FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Scenery3d - Walkable 3D Models in Stellarium

Real-time rendering improvements

BACHELOR'S THESIS

by

BORZA, Andrei Marcel
Registration Number 0625499

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

Contents

1 Introduction **1**
- 1.1 General Information . . . 1
- 1.2 Stellarium . . . 1
- 1.3 Scenery3d . . . 2

2 Scenery3d **3**
- 2.1 Plugin architecture . . . 3
- 2.2 Scene loading . . . 3
- 2.3 Scene rendering . . . 6
- 2.4 Shadow mapping . . . 6
- 2.5 Thesis goals . . . 7

3 Improvements **9**
- 3.1 Scenery3d . . . 9
- 3.2 Shadows . . . 12
  - 3.2.1 Definition . . . 12
  - 3.2.2 Shadow mapping . . . 13
    - 3.2.2.1 Problems with shadow mapping . . . 15
    - 3.2.2.2 Error analysis . . . 16
  - 3.2.3 Fighting perspective aliasing . . . 17
    - 3.2.3.1 Focusing the shadow map . . . 18
    - 3.2.3.2 Cascaded Shadow Maps . . . 20
  - 3.2.4 Filtering . . . 21
    - 3.2.4.1 Poisson disk . . . 22
- 3.3 Normal mapping . . . 22
  - 3.3.1 Lighting . . . 23

4 Conclusion **27**
- 4.1 Future work . . . 28

Bibliography **29**

CHAPTER 1

Introduction

1.1 General Information

For this thesis we focused on improving Scenery3d, a plugin for the popular open source software Stellarium. Scenery3d was developed by students of the Vienna University of Technology to enable walking through models of architectural structures, which were built with astronomical orientation in mind.

First, a short introduction of Stellarium and in particular Scenery3d is given. Afterwards, we discuss the current state of Scenery3d and the thesis goals set during its inception. In part three, the technology and ideas behind the planned improvements are presented. Finally, we draw conclusions and suggest future work that could improve the plugin further.

1.2 Stellarium

Stellarium is an open source software available for free [1]. The project was conceived by Fabien Chéreau in 2001 with the goal to provide a realistic and mathematically correct software which renders skies in real time.

Over the years Stellarium grew into a big and healthy open source project with collaborators located all over the world. Many features were added and Stellarium turned into an easy to use software which is used in planetariums, universities and by professional and hobby astronomers alike.

Popular features include the possibility to render over 210 million stars, a realistic Milky Way, realistic atmosphere, sunrise and sunset but also to project onto planetarium domes due to the fisheye projection Stellarium offers. People with telescopes can connect their devices to Stellar-

ium and control the telescope from within the software.

The beauty of Stellarium is that due to its mathematically correct calculations, it is possible to traverse back in time and calculate and display skies at any given time. This is especially useful for simulating eclipses and other events back in, but even ahead of, time.

Last but not least, Stellarium offers a powerful plugin engine, making it relatively easy to develop and publish your own plugin. Since its inceptions, many additions in form of plugins were added to Stellarium, for example adding artificial satellites, ocular simulation, custom landscapes and, through our plugin, even load in custom architectural structures.

1.3 Scenery3d

The plugin Scenery3d was conceived by Georg Zotti for the project ASTROSIM [2] and was mostly developed by Simon Parzer and Peter Neubauer during the winter semester of 2010 at the Vienna University of Technology. Additional contribution in form of improvements, bug fixes, and testing was added by Georg Zotti. The work, as well as this thesis, was supervised by Professor Michael Wimmer, from the Institute of Computer Graphics and Algorithms at the Vienna University of Technology.

The idea behind Scenery3d was to combine the simulation abilities of Stellarium with a way of interacting more actively with architectural structures and geometry in general.

This however posed some challenges as Stellarium was not intended to be used in such a way. A simple way of loading models or entire scenes had to be implemented. Another big challenge was the fisheye distortion that is a core part of Stellarium.

One key feature, and main focus of this work as well, has been the rendering of realistic and dynamic shadows onto the loaded scene making full use of Stellarium's light sources such as the sun, the moon or venus. After all, this makes the use of our plugin apparent and the plugin itself shines.

CHAPTER 2

Scenery3d

2.1 Plugin architecture

While Stellarium supports dynamically loaded stand-alone plugins, Scenery3d was created as a static plugin. This means that Scenery3d is part of Stellarium's binary files, however it also means, that it requires a custom built version of Stellarium. It is not possible to load our plugin from any other Stellarium binary or branch without explicitly implementing our code.

Scenery3d's main class is `Scenery3dMgr` (cf. Figure 2.1) which is a derived class of `StelModule`. As such, the module is exported using the `StelPluginInterface` which enables Stellarium to load the plugin.

`Scenery3dMgr` provides a configuration dialog through an instance of `Scenery3dDialog`, and scene loading and rendering through an instance of `Scenery3d`.

The configuration dialog (cf. Figure 2.2) can be accessed through a Scenery3d button in Stellarium's bottom menu bar. It lists all loadable scenes, as well as an information text for each scene, and a checkbox for enabling experimental shadow mapping. Throughout this work, the functionality of the plugin was increased and thus the dialog changed a bit. See chapter 3 for more detail.

As mentioned above, the class `Scenery3d` is responsible for our plugin's rendering, user interaction with the scene and scene loading. For latter, a simple model loader based on *Wavefront Technologies' OBJ file format* was developed.

2.2 Scene loading

Scenery3d can load scenes and models that are exported as *Wavefront OBJ* [3] from any modeling software that supports this file format. This format was chosen due to its simplicity, human

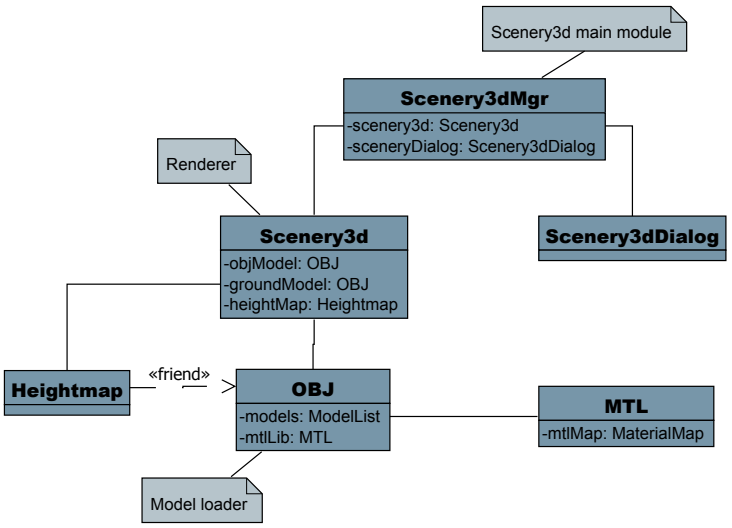

Figure 2.1: *Scenery3d plugin structure*

Figure 2.2: *Scenery3d configuration dialog*

readability and because the .OBJ ascii format is cost free.

