometry shader programma model is geïntroduceerd in DirectX 10. Geometry shaders worden gebruikt om nieuwe grafische primitieven te genereren uit de aangeleverde primitieven. Een manier om regendruppels met behulp van een geometry shader is besproken in Tariq (Tariq, 2007).

3.7.1.2.1 Beschrijving

Tariq maakt gebruik van een geometry shader om vertex-posities, die locaties van regeldruppels voorstellen, op de GPU om te zetten naar sprites. Behalve de vertex-posities moeten de positie van de camera, en de positie van één lichtbron, worden meegeleverd. Deze posities worden gebruikt om, op een versimpelde manier, de texture van een regendruppel te kiezen uit een database van textures. Deze texture toont de juiste lichtinval, kloppend met de positie van de aangeleverde lichtbron en de kijkhoek van de beschouwer. (Tariq, 2007) Een demonstratie hiervan is te zien in Figuur 20.

Figuur 20. Bij het renderen van regendruppels wordt rekening gehouden met lichtinval (Tariq, 2007).

3.7.1.2.2 Discussie

Een voordeel van deze techniek is dat de CPU weinig informatie hoeft te leveren, maar er toch gedetailleerde resultaten uitkomen. De geometry shader zorgt er zelf voor dat de lichtinval per regendruppel goed getoond wordt.

Verder wordt door Tariq een optimalisatiemethode toegepast: er wordt gebruik gemaakt van twee vertex buffers. De output van de geometry shader wordt opgeslagen in één van de buffers. In deze working buffer worden de vertices ook geanimeerd. De output van deze bewerkingen wordt gestreamed naar de andere buffer om te renderen. De volgende keer dat de geometry shader wordt aangeroepen wordt de functie van de twee vertex buffers omgewisseld. Omdat de twee buffers niet opnieuw geinitialiseerd hoeven te worden wordt er hierop tijd bespaard.

3.7.1.3 Layered rain

Tatarchuk (Tatarchuk, 2006) beschrijft een post-processing methode die het visuele effect van meerdere lagen van regen over de scene heen tekent.

3.7.1.3.1 Beschrijving

Tatarchuk combineert het idee van een scrolling texture met het idee van een individuele particle die aangetast wordt door wind en zwaartekracht. Op basis van de vectoren die de windkracht en zwaartekracht voorstellen wordt er gesampled uit een statische texture. Om voor beweging van de druppels te zorgen wordt er bij het samplen een offset toegepast ten opzichte van de vorige frame. Dit laatste is vergelijkbaar met hoe scrolling textures worden geimplementeerd. Het verschil met scrolling textures is echter dat de wind- en zwaartekracht invloed hebben op de draaiing van de getekende regendruppels, en de grootte van de offset die elk frame wordt toegepast.

In een full-screen pass wordt eerst een quad aangemaakt waarop, met de bovenstaande aanpak, meerdere lagen van regen worden getekend. Elke laag van regen toont een andere diepte. Uiteindelijk wordt het resultaat geblend met de scene die al gerendered was. (Tatarchuk, 2006)

3.7.1.3.2 Discussie

De regen wordt op het einde over het huidige beeld heen getekend, de berekende regendruppels zullen dus altijd zichtbaar zijn. Dit zorgt voor twee belangrijke nadelen. Het eerste nadeel is dat regen niet specifiek in een gebied kan worden getekend.

Het tweede nadeel is dat er geen gebruik wordt gemaakt van de depth buffer. Als er een object dicht in de buurt van de toeschouwer staat, worden verre regendruppels alsnog over het object heen getekend.

Het voornaamste voordeel is dat het weinig rekenkracht kost om deze methode uit te voeren, juist omdat er geen gebruik wordt gemaakt van tussenstappen die bepalen of een druppel zichtbaar moet zijn of niet. Doordat er meerdere lagen van regendruppels worden getekend treedt er toch een realistisch diepte-effect op. Dit wordt versterkt door de mogelijkheid om verschillende wind- en zwaartekracht parameters in te stellen voor elke aparte laag.

3.7.2 Humidity

Er worden nu technieken besproken die de gevolgen van regen tonen. Er wordt ingegaan op manieren om te bepalen welke oppervlakten nat moeten worden weergegeven. Ook wordt er ingegaan op technieken die de indruk geven van hoge luchtvochtigheid. Bij dit onderdeel moet in het

achterhoofd worden genomen dat er sprake is van verschillende resultaten en het daarom wellicht mogelijk zal zijn om meerdere technieken tegelijk te selecteren.

3.7.2.1 Wetness maps

3.7.2.1.1 Beschrijving

Wetness maps kunnen gebruikt worden om aan te geven welke gebieden in de wereld een hoge vochtigheidsgraad hebben. Dit is bijvoorbeeld het geval als het regent of als er in een rivier of ocean in de buurt is.

Als in de shader, via deze wetness maps, bekend is dat het materiaal nat gerenderd moet worden dan kan hier rekening mee worden gehouden. Twee belangrijke eigenschappen van nat materiaal ten opzichte van droog materiaal is dat het vaak donkerder is en een hogere specular factor heeft. Deze beide eigenschappen zijn zeer eenvoudig en goedkoop te implementeren. Een andere belangrijke eigenschap van nat materiaal is dat het een weerspiegelend effect krijgt. Dit voegt veel realisme toe maar kost ook een significante hoeveelheid aan performance.

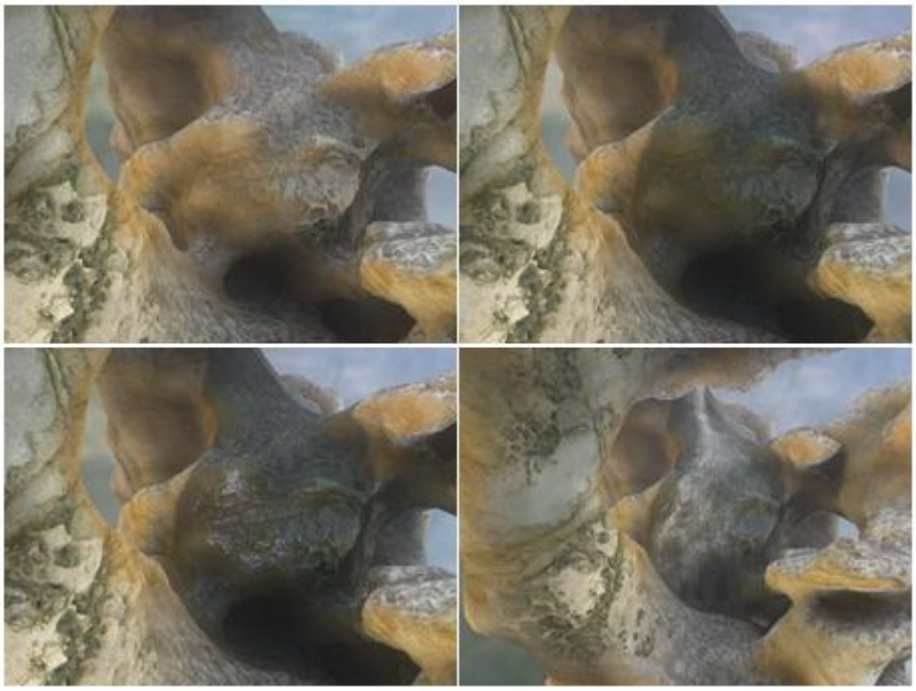
3.7.2.1.2 Discussie

Aanpassingen in de shader om materiaal nat te laten lijken voegen veel toe aan een regenachtige scene. Zeker het donkerder maken van het materiaal en het verhogen van de specular factor zijn erg goedkope technieken die al een overtuigende werking hebben.