Every scene consists of a configuration file, .OBJ files that represent the geometry and possibly texture files if a textured scene is at hand. Detailed information about scene configuration can be found in section *4.2 Configuring OBJ for Scenery3d* [4] of the existing documentation.

The model loader only supports a subset of the entire *Wavefront OBJ* file format. Following lines are valid with Scenery3d's OBJ loader.

- Vertex data
  - `v` geometric vertices
    Each vertex has three components `x, y` and `z` where each component is a floating point number representing the vertex position in that coordinate.
  - `vt` texture vertices
    Each texture vertex has two components `u` and `v` where each component is a floating point number. `u` represents the horizontal direction, `v` represents the vertical direction of the texture.
  - `vn` vertex normals
    Each vertex normal has three components `i, j` and `k` where each component is a floating point number representing the vertex normal.
- Elements
  - `f` face
    Each face consists of three triplets of values. Values are separated by slashes (*/*) and each value references the previous mentioned vertex data in the same order.
- Grouping
  - `g` group name
    Models in a scene can be grouped by using this statement followed by a group name. This helps to organize the scene into several sub models. Every statement following this line belongs to the defined group until the next group is set or the file ends.
- Display/render attributes
  - `usemtl` material name
    This line must be placed directly after a group defining line and specifies the material to be used for this group.
  - `mltllib` material library
    A material file (*.mtl*) defines the previously mentioned materials. The path to the file has to be specified after the key word.

It is noteworthy to mention, that the loaded scene is rotated by a rotation matrix *rot_z* which can be specified in *scenery3d.ini*. This adjusts the scene from geographical north to grid north.

Finally the data is converted to Stellarium's vector format and can be accessed via *OBJ::getStelArrays*. This data is ready to be used with Stellarium's *StelPainter::setArrays* and *StelPainter::drawFromArray* to render the data.

2.3 Scene rendering

While Stellarium offers many more projections, Scenery3d focuses on supporting perspective and fish-eye projections. This is due to the difficulty of rendering non-linear projections in OpenGL.

If the hardware supports the *GL_EXT_framebuffer_object* OpenGL extension, Scenery3d will always choose the fish-eye projection for rendering, otherwise the perspective projection will be chosen.

To achieve rendering this type of projection, the plugin renders the scene with a perspective projection to each face of a cubemap. Afterwards Stellarium's *StelPainter::drawSphericalTriangles* function is used to distort the cubemap.

Stellarium itself offers the use of shader programs, however Scenery3d makes exclusively use of OpenGL's fixed function pipeline.

2.4 Shadow mapping

Being able to traverse forward and backward through time at desired pace in Stellarium, enables the user of Scenery3d to search for possible astronomical alignments of any given architectural model. It is of particular interest to observe sundials and other shadow casters and analyze their influence on the structure.

For this, the previous authors of Scenery3d implemented *shadow mapping* [5] as an experimental feature. The implementation is based on [6] and uses three render passes to render the scene with shadows.

First a *frame buffer object* is created and bound for the first pass. The view is switched to the light source's point of view and the scene's depth values are rendered to the offscreen texture.

Afterwards the view is switched back to the camera's point of view and the scene is rendered using dim diffuse and zero specular lighting.

Finally the scene is rendered one more time using full light as specified in the *.MTL* of the scene. During this pass the actual depth comparison takes place, so that only points lying outside of shadows are lit.

However as mentioned above, this feature was added only experimentally. Due to the nature of traditional *shadow mapping*, aliasing effects occur when scenes are too large and the shadow map has a too low resolution to cover the entire scene. Additionally, the authors ran into problems with multitexturing and Stellarium, thus when the effect is activated, only the part of the scene that is in shadow is fully textured while the rest of the scene is textureless (cf. Figure 2.3).

These and some other aspects were improved by this work and will be discussed in more detail in the following chapter.

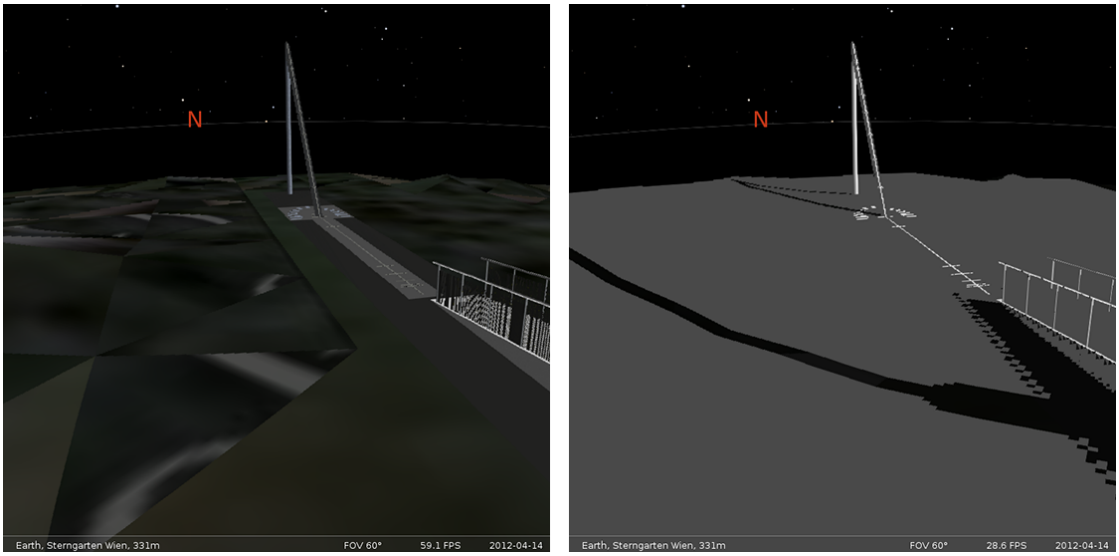

Figure 2.3: *Left: A fully textured scene loaded in Scenery3d. Right: The same scene with experimental shadow mapping on. Notice the lack of textures due to the problem with multitextures and Stellarium.*

2.5 Thesis goals

The primary goal for this thesis was to improve the rendering effects of the Scenery3d plugin. This includes reworking the already implemented shadow mapping algorithm, overhauling the lighting system and adding texture effects. In addition to that, refining the model loader and improving rendering speed were set as side goals but will not be discussed here.

Following are the details regarding the most important goals.

- **Shadow mapping**
  Solving the multitexture problem was the first thing to consider for this effect overhaul. Moving away from the fixed function pipeline by using shader programs, and thus increasing rendering speed, was crucial.

  Finally the poor shadow resolution had to be improved by implementing more clever shadow mapping algorithms.

- **Normal mapping**
  Simple texture mapping was implemented by the previous authors, however to increase realism, it was decided that normal mapping should be added.

- **Model loading**
  In order to support the previously mentioned normal mapping effect, the model loader had to be reworked. This includes supporting a broader subset of the .OBJ standard [3] than the original model loader did.

  In addition, improving loading time, decreasing memory usage and fixing bugs was also desired.

CHAPTER 3

Improvements

In this chapter the defined core goals for this thesis will be discussed in detail. For this, we present related work and reasons for choosing following techniques.

First, details on minor improvements to Scenery3d and Stellarium will be given. Then we present traditional shadow mapping, followed by *z-Partitioning* (also known as *Cascaded Shadow Maps*). Afterwards, we briefly review filtering our shadows to make them appear less hard and finally, we discuss normal mapping to boost the realism of the loaded scene at minor costs.

3.1 Scenery3d

In order to boost performance we decided to move away from `glDrawArrays`, which traverses the vertex array in a straight forward fashion. This means that shared vertices need to be repeated per face. For example, for a cube with 3 vertices per face, the vertex array consists of 36 vertices.

Instead, we are using `glDrawElements`, which is able to hop through the vertex array based on additional info. For this, an index array has to be specified which instructs OpenGL on which vertices make up a face. Vertices do not have to be saved redundantly into the vertex array anymore. For the previous example, only 8 vertices, those that make up the cube, would need to be stored.

While Stellarium supports indexed drawing, it assumes that the byte offset to the next vertex (also known as stride) is 0. In order to allow a different stride, `StelPainter` from Stellarium's core had to be modified slightly. The method `StelPainter::drawFromArray` takes an additional optional parameter `int stride`, which by default is set to 0 to retain compatibility with previously written source code.

To support the loading of external shader files, the `StelShader` class was introduced from Stellarium's *Socisplanet* branch [7] and small changes were made to `StelShader::loadShader`.

Furthermore, small changes to Scenery3d's architecture were made (cf. Figure 3.1). In particular the model loader was completely replaced and the `MTL` class removed in favor of a simple `struct Material` inside `OBJ`.

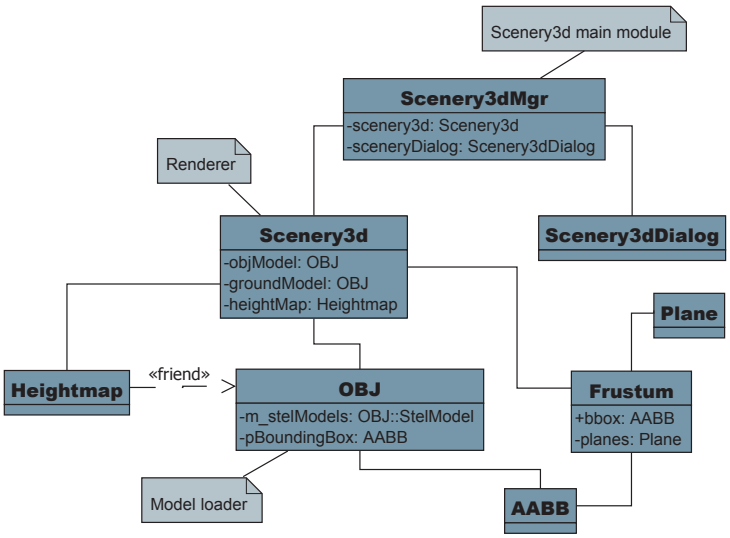

Figure 3.1: *Updated Scenery3d plugin structure*

The following new classes have been introduced to mainly aid the new shadow mapping algorithm. They could be of further use for rendering optimizations in the future.

- `AABB` represents an axis aligned bounding box
  Computing an object's boundaries is often crucial for optimizing rendering speed further. The object is placed within a tightly fitting bounding volume, which is then used as its representation for further tests. This is computationally inexpensive and leads to less complex algorithms while delivering good enough results in most cases.

- `Frustum` represents a view frustum
  In computer graphics, the view frustum is used to describe a volume that holds the entire field of view of a virtual camera. For example, an object fully inside this volume would also appear fully displayed on the screen. Objects that are outside this volume would not be displayed at all (but still be rendered).

The shape of the view frustum depends on the projection used for the virtual camera. However, it usually resembles a cut off pyramid consisting 6 planes (cf. Figure 3.2).

- `Plane` represents a polygon plane
  A plane can be created by specifying three vertices.

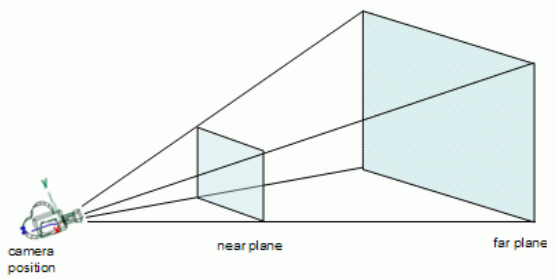

Figure 3.2: *A typical view frustum based on a perspective projection. Image courtesy of Lighthouse3d.com [8].*

As mentioned before, the previous authors of Scenery3d had problems implementing multitexturing due to incompatibilities with Stellarium's core. However, for this work, we introduced the use of shader programs, thus bypassing the fixed pipeline (at least partially) and making it possible to access several texture units (cf. Figure 3.3).
We picked the following texture units for Scenery3d.