Bij het toevoegen van reflecties moet er rekening gehouden worden met de hoge kosten qua performance. Hierbij moet wel weer worden opgemerkt dat, ondanks de extra pass benodigd voor de reflection, de performance kosten kunnen worden beperkt door de reflection naar een render target met een veel lagere resolutie te renderen.

Het gebruik van wetness maps kan helpen bij het markeren van de natte gebieden in de wereld. Hierbij is het belangrijk om te kiezen voor een juiste resolutie zodat er van een goede tradeoff tussen geheugen gebruik en kwaliteit.

Het alternatief voor het gebruik van wetness maps is het instellen van wetness door een globale constante. Hierdoor kan er nog wel onderscheidt worden gemaakt tussen nat en droog materiaal, maar is het niet mogelijk hier lokale verschillen in te introduceren.



Figuur 21 Links boven het orginele materiaal, rechtsboven is het materiaal donkerder gemaakt, linksonder is de specular verhoogt, rechts onder is het eindresultaat te zien met particle effects toegevoegd.

3.7.2.2 Kawase's bloom filter

Kawase's bloom filter is een post-processing methode, die het inkomende beeld in meerdere passes blurred en het resultaat vervolgens blend met het oorspronkelijke beeld.

3.7.2.2.1 Beschrijving

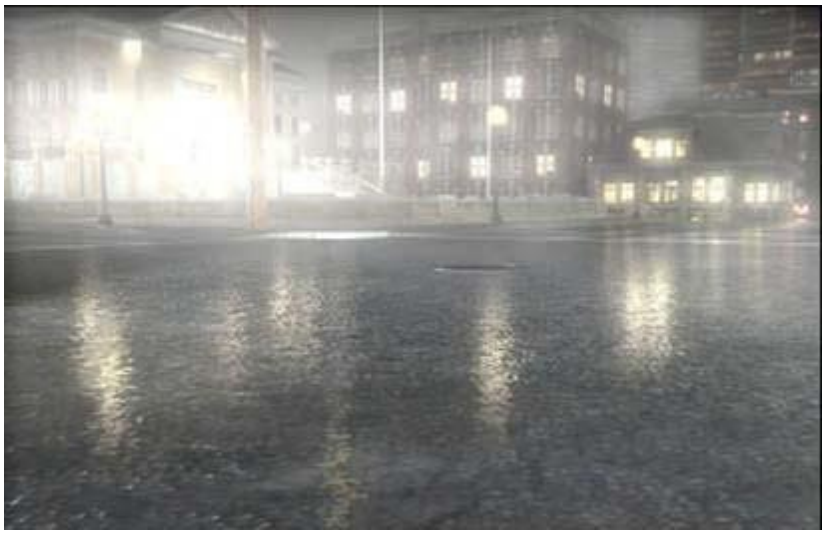
Bij Kawase's Bloom Filter wordt In elke pass gebruik gemaakt van een Gaussian blur filter, waarbij de omliggende sampling points per pass steeds verder worden gekozen van de pixel die berekend wordt. Hierdoor is het invloedbereik van elke pixel op zijn omliggende pixels proportioneel aan het aantal passes waarvoor gekozen is. (Tatarchuk, 2006)

3.7.2.2.2 Discussie

In een regenachtige omgeving komt het natuurlijk over als lichtbronnen een sterke gloed (bloom) afgeven. De verklaring hiervoor is dat bij hoge luchtvochtigheid er veel waterdeeltjes in de lucht hangen die ervoor zorgen dat lichtstralen worden verstrooid.

Kawase's bloom filter kan gebruikt worden om dit effect na te bootsen. Door een relatief hoog aantal passes te kiezen wordt het effect van pixels met sterke intensiteit, meestal afkomstig van sterk belichte oppervlakken, naar omliggende pixels uitgesmeerd. In het eindresultaat is dan een sterke gloed te zien, wat het gevoel geeft van hoge luchtvochtigheid. In figuur 7 is een voorbeeld te zien waarin Kawase's bloom filter met een overdreven aantal passes is gebruikt.

Een nadeel van deze methode is dat het effect niet te isoleren is. Dit is een algemeen nadeel van post-processing effecten. Het is niet mogelijk om bijvoorbeeld in een bepaald gedeelte van het beeld sterke bloom te hebben, terwijl een ander gedeelte scherp gehouden wordt. Dit is niet een groot nadeel, omdat een bepaalde mate van blur juist toevoegt aan het realisme van het beeld.



Figuur 22. Demonstratie van Kawase's bloom filter in de ToyShop demo van Ati (Tatarchuk, 2006).

3.7.3 Conclusie

Het onderzoeksgebied “rain effects” wijkt af van de andere onderzoeksgebieden, omdat realistische regen een abstracte wens is. In dit gedeelte zijn technieken behandeld die op verschillende manieren bijdragen aan het realistisch tonen van regen. Hierdoor is er, in tegenstelling tot de meeste andere behandelde onderwerpen in dit verslag, veel mogelijkheid tot combineren van methoden.

In Tabel 1 staan de sterke en zwakke kanten van de behandelde technieken uitgezet volgens de eerder gekozen criteria.

Tabel 1. Waardering van de behandelde methode via de eerder genoemde criteria.

Techniek (Rain drops)	Moeilijkheidsgraad implementatie	Performance	Externe benodigdheden	Doeltreffendheid
Particle instancing	+	+	++	+/-
Rain using geometry shader	++	+	++	++
Layered rain	++	++	+	-

Techniek (Humidity)	Moeilijkheidsgraad implementatie	Performance	Externe benodigdheden	Doeltreffendheid
Wetness maps	-	-	++	+
Kawase’s bloom filter	++	++	++	+

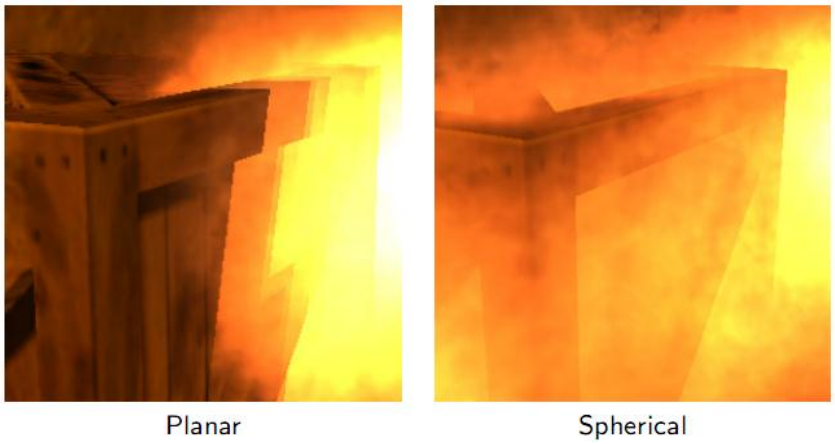
3.8 Volumetric fog

3.8.1 Spherical billboards

Veel games maken gebruik van planar billboards om doorzichtige volumes weer te geven, zonder dat dit veel rekenkracht vereist (Umenhoffer, Szirmay-Kalos, & Szijártó, 2006). Bij planar billboards treden er snel clipping artifacts op, wanneer een billboard snijdt met geometrie. Spherical billboards vormen een variant die kan worden gebruikt om clipping artifacts te verminderen, terwijl de vereist rekenkracht niet veel groeit.

3.8.1.1 Beschrijving

Een nadeel van planar billboards is dat wanneer dit vlakte andere geometrie doorsnijdt, er een ongewenst visueel effect optreedt genaamd clipping artifacts. Er is een abrupte overgang van het gas dat gesimuleerd wordt, naar de geometrie waarmee het snijdt. Een voorbeeld hiervan is te zien in Figuur 23.