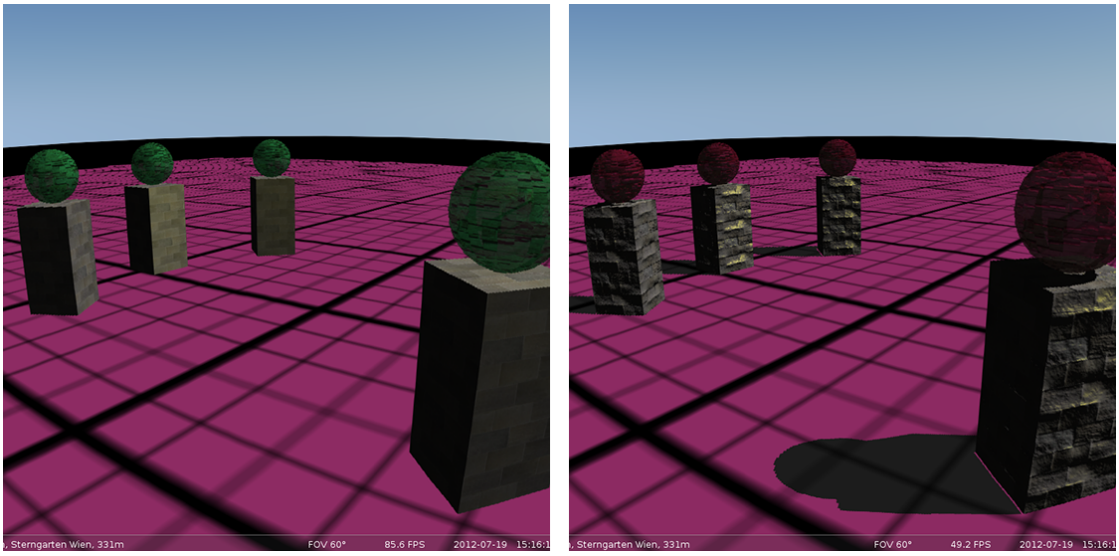

Figure 3.3: *Left: Scene rendered using fixed function pipeline with no special render-features activated. Right: Scene rendered using shader programs with normal mapping and shadow mapping activated. Unlike before, the object textures are displayed properly.*

- `GL_TEXTURE0`
  This is the default texture unit. It is used for traditional texturing of objects.
- `GL_TEXTURE2`
  We use this texture unit for the normal map.
- `GL_TEXTURE3..6`
  Depending on how many frustum splits are chosen for cascaded shadow maps, texture units 3 through 6 are used to store the depth maps. For each depth map, one texture unit is used.

In order for our plugin to function, it is crucial that these units are not being used elsewhere.

3.2 Shadows

Shadows play an integral role in our lives. Not only do they give us invaluable information on the viewable scene, but also help us to judge its out of sight surroundings, the spatial relationships between objects, the direction and possible position of the light sources, and many more things. As such, shadows also play an integral role in computer graphics.

Over the years many papers, articles and books have been published on the topic of rendering shadows. Finding a solution for real-time shadow rendering is of particular interest nowadays. Although there exists many clever techniques, an approach that works for any context has yet to be found.

However, with clever tuning and selecting of appropriate shadow rendering algorithms, it is possible to achieve an acceptable level of realism while at the same time keeping performance and interactivity stable.

Before introducing techniques that were important to us, we first want to give an overview of how shadows, in terms of computer graphics, are defined. Afterwards we introduce the most popular algorithm to render shadows in real-time known as *shadow mapping* [5] followed by a technique called *z-Partitioning* [9] or *Cascaded Shadow Maps* [10] that uses multiple shadow maps to increase the shadow resolution drastically.

Finally, we briefly mention filtering to reduce remaining aliasing and soften the produced shadows.

3.2.1 Definition

Defining *shadows* is not as straight forward as it seems. For years people have been researching this topic, with Lenoardo DaVinci's *Codex Urbinas* probably being the first work on shadows.

Hasenfratz et. al [11] defines shadows for computer graphics as:

The region of space for which at least one point of the light source is occluded.

From this definition we can immediately deduct two major points.

- Global illumination is not considered.
  This means, only light coming directly from the light source has an effect on the scene. Indirect illumination, for example from reflective surfaces, is ignored.

- Light-occluders are opaque.
  This is very often not the case, in real life as well as computer graphics.

While these constraints reduce the degree of realism quite a bit, this definition of shadows is sufficient for this work, at this stage. Everything beyond is subject to potential future improvements.

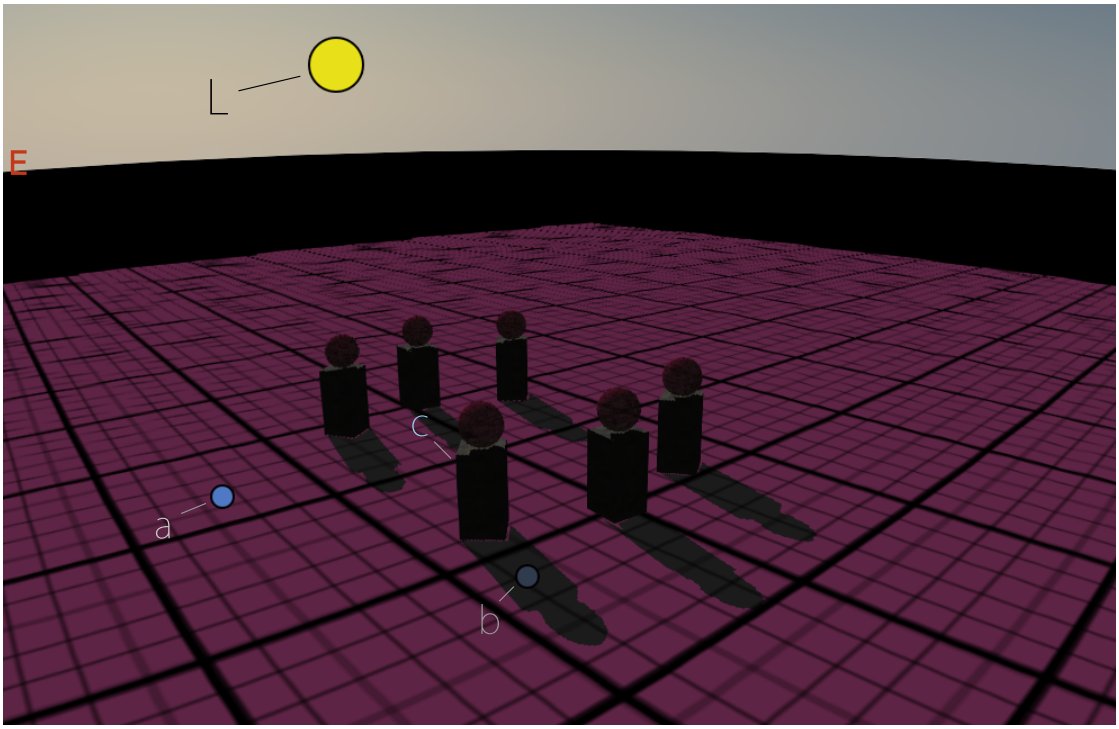

Figure 3.4: *For the light source* L *and the occluder* c*, point* b *lies in shadow, while point* a *is fully lit because no obstacle blocks the light coming from* L*.*

3.2.2 Shadow mapping

There are many clever techniques to create and improve shadows in real-time rendering. Many of them build up on an algorithm introduced by Williams in 1978, often referred to as *shadow mapping* [5]. Although other ways of creating shadows exist, today, shadow mapping is the most

popular way of doing so.

In the previous subsection we mentioned that a given point in a scene is in shadow if, and only if, the light source does not reach this point without intersecting at least one occluder. In order to find out if that is the case, geometric computations could be performed. However, a more clever and robust way exists in the form of shadow mapping.

The algorithm consists of two render passes. Following are the steps in detail:

- 1. Switch the view to the light source's point of view
- 2. Render the depth values into a depth map, also often referred to as *shadow map*, using a z-buffer.
- 3. Switch the view back to the camera's point of view
- 4. While rendering again, bring each point from the scene into light space
- 5. Compare the transformed point with its counter-part from the previously rendered depth map
- 6. If the point's depth is bigger than the stored point from the depth map, the point must be in shadow, otherwise the point is fully lit (cf. Figure 3.5).

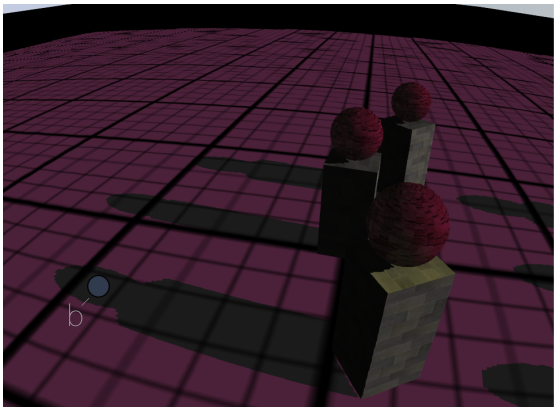

Figure 3.5: *Left: Scene rendered from the camera's point of view showing point* b *as in shadow after comparing it to point* b' *in the right picture. Right: Scene rendered as seen from the light sources' point of view, saving only the depth values.*

Due to its popularity and simplicity, shadow mapping is fully supported by modern hardware. A simple implementation can be seen in chapter 4.

3.2.2.1 Problems with shadow mapping

Shadow mapping relies on creating a depth map using the discrete z-buffer as discussed previously. As such, we are facing two problems due to the limited possible shadow map resolution and limited precision.

1. Depth-Bias
When using shadow mapping to render shadows, we often create unwanted false self-shadowing as well, which is known as z-fighting or surface-acne (cf. Figure 3.6). This happens due to the fact that the shadow map is finite.

A pixel in the depth map is based on the point that maps to the pixel's center. Thus, when looking up a point, it will often not relate to the point that was actually sampled in the depth map. Unwanted shadows show up when a point is farther away from the light source than the looked up pixel's value.

To counter this, we need to set up a bias that gives us some tolerance for the depth test against the depth map.

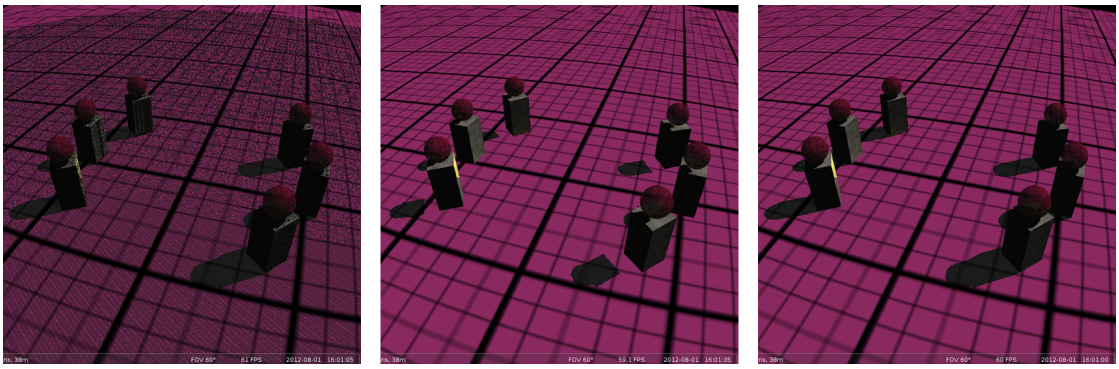

Figure 3.6: *Left: Moiré patterns emerge due to too little bias. Middle: Too high bias leads to the so called Peter-Panning of shadows (shadows are detached from their corresponding objects. Right: No self-shadowing artifacts emerge, due to properly chosen bias.*

2. Aliasing
Since the depth map is an image-based scene representation, the quality of shadows are directly related to the depth map's resolution. It is entirely possible – and often the case – that two or more different view points are mapped to the same pixel in the shadow map while following lookups might access adjacent pixels. This results into visible stair-stepping-artifacts (cf. Figure 3.7) and is known as *perspective aliasing* while *projection aliasing* occurs when light rays are almost parallel to a surface.