Figuur 23. Vergelijking tussen planar en spherical billboards (Umenhoffer, Szirmay-Kalos, & Szijártó, 2006).

Spherical billboards maken gebruik van een bolvormig model dat in de 3D wereld geplaatst is, om de texture op te renderen. De texture wordt alleen op het oppervlakte getekend. Met spherical billboards treden clipping artifacts minder vaak op, omdat als men een bol bekijkt, de afstand naar het boloppervlak toeneemt naarmate men van het midden van de bol afgaat. Dit zorgt voor een smoothing effect in sommige situaties waarbij er bij planar billboards harde clipping optrad.

3.8.1.2 Discussie

Spherical billboards lijken wel voordeel te bieden boven planar billboards, terwijl het niet veel extra kost qua performance of implementatielast. Het voordeel hangt af van hoe de spherical billboard het andere stuk geometrie snijdt, in sommige gevallen is er helemaal geen verbetering.

Een ander voordeel van spherical billboards is dat het makkelijk is om collision toe te passen op een bolvormig model. Dit kan gebruikt worden om bijvoorbeeld rook interactie te laten hebben met de omgeving. Een groot rook volume kan worden opgebouwd in kleinere spherical billboards. Met toepassing van physics op bolvormige vormen kan het rook dan uit elkaar worden geduwd.

Een nadeel van spherical billboards, wat zwaar telt met betrekking tot realisme, is dat er geen rekening wordt gehouden met de dikte die gezien wordt. De dikte hangt volledig af van de texture; als er een object zich in de sphere bevindt, is de waargenomen dikte van het gas niet minder. Een ander nadeel is dat de camera zich niet in de sphere mag bevinden. Wanneer dat wel gebeurd ziet men de texture niet.

3.8.2 Ray-Traced Fog Volumes

Er kan ook gebruik worden gemaakt van complexere modellen dan planes en spheres om een fog volume te representeren. Een manier om dit te doen is ray-traced for volumes.

Bij ray-traced fog volumes wordt een volume afgebakend door een gesloten polygon-model. In tegenstelling tot billboards worden er geen textures op de faces van dit model getekend. In plaats daarvan worden de faces gebruikt om te berekenen hoe diep de camera in het volume kan zien.

Het berekenen van hoe diep een camera in het volume kan kijken gebeurt in twee passes: In de eerste pass worden de afstanden van de camera naar alle front facing polygons gesommeerd. In de tweede pass gebeurt hetzelfde, maar dan van alle back facing polygons. Het tweede resultaat wordt van het eerste resultaat afgetrokken, waarmee men de zichtbare diepte heeft berekend. De diepte wordt dan gebruikt om een dikte te bepalen voor de te renderen pixel.

3.8.2.1 Discussie

Deze methode is goed te combineren met deferred rendering. In de depth buffer, die vooraf berekend is bij deferred shading, is per pixel bekend wat de afstand is naar het eerste ondoorzichtige object. Deze waarde kan worden gebruikt om te zorgen dat het gedeelte van het mist-model dat zich achter dit punt bevindt, niet meegenomen wordt in het berekenen van de zichtbare diepte. Al deze operaties zijn op de GPU uit te voeren.

Het nadeel is dat ray tracing in het algemeen relatief duur is. Het proces van het berekenen van de mistdikte moet voor elke pixel gebeuren. Er kan werk worden bespaard door dit te laten gebeuren op een lagere resolutie en dan naar boven te schalen middels bijvoorbeeld bilinear filtering.

Ten slotte heerst er de vraag of het wel de moeite waard is om complexe modellen te ondersteunen. Polygonale modellen moeten op een manier gedefinieerd worden. Dit moet door een artist gedaan worden, of procedureel. In elk geval kost het meer moeite dan simpele vormen zoals planar en spherical billboards.

3.8.3 Analytic single scattering model

Sun (Sun) stelt een fog model voor waarin er rekening wordt gehouden met single scattering. Single scattering is het fenomeen dat lichtstralen door deeltjes worden gebroken. In een mistige omgeving zul je daarom een gloed zien van lichtbronnen.

3.8.3.1 Beschrijving

Het doel van het model van Sun is vergelijkbaar met het doel van Kawase's bloom filter, maar dit model benadert het doel op een realistischere wijze. Het model wordt geimplementeerd als een pixel shader. Sun berekent eenmalig een aantal 2D lookup tables met numerieke integratiewaarden van *single scattering light transport equations*. Vervolgens worden real-time slechts de fysieke variabelen aangeleverd.

De variabelen die aangeleverd worden zijn:

- De locatie van de toeschouwer
- Locatie van lichtbronnen en de intensiteit en kleur per bron.
- Diffuse- en specularwaarden van de geometrie die correspondeert met de pixel

Met behulp van de aangeleverde variabelen worden er waarden uit de lookup tables opgevraagd, en ingevoerd in formules die de airlight integral benaderen (Sun). Het resultaat is dat er mist in de scene kan worden gerenderd waarin lichtbronnen, specular reflection en diffuse reflection niet lineair worden uitgedoofd maar natuurgetrouw worden benaderd. Een vergelijking is te zien in Figuur 24.

Figuur 24. Links: OpenGL distance fog. Rechts: Analytic single scattering model (Sun).

3.8.3.2 Discussie

Figuur 24 toont gewenste resultaten met betrekking tot realisme. Het eerste wat opvalt is dat reflectieve oppervlakken die dichtbij de camera staan wel worden gedimd (rechts), dit is in tegenstelling tot de lineaire aanpak van distance fog (links). Ook de gloed van de lichtbron ziet er realistischer uit dan wat in dit geval bereikt zou worden met Kawase's bloom filter.

Het model is het krachtigst in donkere omgevingen met sterke lichtbronnen. In SketchaWorld zou dit vooral zijn in een nachtomgeving met lantaarnpalen. Een voorbeeld van deze situatie is te zien in Figuur 25.

Figuur 25. Analytic single scattering model in een donkere straatomgeving (Sun).

De haalbaarheid van deze techniek is een twijfelgeval. In Sun is pseudocode voor de methode gegeven, maar we durven nog niet te zeggen hoe moeilijk het is om deze te implementeren. Het model is toepasbaar voor alle vormen van mist. Het is dus ook toepasbaar in overdag-situaties, als realistische vorm van distance fog. Dit resultaat is echter ook al bereikt via atmospheric scattering, wat in Sectie 3.5.2 is behandeld. De beste resultaten van dit model zijn in beperkte situaties echt zichtbaar.

3.8.4 Conclusie

Er zijn drie vormen van volumetric fog behandeld. Spherical billboards zijn het goedkoopst qua rekenkracht, maar tonen beperkte resultaten. Ray-traced fog volumes daarentegen zijn krachtig en ondersteunen alle willekeurige polygonale modellen, maar vereisen veel rekenkracht. Het analytic single scattering model toont erg realistische resultaten waarbij rekening wordt gehouden met lichtbronnen en reflectie, maar zou meer tijd vergen om te begrijpen en om te implementeren. In Tabel 2 staat een overzicht van de waardering.

Tabel 2. Waardering van volumetric fog technieken.