Although increasing the depth map's resolution would solve this problem, as so often in com-

puter science, we have limited resources and especially in real-time rendering it is important to be as efficient as possible in order to guarantee fluid interactivity. Therefore many clever techniques were developed to improve the resolution of shadows without exceeding the limited resources at hand.

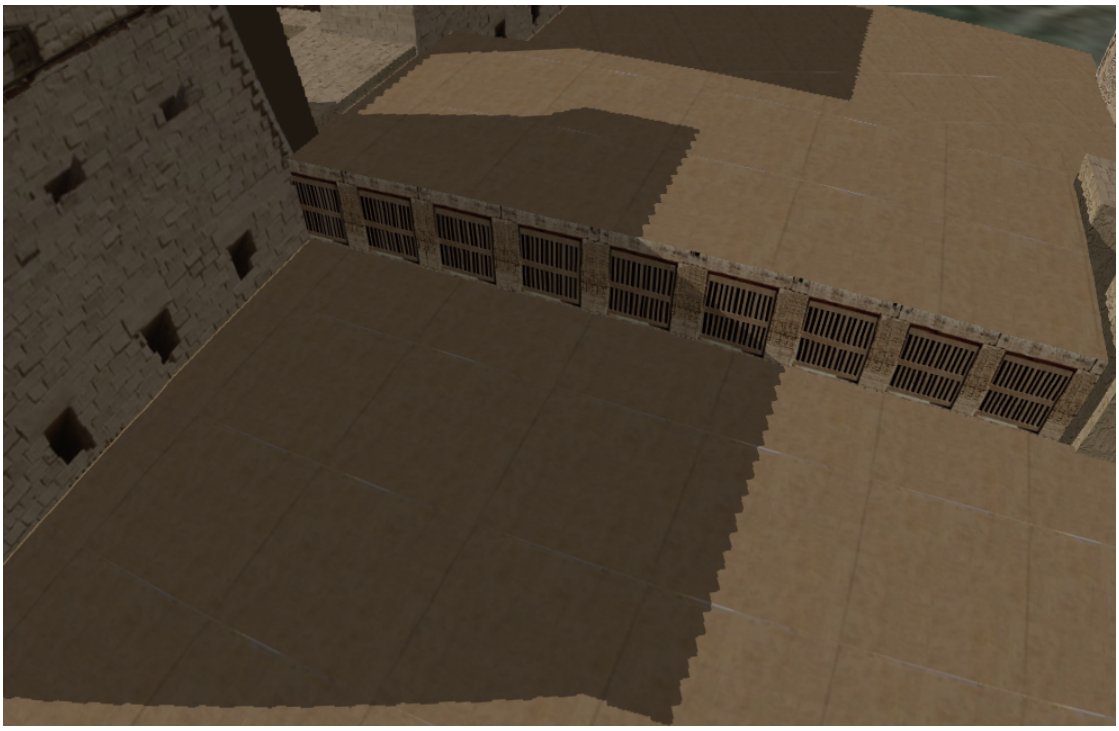

Figure 3.7: *Aliasing effect from limited depth map resolution. The stair-stepping artifacts are especially visible for receivers close to the camera.*

3.2.2.2 Error analysis

We want to briefly discuss the aliasing errors in this section. For this we look at *Light Space Perspective Shadow Maps* [12] in which a simple aliasing error analysis was introduced for directional lights.

The aliasing error of the shadow map shown in Figure 3.8 is

$$\frac{dp}{ds} = \frac{z_n}{z}\frac{dz}{ds}\frac{cos\alpha}{cos\beta}. \tag{3.1}$$

When a light ray coming through the shadow map hits the object's surface, its intersection of length dy projects onto the view plane as length dp. Aliasing occurs when dp is larger than a

pixel on the view plane.

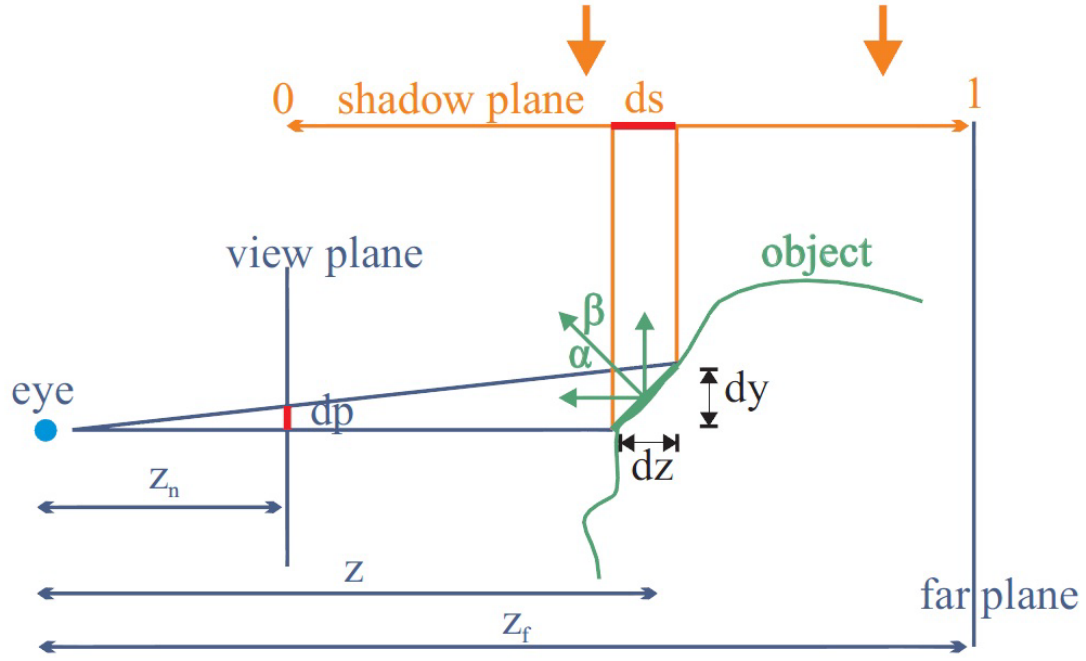

Figure 3.8: *Aliasing in shadow mapping. Image courtesy of Wimmer et al. [12]*

There are two scenarios for when that happens:

- *Projection aliasing* occurs if $cos\alpha/cos\beta$ is large.
- *Perspective aliasing* occurs if $dz/(zds)$ is large.

We are mainly interested in dealing with perspective aliasing due to the larger artifacts (cf. Figure 3.7). These errors are harder to deal with than the errors coming from projection aliasing.

For a more thorough analysis of shadow map aliasing we recommend Chapter 3 in *Real-time shadows* [13].

3.2.3 Fighting perspective aliasing

In the previous section we have discussed the traditional shadow mapping algorithm by Williams [5] and its aliasing problems. For this section, we present our choices of state of the art algorithms to tackle these problems for Scenery3d.

We start with focusing the shadow map as introduced by Brabec et al. [14] as well as Stamminger and Drettakis [15] and improved by Wimmer and Scherzer [16]. Afterwards we discuss *Cascaded Shadow Maps* [10] to achieve high resolution shadows.

3.2.3.1 Focusing the shadow map

The very first thing to do when trying to improve shadow quality is to focus the shadow map. This means, making sure that the shadow map is generated only for the visible part of the scene, i.e. everything in the view frustum.

However, focusing just on the view frustum will often lead to incorrect shadowing. Objects outside of the camera view are excluded from the shadow map even though their shadow might be cast inside the view frustum (cf. Figure 3.9).

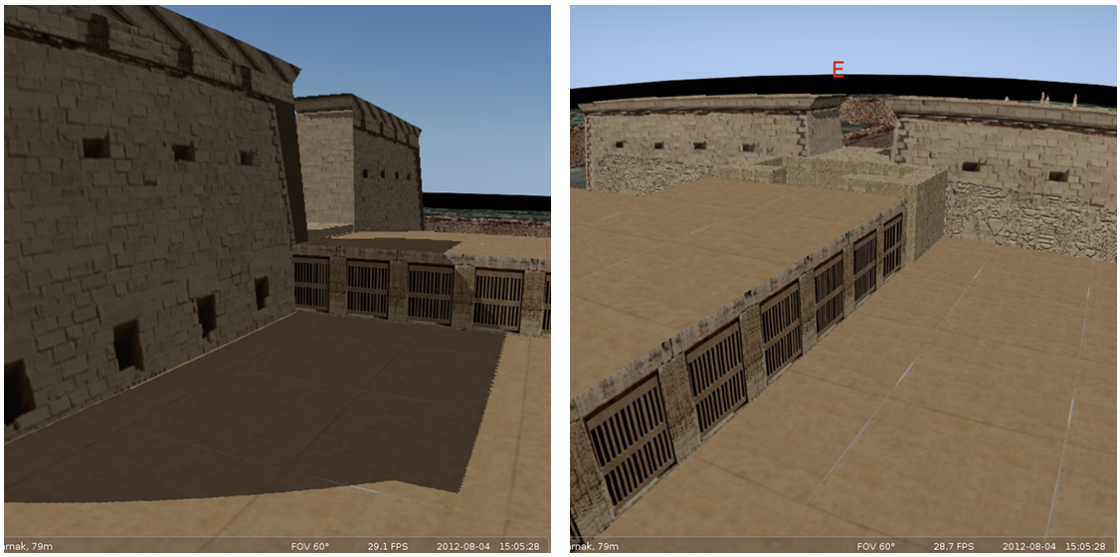

Figure 3.9: *Left: The wall on the left is casting a shadow onto the ground. Right: The wall is now behind the view frustum, the shadow on the ground is missing.*

In order to deal with this, we have to find a convex body that encompasses all the potential shadow casters as well as all the shadow receivers. Based on the suggestions in *Perspective Shadow Maps* [15], Wimmer and Scherzer [16] recommend a slightly different approach which we will present here.

For spot lights, we first compute the intersection of the camera view frustum V and the scene's bounding box S to receive the body $H = V \cap S$. We then combine H and the light source's position l to build the convex hull M. Finally, we exclude all points that are outside of the scene's bounding box and the light source's frustum by computing the body $B = M \cap S \cap L$ (cf. Figure 3.10).

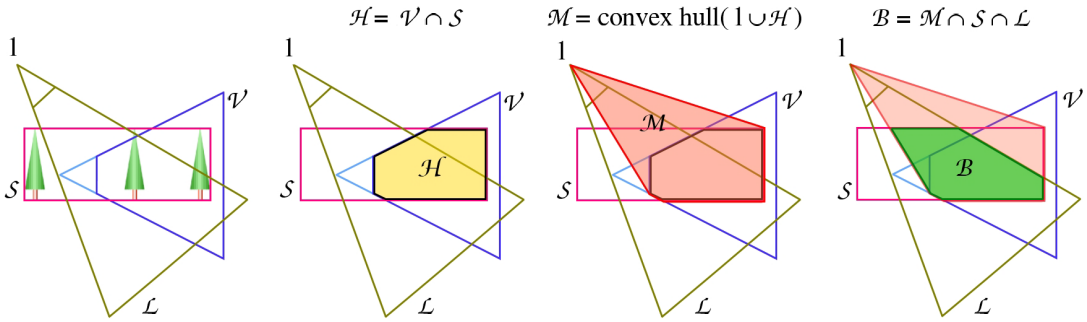

Figure 3.10: *Finding the convex body to fit the shadow map for spot lights. Image courtesy of Wimmer and Scherzer [16]*

For directional lights, we cannot compute the convex hull because $l = \infty$. In order to obtain the convex body B, we have to extrude every point of H along the light direction until the scene bounding box S is hit (cf. Figure 3.11).

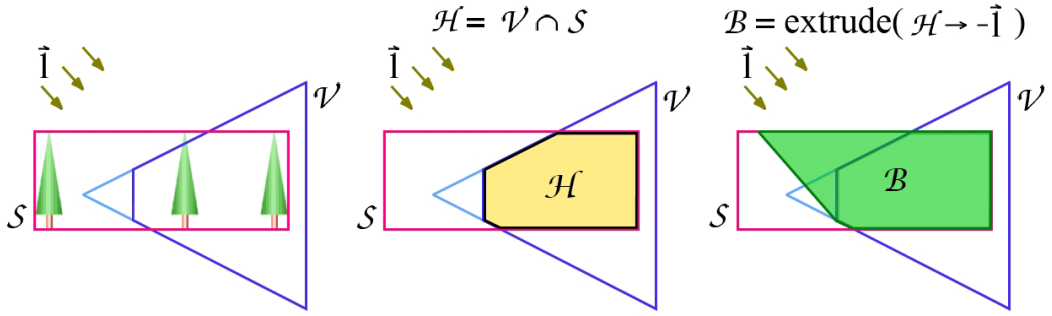

Figure 3.11: *Finding the convex body to fit the shadow map for directional lights. Image courtesy of Wimmer and Scherzer [16]*

After the body B is computed, all that is left to do, is to actually focus the shadow map onto it. For this we project its points into light space, compute the axis aligned bounding box and use the newly acquired points $(x_{min}, y_{min}, z_{min})$ and $(x_{max}, y_{max}, z_{max})$ to compute the crop matrix F.