Techniek (Rain drops)	Moeilijkheidsgraad implementatie	Performance	Externe benodigdheden	Doeltreffendheid
Spherical Billboards	++	++	++	-
Ray-Traced Fog Volumes	+	-	-	++
Analytic Single Scattering Model	-	+/-	++	++

3.9 Indirect Global Illumination

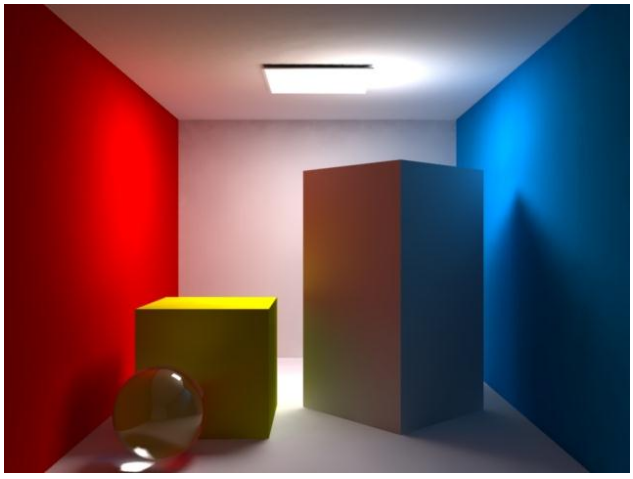
Behalve het licht dat via directe inval de scene beïnvloedt is er ook licht dat via diffuse weerkaatsing de scene belicht. Alle technieken die deze effecten, of delen hiervan, proberen te benaderen vallen onder de categorie indirect illumination. Het belangrijkste verschil met directe illumination is dat er niet meer één enkele lichtbron is. Bij indirect illumination is alle omliggende geometrie van invloed op de belichting, hierdoor wordt vaak de term *global* gebruikt.

Omdat bij indirect global illumination technieken rekening gehouden moet worden met alle omliggende geometrie is het niet eenvoudig dit real-time toe te passen. Er zijn echter verschillende mogelijkheden waarmee bepaalde global illumination effecten real-time benaderd kunnen worden.

3.9.1 Baked maps

3.9.1.1 Beschrijving

Een vaak gebruikte oplossing is het gebruik van lightmaps die van te voren worden gegenereerd. Hierbij worden zeer accurate algoritmes gebruikt om de global illumination effecten door te rekenen. Nadat deze zijn opgesteld is een sample uit de lightmap voldoende om heel realistische resultaten te vekrijgen.



Figuur 26. Voorbeeld van een scene met hoge kwaliteit illumination maps met radiosity en AO.

3.9.1.2 Discussie

Een voordeel van baked lightmaps is dat deze kunnen worden gegenereerd door zeer accurate algoritmes en dus fysisch correct zijn. Daarnaast is ook het feit dat de renderer enkel een simple sample uit de lightmap hoeft te doen een groot voordeel wat betreft de perfomance.

Een erg belangrijk nadeel is echter dat de lightmaps niet gebruikt kunnen worden in dynamische scenes. Als objecten verplaatst worden dan zouden de maps geheel opnieuw gegenereerd moeten worden. Daarnaast heb je voor een realistisch resultaat lightmaps met een hoge resolutie nodig, dit zal dus een impact hebben op het geheugengebruik.

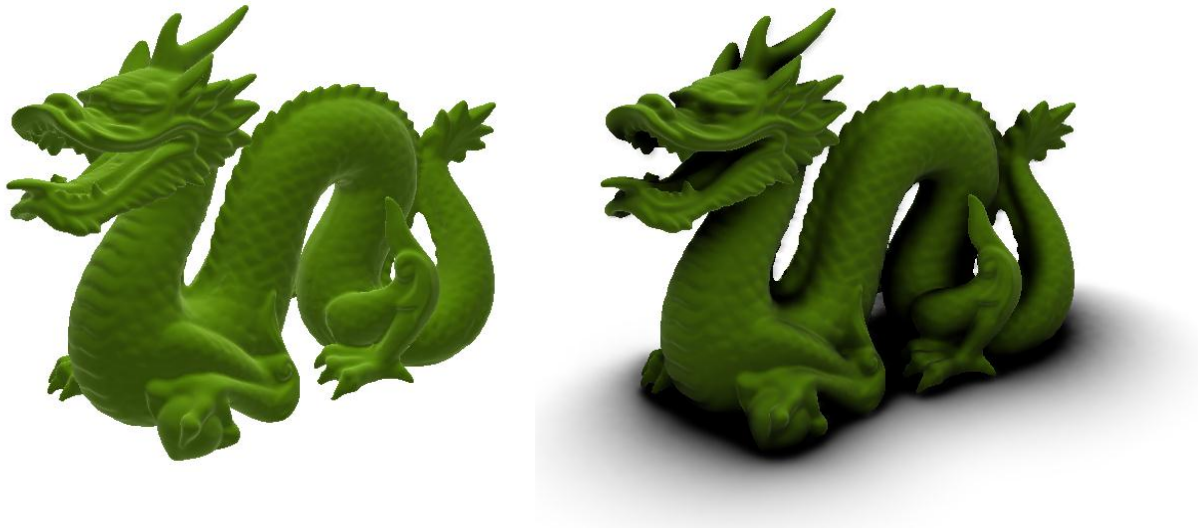
3.9.2 Screen-space ambient occlusion

3.9.2.1 Beschrijving

Screen-space ambient occlusion is een techniek die zich specifiek richt op het berekenen van ambient occlusion. Met andere global illumination effecten zoals bijvoorbeeld radiosity wordt geen rekening gehouden.

De ambient occlusion factor wordt bepaalt met behulp van de depth buffer. Uit deze buffer wordt voor elke pixel meerdere omliggende pixels gesampled. Op basis hiervan kan worden berekend of dit punt in een dal ligt en dus minder belicht moet worden of juist heel open ligt en meer licht ontvangt.

Voor een resultaat met goede kwaliteit is het noodzakelijk dat er voldoende omliggende waardes worden gesampled. Omdat het samplen van veel waardes een performance probleem oplevert wordt er vaak gesampled via een random distributie. Door het samplen op deze random posities ontstaat echter noise in het uiteindelijke resultaat, deze wordt vaak verminderd door het resultaat te blurren.



Figuur 27. Vergelijking tussen zonder SSAO (links) en met SSAO (rechts).

3.9.2.2 Discussie

Screen-space ambient occlusion is een elegante techniek omdat het een redelijke goede benadering kan geven van ambient occlusion zonder de geometrie te moeten analyseren. Omdat de complexiteit van de techniek enkel afhankelijk is van het totale schermoppervlak en niet van de geometrie in world space presteert dit algoritme vaak erg goed.

Nadeel van screen-space ambient occlusion is het feit dat het een benadering van de occlusion factor geeft en hier fouten in kunnen zitten. Occlusion door objecten die niet in beeld zijn worden niet meegenomen in het resultaat. Omdat er maar een beperkt aantal samples gebruikt kan worden hebben objecten die ver weg staan geen invloed meer op de occlusion factor. Daarnaast zorgt het

blurren van het resultaat ervoor dat de occlusion factor kan bleeden naar andere objecten, dit is eventueel te verhelpen door een edge detection filter toe te passen.

3.9.3 Conclusie

De twee belangrijkste nadelen van baked ambient maps ten opzichte van de screen-space ambient occlusion is het niet ondersteunen van dynamische situaties en het gebruik van extra geheugen voor de texture maps. Het voordeel is dat de maps vooraf gegenereerd kunnen worden en de performance niet afhankelijk is van de accuratie maar enkel van de gewenste resolutie.

Techniek	**Moeilijkheidsgraad implementatie**	**Performance**	**Externe benodigdheden**	**Doeltreffendheid**
Baked maps	++	-/+	--	-/+
Screen-space ambient occlusion	-/+	-/+	++	-/+

Het nadeel dat baked maps niet kunnen omgaan met dynamische situaties en van te voren gegenereerd moeten worden weegt zwaar bij het gebruik in een procedureel gegenereerde wereld. Ook het feit dat we te maken hebben met large scale environments maakt het efficient opslaan en in het geheugen laden van de vooraf gegenereerde maps lastig. De voorkeur gaat dus uit naar een screen-space ambient occlusion systeem. Hierbij is bewust een afweging gemaakt om met minder accuratie genoegen te nemen en in ruil daarvoor een dynamischer en eenvoudiger systeem te gebruiken.

3.10 Procedural terrain shading

In large outdoor environments it is important that the terrain is shaded realistically. To achieve this realism the terrain has to be shaded with varied terrain types and must contain enough detail in the texturing when viewing it up close. It is also important that the correct terrain types are placed at the right locations.

We zullen verschillende technieken bespreken die allemaal in meer of minder mate aan de eisen voldoen.

3.10.1 Blend maps

De klassieke en nog steeds veelgebruikte manier van terrain texturing werkt met behulp van blend maps. Deze maps zijn van een relatief lage resolutie en coderen voor elke plek op het terrein het bijbehorende terrein type. Dit betekent dat er in de fragment shader eerst een sample wordt gedaan uit de blend map, op basis van deze sample wordt vastgesteld welk terrain type gerendered moet worden. De uiteindelijke kleur in de output wordt dan gesampled uit de bijbehorende detail texture.

3.10.1.1 Discussie

Het gebruik van blend maps is een relatief eenvoudige oplossing die de artist een redelijke hoeveelheid vrijheid geeft in het plaatsen van terrein types. De techniek is al vaak toegepast en heeft zichzelf bewezen als een werkbare oplossing.

Er zijn per fragment echter wel meerdere samples nodig. Op het moment dat er meer dan vier verschillende terreintypes voorkomen zullen er meerdere blend maps gebruikt moeten worden, of er moet een speciale codering geintroduceerd worden.

3.10.2 Procedural shading

Een alternatief voor het gebruik van blend maps voor het bepalen van het terrein type is procedural shading. In deze techniek wordt het terrain type niet langer uit een blend map gesampled maar deze wordt real-time in de shader berekend uit verschillende terrein eigenschappen.

Veel gebruikte eigenschappen voor het bepalen van het terreintype zijn de hoogte en helling van het terrein. Deze eigenschappen kunnen eenvoudig afgeleidt worden uit de positie en normaal vectoren van de terrein vertices en zijn dus al reeds aanwezig in de shader. Het is echter ook denkbaar om extra factoren mee tenemen in de berekening.

Figuur 28. Terrain dat geheel procedureel wordt getextured.

3.10.2.1 Discussie

Door de recente ontwikkelingen in GPU hardware is de verhouding tussen aritmetische bewerkingen en geheugen operaties veranderd. Het is relatief goedkoop geworden om berekeningen in de shader uit te voeren in tegenstelling tot het samplen van textures. Hierdoor kan procedural terrain shading op moderne hardware een erg efficiente techniek zijn.

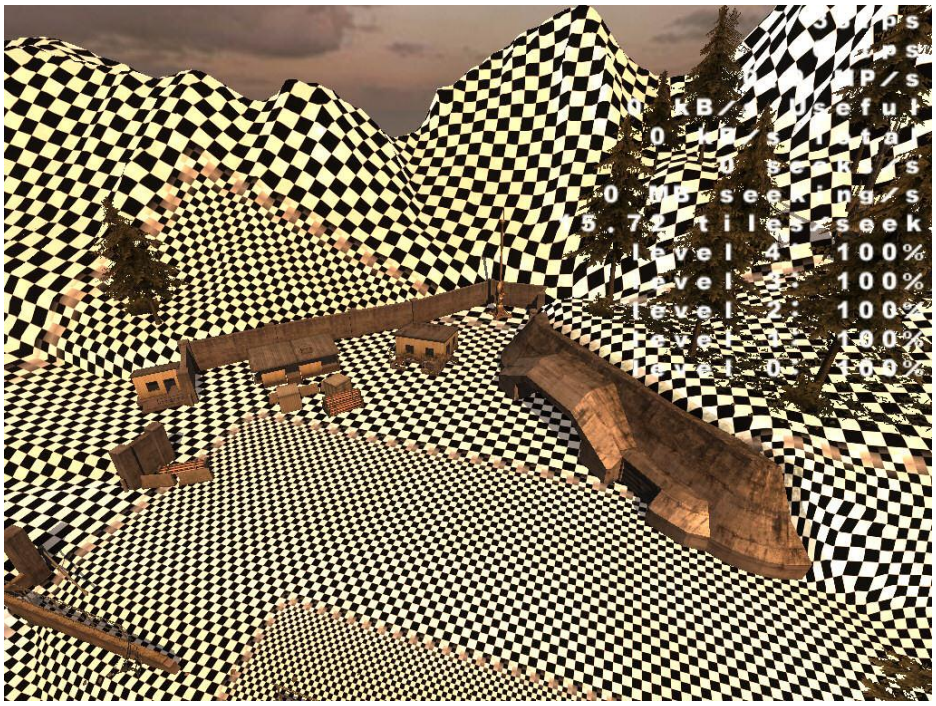
Omdat het terreintype enkel wordt vastgesteld op basis van de terrein eigenschappen is het niet eenvoudig om lokale uitzonderingen op het terrein type te introduceren. Dit limiteert de mogelijkheden en vrijheden van een artist.

3.10.3 Megatexturing

Een techniek met een geheel andere aanpa dan procedural terrain shading is het gebruik van ‘megatextures’. Deze term is geintroduceerd door John Carmack en wordt toegepast in de Doom 3 engine van id Software.

Het idee van megatexturing is gebaseerd op één hele grote texture die over het gehele terrein wordt gelegd. Er wordt dus geen gebruik meer gemaakt van detail textures die over het terrein getiled worden. Omdat deze megatexture echter zo groot is, is het niet mogelijk deze in één keer in het geheugen te laden. In plaats daarvan wordt tijdens het renderen dynamisch bepaalt welke stukken van het terrein meer resolutie nodig hebben, deze worden dan in een queue geplaatst om ingeladen te worden.

Het is niet eenvoudig om te bepalen welke stukken terrein veel screen-space innemen en dus een hogere resolutie texture nodig hebben. Hiervoor kunnen heuristics worden gebruikt maar het is ook mogelijk een aparte render pass hiervoor te introduceren.



Figuur 29 De checkboard texture geeft de resolutie aan die aangevraagd wordt

3.10.3.1 Discussie

Megatexturing is een zeer interessante techniek omdat het de artist in staat stelt het terrein uitermate gevarieerd te texturen. Er bestaat geen afhankelijkheid meer van het gebruik van tiling detail textures.

Er is over de precieze implementatie van megatextures echter niet heel veel literatuur beschikbaar en er bestaat nog discussie over verschillende onderdelen van de techniek. Dit maakt een juiste implementatie ervan complex.

3.10.4 Conclusie

Er kan worden geconcludeerd dat het gebruik van blend maps de meest eenvoudige en in de praktijk bewezen oplossing is. De voordelen die procedural terrain shading biedt zijn echter ook zeer interessant. Het gebruik van geavanceerdere technieken zoals megatexturing heeft zeker voordelen maar introduceert te veel complexiteit in de implementatie om voor ons interessant te zijn.

Onze voorkeur gaat dus uit naar het gebruik van blend maps waarbij op sommige punten gecombineerd gebruik kan worden gemaakt van procedurele eigenschappen. Hierbij kan bovendien worden opgemerkt dat, aangezien het terrein op dit moment al gebruik maakt van blend maps, het nadeel van externe benodigdheden niet meer van toepassing is.

Techniek	Moeilijkheidsgraad implementatie	Performance	Externe benodigdheden	Doeltreffendheid
Blend maps	++	+	-	+
Procedural terrain	-/+	+	++	+

shading				
Megatexturing	--	-/+	--	+

4 Discussie

In dit hoofdstuk wordt er gekeken naar hoe het gezamenlijk gebruik van de gekozen technieken zal werken. In het vorige hoofdstuk is duidelijk geworden hoe individuele technieken toegepast kunnen worden, en is er een afweging gemaakt bij alle technieken hoe deze het best in SketchaWorld te gebruiken zijn. Er wordt in deze discussie in kaart gebracht hoe de gekozen technieken zich tot elkaar verhouden, en wat voor problemen kunnen optreden bij het implementeren en integrereren van de verschillende technieken tezamen.

Als shading method komt Deferred Shading naar voren als de gewenste oplossing. Deze techniek is goed bruikbaar in combinatie met veel gebruikelijke technieken, zoals texture/normal/specular mapping en belichting. Deferred Shading is in staat meer lichtbronnen tegelijk te renderen met een veel beheersbaardere tijdscomplexiteit dan forward shading. Deze eigenschap is gewenst voor SketchaWorld. Deze keuze maakt het gebruik van transparency wel moeilijker en andere technieken maken hier gebruik van. Er zal een extra pass aan het einde van het proces moeten worden toegevoegd voor deze technieken.

Zowel normal mapping als parallex mapping zullen worden geïmplementeerd om zo een goede balans te vinden in performance en kwaliteit. We moeten later nog een afweging maken over welke materialen we daarmee willen shaden. We voorzien geen problemen in verband met deferred shading.

Voor dynamic soft shadows is de voorkeur een combinatie van Percentage Closer Soft Shadows met Cascaded Shadow Maps. Deze technieken kunnen zonder problemen draaien in combinatie met andere technieken, waaronder Deferred Shading.

High dynamic range (HDR) rendering moet geïmplementeerd worden om hogere kwaliteit beelden te krijgen in combinatie met veel van de andere technieken, zoals bij skies en rain, misschien ook bij kunstmatig licht (straatlicht). Het levert geen conflicten op met andere technieken omdat het slechts de surface format beïnvloed en verder een postprocessing effect is.

De sky en atmosphere effecten zullen hoogstwaarschijnlijk prima in de deferred renderer kunnen werken. Deze kunnen als een extra laag in de combinatie pass van de deferred renderer gebruikt worden, waardoor het relatief geïsoleerd kan draaien van de andere technieken. Het enige wat dit extra zou kosten is een rendertarget om de sky in te renderen. Misschien kan die gedeeld worden met andere technieken.

Rain effects zijn onmogelijk in het proces van deferred rendering te verwerken, vanwege de transparency die bij deze techniek hoort. Zoals eerder besproken wordt dit opgelost met een extra pass na het deferred renderen. Wij gaan uit van rain using geometry shading, een techniek die veel efficiënter werkt dan andere regentechnieken. Daarnaast is de techniek ook gevoelig voor lichtbronnen.

Analytic single scattering kan gebruikt worden om fog goed weer te geven. Tegelijkertijd zorgt deze techniek voor een overtuigende weergave van lichten in mistige situaties, een eigenschap die ontbreekt bij andere besproken technieken. Het is niet heel duidelijk wat voor conflicten deze fog techniek kan veroorzaken, dat zal moeten blijken uit de praktijk.

Wat betreft global illumination is er besloten dat, zeker in een procedurele en dynamische wereld, er enkel gekozen kan worden voor de SSAO techniek. Deze techniek maakt gebruik van de depth buffer in een post processing effect. Aangezien de depth buffer al beschikbaar is vanwege het gebruik van een deferred renderer is dit een efficiente combinatie.

Voor het terrein gaat onze voorkeur uit naar het gebruik van blend maps met eventuele enkele procedurele invloeden. Het is niet onze verwachting dat deze keuze conflicten zal gaan veroorzaken met de overige gekozen technieken.

Tot slot valt op te merken dat de grootste problemen qua implementatie zich in de technieken bevinden die gebruik maken van transparantie, verder lijken de technieken goed aan te sluiten op elkaar en op deferred rendering.

5 Conclusie

Duidelijk is geworden hoe een grote set rendering technieken toegepast kunnen worden. Per categorie zijn verschillende technieken besproken en beoordeeld op basis van meerdere criteria. Mede op basis van deze criteria is er per categorie een inschatting gemaakt over de best mogelijke aanpak.

Omdat het mogelijk is dat bepaalde technieken elkaar beïnvloeden of uitsluiten hebben we in de discussie de interactie tussen de gekozen technieken besproken. Hier is naar voren gekomen dat de meeste gekozen technieken goed met elkaar te combineren zijn. De voornaamste beperking wordt opgelegd door de keuze voor deferred rendering, hierdoor is het niet eenvoudig om transparente objecten te renderen. Dit zorgt echter niet voor onoverkoombare problemen.

Dankzij dit orientatieverslag kan in de komende implementatiefase snel vordering worden gemaakt. Elk groepslid is daarnaast ook in staat om met behulp van dit verslag de door ons besproken theorie en literatuur op efficiënte wijze op te frissen.

Op basis van dit onderzoek als geheel hebben we een keuze kunnen maken voor de juiste combinatie van technieken. Een samenvatting van de selectie is te vinden in ons RAD.

6 Glossary

Deze glossary biedt achtergrondinformatie voor gebruikte termen die nadere toelichting vereisen.

6.1 Render targets

Een render target geeft aan waar de output van de GPU aan gericht is. De default render target is dan ook de backbuffer, ofwel het scherm. De render target kan ook een van tevoren gereserveerd stuk videogeheugen zijn, in de vorm van een 2d texture, waarvan de informatie die per pixel opgeslagen is ook aangegeven kan worden. Door hier slim gebruik van te maken kan er in de texture meer informatie opgeslagen worden, die vervolgens in een latere fase weer gebruikt kan worden. Er zijn dan dus meer operaties mogelijk voordat uiteindelijk de backbuffer gevuld wordt met het uiteindelijke beeld.

6.2 World-space

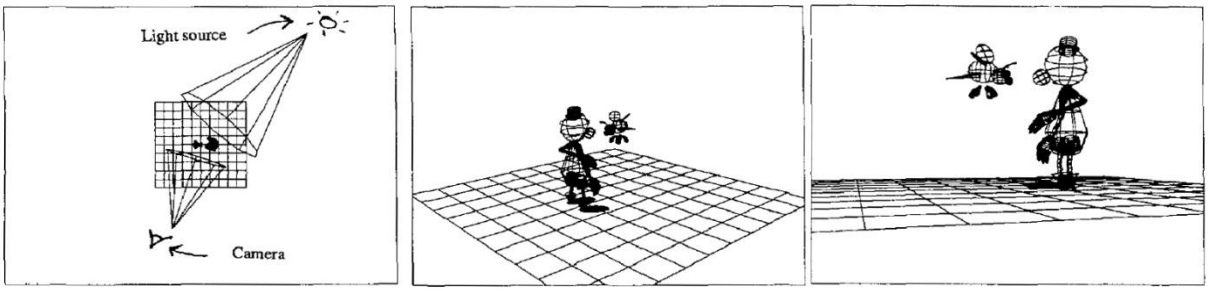
World space geeft aan dat een punt in het coordinatenstelsel van de scene staat. De objecten in de scene zijn opgeslagen en worden bewogen in world space.

6.3 Screen-space

Screen space geeft aan dat een punt in het coordinatenstelsel van het scherm staat. Met de view en projection matrices van de camera en de world matrix van elk object wordt er van world space naar screen space getransformeerd. Zodra een punt in screen space staat, kan het gezien worden als een pixel van het scherm.

6.4 Shadow Map

Een shadow map wordt gebruikt door veel shadowing technieken. Het werkt door de scene objecten te renderen vanuit het oogpunt van de lichtbron, en dit op te slaan in een render target. Dit is dan de shadow map. Als de scene dan gerenderd wordt vanuit de camera kan de shadow map gebruikt worden om per screen space pixel te zien of licht van de lichtbron de pixel kan bereiken. Dit vereist een transformatie van de screen space pixel naar de shadow map. De standaard shadow map is een hard shadow (geen smoothing) en is beperkt in kwaliteit door de resolutie van de shadow map. Zie Figuur 30.



Figuur 30. (1) View van bovenaf de scene. (2) View vanuit de lichtbron. (3) View vanuit de camera (Reeves, Salesin, & Cook, 1987).

7 Geciteerde werken

Akenine-Möller, T., Haines, E., & Hoffman, N. (2008). *Real-Time Rendering* (3rd ed.). Wellesley, Massachusetts, USA: A K Peters.

Bunnell, M. (2004). *GPU Gems.* Addison-Wesley Prfessional.

Dimitrov, R. (2007). *Cascaded Shadow Maps.* NVIDIA Corporation.

Donnelly, W., & Lauritzen, A. (2006). Variance shadow maps. *Proceedings of the 2006 symposium on Interactive 3D graphics and games* (pp. 161-165). Redwood City, California: ACM.

Engel, W. (2009). *ShaderX7: Advanced Rendering Techniques.* Boston, USA: Course Technology.

Fernando, R. (2005). Percentage-closer soft shadows. *ACM SIGGRAPH 2005 Sketches.* Los Angeles, California: ACM.

Hasenfratz, J.-M., Lapierre, M., N., H., & Sillion, F. (2003). A Survey of Real-time Soft Shadows Algorithms. *EUROGRAPHICS 2003.* France: The Eurographics Association.

Policarpo, F., & Fonseca, F. (2005). *Deferred Shading Tutorial.* Rio de Janeiro: ICAD/Igames/VisionLab.

Preetham, A. J. (1999). A practical analytic model for daylight. *ACM Press/Addison-Wesley Publishing Co. New York, NY, USA ©1999*.

Reeves, W. T., Salesin, D. H., & Cook, R. L. (1987). Rendering Antialiased Shadows with Depth Maps.

Richards, G. (2005, 6 14). *Half Life 2: Lost Coast HDR overview*. Opgeroepen op 3 25, 2011, van BitGamer: http://www.bit-tech.net/gaming/pc/2005/06/14/hl2_hdr_overview/3

Stamminger, M., & Drettakis, G. (2002). *Perspective Shadow Maps.* France: Association for Computing Machinery, Inc.

Stone, A. (2010, Januari 25). *Cascaded Perspective Variance Shadow Mapping.* Opgeroepen op Maart 25, 2011, van Game Angst: http://gameangst.com/?p=339

Sun, R. R. (sd). A practical analytic single scattering model for real time rendering. *ACM SIGGRAPH 2005 Papers .*

Tariq, S. (2007). Rain SDK White Paper., (pp. 1 - 8).

Tatarchuk, N. (2006). Artist-Directable Real-Time Rain Rendering in City Environments. *SIGGRAPH '06* (pp. 23 - 63). Boston, Massachusetts: ACM.

Umenhoffer, T., Szirmay-Kalos, L., & Szijártó, G. (2006). Spherical Billboards and their Application to Rendering Explosions. GI '06 Proceedings of Graphics Interface 2006 .

Appendix D: Implementation Plan (Dutch)

TU DELFT – COMPUTER GRAPHICS – B.SC. PROJECT

Realistic Rendering Of Virtual Worlds

Plan van Aanpak

Zhi Kang Shao, Mattijs Driel, Quintijn Hendrickx, Korijn van Golen

13-4-2011

Het Plan van Aanpak van het B.Sc. project “Realistic Rendering of Virtual Worlds”. Het beschrijft hoe het doel van een betere grafische weergave in SketchaWorld behaald zal worden, door middel van uitwerkingen van de projectopdracht, aanpak en tijdsplanning, projectinrichting en kwaliteitsborging.

Inhoudsopgave

1 Samenvatting

Dit document beschrijft hoe het BSc. project “Realistic Rendering of Virtual Worlds” voor de TU Delft aangepakt zal worden door de projectgroep. De afspraken tussen de projectgroep en de opdrachtgever is hierin vastgelegd. Er is duidelijk gemaakt hoe de requirements van het project is ingedeeld voor de beschikbare tijd.

Het implementatiewerk zal zich op de TU plaatsvinden en er zullen regelmatige status updates gegeven worden van het werk aan de opdrachtgever.

2 Introductie

Het onderzoek “Realistic rendering of virtual worlds” is ingesteld teneinde de waargenomen werkelijkheid van SketchaWorld te verbeteren. Het doel van een realistischere grafische weergave zal behaald worden door verschillende shading technieken toe te passen. Dit plan van aanpak zal beschrijven hoe de gestelde doelen bereikt moeten worden en wat voor omgeving daarbij relevant is.

2.1 Aanleiding

Naar aanleiding van communicatie tussen de onderzoeksgroep Computer Graphics & CAD/CAM op de TU Delft en de huidige projectleden is het bachelor project opgestart. Het idee van de opdrachtformulering is ontstaan uit de wens om de visualisatie van SketchaWorld realistischer te maken.

2.2 Accordering en bijstelling

Het plan van aanpak wordt zowel door de opdrachtgever Ruben Smelik als door de projectgroep ondertekend voor akkoord. Eventuele wijzigingen in het plan van aanpak worden in gezamelijk overleg besproken en doorgevoerd in een nieuwe versie van het document.

2.3 Toelichting op de opbouw van het plan

Het plan van aanpak is opgesplitst in vier onderdelen. Eerst beginnen we met een beschrijving van de projectopdracht, hierin wordt de intentie van de opdracht besproken. Vervolgens zullen we een grove tijdsplanning opstellen waarin per week wordt aangeven wat er gedaan moet worden. In het hoofdstuk projectinrichting wordt de werkvorm beschreven. Tot slot wordt er nog kort samengevat hoe de kwaliteit van het werk zal worden beoordeeld.

3 Projectopdracht

3.1 Projectomgeving

Het project vindt plaats binnen de al bestaande omgeving van SketchaWorld. Om de compatibiliteit van de geschreven software te waarborgen zal er ontwikkeld worden met behulp van dezelfde tools zoals die nu al in gebruik zijn. Dit houdt in dat er gebruik zal worden gemaakt van de programmeertaal C++ in combinatie met Visual Studio 2008. Daarnaast zal er uitvoerig gebruik worden gemaakt van de OpenSceneGraph library versie 2.9.11 in combinatie met de OpenGL shader taal GLSL. Als aan het begin van de implementatie fase een stable release van versie 2.8.4 beschikbaar is dan kan eventueel in overleg met de projectbegeleider gekozen worden voor deze versie.

3.2 Doelstelling project

De doelstelling van het project is om de realtime visualisering van de werelden in SketchaWorld realistischer te maken. Hierbij zullen we gebruik maken van de laatste technieken op het gebied van realtime graphics shaders. Belangrijke aspecten binnen het project zijn onder andere normal- en parallax mapping, real-time soft shadows en een realistisch skymodel. Er zijn echter nog veel andere categoriën waarin de bestaande visualisering te verbeteren is, deze worden uitvoerig behandeld in het oriëntatieverslag.

3.3 Opdrachtformulering

De bachelor opdracht is geformuleerd in het document 'B.Sc. project: Realistic Rendering of Virtual Worlds' zoals opgesteld door Rafael Bidarra en Ruben Smelik.

3.4 Op te leveren producten en diensten

Het opgeleverde product bestaat uit een toevoeging aan de huidige codebase van SketchaWorld. Binnen de al bestaande code zullen wij functionaliteit toevoegen voor het realistischer renderen van de wereld in SketchaWorld. Er is bewust voor gekozen deze functionaliteit rechtstreeks in het bestaande project te verwerken, om te voorkomen dat er aan het einde van het project problemen optreden bij het integreren.

3.5 Eisen en beperkingen

Een belangrijke eis aan het te ontwikkelen systeem is dat het in staat moet zijn om te draaien op hardware vanaf de NVidia GeForce 8 serie of hoger. De performance moet op deze kaarten acceptabel zijn voor een real-time systeem.

Met de projectbegeleiders is afgesproken dat er door ons geen content zal worden geproduceerd. Hieronder verstaan wij de benodigde 3d-modellen en textures die in de wereld gebruikt worden, dus ook additionele 3d-modellen en textures die onze shaders vereisen. Voor test doeleinden zullen wij wel zelf placeholders produceren die later vervangen zullen moeten worden door een artist.

3.6 Cruciale succesfactoren

Voor een succesvolle afronding van het project is het belangrijk dat we van te voren kennis hebben gemaakt met de ontwikkelomgeving. Daarnaast moeten alle groepsleden goed op de hoogte zijn van de technieken zoals ze beschreven zijn in het oriëntatieverslag.

4 Aanpak en tijdsplanning

Een wekelijkse tijdsplanning is te vinden in bijlage 1 waarin alle belangrijke onderdelen uit het RAD over de beschikbare tijd zijn verdeeld. Er is vanuit gegaan dat er parallel in groepen van twee personen gewerkt gaat worden. Aan het begin van de week wordt telkens een taakverdeling opgesteld waarin aan elke groep een taak uit de planning in 7.2 wordt toebedeeld. Om onvoorziene problemen op te kunnen vangen hebben we ook een week gereserveerd voor eventuele uitloop.

Daarnaast is er tijd ingepland voor de afronding van het project, dit omvat zowel het eindverslag als de voordracht voor de bachelor commisie.

5 Projectinrichting

5.1 Organisatie

De projectleden zijn verantwoordelijk voor de uiteindelijke implementatie van de technieken beschreven in het oriëntatieverslag. De integratie in SketchaWorld is een gedeelde verantwoordelijkheid van de projectleden en Ruben Smelik.

5.2 Personeel

De projectleden zullen in totaal één kwartaal full time aan het project werken. In het derde kwartaal zal op twee dagdelen gewerkt worden, het vierde kwartaal zal op 8 dagdelen gewerkt worden.

Van de begeleiders wordt verwacht dat ze op zijn minst wekelijks tijd hebben om de voortgang te evalueren.

5.3 Administratieve procedures

Elke 14 dagen zal er een progress report worden ingevuld. Deze zullen ook naar de begeleiders worden gestuurd en zijn onder andere als controle bedoeld om de voortgang van het project te monitoren. Eventuele afwijkingen van de planning kunnen zo snel worden opgemerkt.

Dagelijks zal er aan het begin van de dag worden vergaderd. De notulen hiervan zullen op de SVN-repository worden geplaatst. Hierbij zal Korijn voorzitter zijn en wordt er door Zhi-Kang genotuleerd.

5.4 Financing

Financiering is niet van toepassing. Voor zover wij kunnen inschatten zijn er voor een succesvolle afronding van het project geen financiele middelen nodig.

5.5 Rapportering

Zoals beschreven in 5.3 zal de voortgang gerapporteerd worden door middel van progress reports en notulen uit de vergaderingen.

5.6 Resources

Met de begeleiders is er afgesproken dat er op de EWI faculteit een werkplek beschikbaar wordt gesteld. Hierbij is het belangrijk dat er computers aanwezig zijn met een goede grafische kaart en voldoende rechten om te kunnen werken in de opgelegde ontwikkelomgeving.

6 Kwaliteitsborging

Door op zijn minst wekelijks te spreken met de begeleider worden alle betrokkenen actief op de hoogte gehouden. De kwaliteit van het werk wordt gepresenteerd aan de hand van korte demonstraties. Eventuele problemen in verband met de kwaliteitsborging kunnen zo snel worden opgelost. Ook wordt wekelijks besproken wat er voor de volgende week gepland is, zo nodig met diagrammen om de planning duidelijk te maken.

7 Bijlagen

7.1 Opdrachtomschrijving

‘B.Sc. project: Realistic Rendering of Virtual Worlds’, December 10, 2010

7.2 Weekschema planning

Projectweek	Jaarweek	Geplande taken	Tentamens
1	16	Installatie werkomgeving: Visual Studio en OpenSceneGraph Begin implementatie deferred renderer Begin dynamic soft shadows	
2	17	Implementatie deferred renderer en mapping techniques Dynamic soft shadows	
3	18	Integratie deferred renderer en mapping techniques Dynamic soft shadows	
4	19	Integratie dynamic soft shadows Point lights and spot lights in deferred renderer	
5	20	Integratie Point lights and spot lights in deferred renderer HDR	
6	21	Integratie HDR Begin skymodel en clouds	
7	22	Skymodel en clouds Rain effects	
8	23	Integratie skymodel en clouds Integratie rain effects	
9	24	SSAO Optimalisaties	
10	25	Integratie SSAO Afronding verslag en voorbereiden presentatie	X
11	26	Gereserveerd voor uitloop Night vision	X

7.3 Aanwezigheid groepsleden

Maandag	Dinsdag	Woensdag	Donderdag	Vrijdag
	ZKMQ	ZKMQ	ZKQ	ZMQ
ZKM	ZKMQ	KMQ	ZKMQ	

Z = Zhi Kang, K = Korijn, M = Mattijs, Q = Quintijn

Appendix E: Requirement Analysis Document (Dutch)

TU DELFT – COMPUTER GRAPHICS – B.SC. PROJECT

Realistic Rendering Of Virtual Worlds

Requirements Analysis Document

Zhi Kang Shao, Mattijs Driel, Quintijn Hendrickx, Korijn van Golen

1-4-2011

Het RAD van het B.Sc. project “Realistic Rendering of Virtual Worlds”. Het beschrijft de eisen waaraan het eindproduct en de procedures moeten voldoen.

Inhoudsopgave

1 Scope

- The project results should include a working implementation of the selected rendering techniques (see chapter 7);
- The main focus of this project is on rendering, therefore (procedural) generation of e.g. input maps or geometry is not this project's primary responsibility.

2 Integration

- A separate development and testing environment and repository is allowed;
- However, regular milestones and integration steps have to be planned to demonstrate that the developed components work in SketchaWorld;
- The separately developed techniques should be compatible with each other.

3 Reusability

- The aim of the project should be to develop reusable rendering components;
- The exception to this is the need for direct integration of shader fragments, render states and variables, etc.;
- A programming guide should describe how to integrate the components with an existing OSG-based setup.

4 Implementation

- The implementation of the rendering components will be written in C++, on top of the rendering library OpenSceneGraph. Shaders will be written in GLSL.

5 Open source

- The project should aim at releasing (part of) the components as open source, for instance as contributions to the OpenSceneGraph community or as separately maintained nodekits.

6 Performance considerations and constraints

- The rendering framework should run at an acceptable frame-rate on a DX10 GPU (e.g. a Geforce 8800 GT)
- Performance tests should also be performed on larger scale landscapes (e.g. 16x16 km)

7 Selected Rendering Techniques

7.1 Must Have

1. Deferred Shading
2. Cascaded Shadow Maps
3. Percentage-Closer Soft Shadows
4. Normal Mapping
5. Parallax Mapping
6. High Dynamic Range & Tone Mapping
7. Raindrops using Geometry Shaders

8. Cloud Density Layers

7.2 Should Have

1. Screen Space Ambient Occlusion

7.3 Could Have

1. Procedural Terrain Shading
2. Analytic Single Scattering
3. Atmospheric Scattering