Focusing the shadow map can be done by applying the matrix F to the light source's view-projection matrix:

$$F = \begin{pmatrix} s_x & 0 & 0 & o_x \\ 0 & s_y & 0 & o_y \\ 0 & 0 & s_z & o_z \\ 0 & 0 & 0 & 1 \end{pmatrix}, \tag{3.2}$$

with $s_x = \frac{2}{x_{max}-x_{min}}$, $s_y = \frac{2}{y_{max}-y_{min}}$, and $s_z = \frac{1}{z_{max}-z_{min}}$ being the scales, and $o_x = -\frac{s_x(x_{max}+x_{min})}{2}$, $o_y = -\frac{s_y(y_{max}+y_{min})}{2}$, and $o_z = -z_{min}s_z$ being the offsets in in x, y, and z.

As a result, we now have all potential shadow casters in the shadow map ensuring correct shadowing (cf. Figure 3.12).

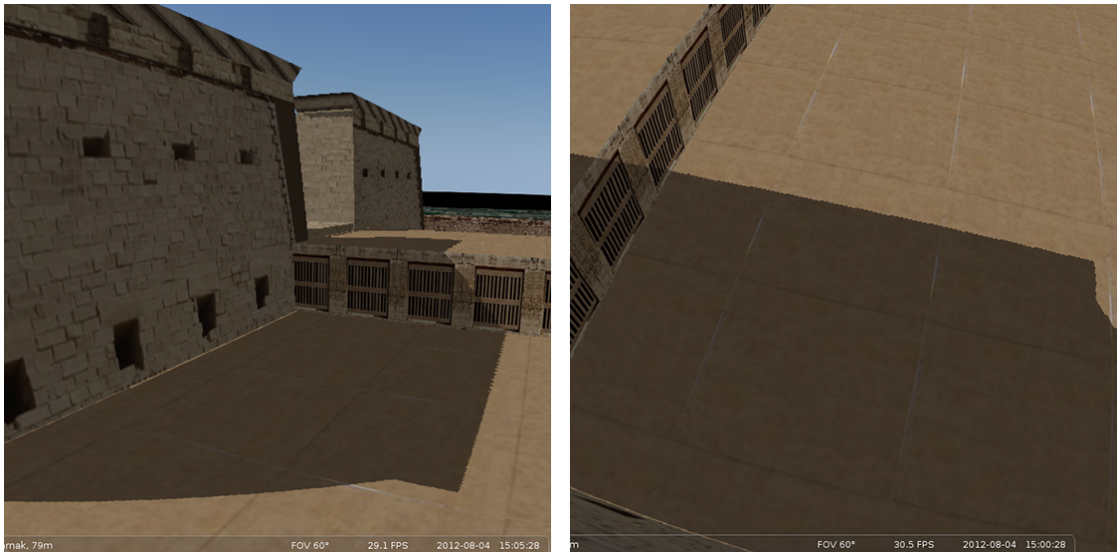

Figure 3.12: *Left: The wall on the left is casting a shadow onto the ground. Right: The wall is now behind the view frustum, the shadow on the ground is there even though the caster is behind the view frustum.*

3.2.3.2 Cascaded Shadow Maps

Focusing the shadow map is necessary, but often not enough to improve the quality of shadows. For large scenes, e.g. outdoor scenes, a single typical (i.e. as presented in the previous sections, limited resolution) shadow map is not going to be enough.

We witness aliasing effects for objects that are near to the camera while objects far away are oversampled. In other words, we want to have an area close to our observation point in high resolution whereas shadows in the distance can be less detailed.

Tadamura et al. [9] were the first to propose the use of multiple shadow maps to counter the perspective aliasing effect. They subdivide the view frustum into several sub frusta and assign shadow maps with different resolutions to each sub frusta.

Cascaded Shadow Maps [10] builds up on this idea and was chosen for this work because it fits our outdoor scenes and is more robust against light that is coming from directly behind the viewer than shadow map warping solutions.

The CSM algorithm consists of following steps:

- 1. Split the view frustum into *n* partitions
- 2. For each partition *i*, compute the convex body, as described previously, based on the current subfrustum
- 3. Focus the light source's frustum onto the split's convex body and render a shadow map
- 4. Render the scene and perform a lookup, based on the fragment's z-coordinate, to find out which shadow map to query

It is noteworthy to mention, that for this work we ran into problems querying the fragment's depth and thus decided to base the lookup in view space by using the squared viewing distance instead.

One question left to answer is related to how we split the view frustum. A naive approach would be to cut it into N uniform sliced view frusta using:

$$C_i^{uni} = z_n + (z_f - z_n)(i/N). \tag{3.3}$$

However, this means that subfrusta closer to the near plane of the camera receive the same shadow map resolution as points near the far plane. As we have discussed above, undersampling happens closer to the near plane and is less of an issue the further away we move from this plane. Thus, a logarithmic split scheme is a better solution for splitting the view frustum:

$$C_i^{log} = z_n(z_f/z_n)^{i/N}. \tag{3.4}$$

This split scheme on the other hand tends to result in oversampling close to the near plane and undersampling close to the far plane. Zhang et al. [17] thus introduced the *practical split scheme* (cf. Figure 3.13) that is based on the weighted average between the uniform and the logarithmic split scheme:

$$C_i^{pract} = \alpha C_i^{log} + (1 - \alpha)C_i^{uni} \tag{3.5}$$

with $\alpha \in [0, 1]$ as weight.

3.2.4 Filtering

In the previous section we dealt with aliasing resulting from the image based shadow mapping approach. We implemented techniques to improve the initial sampling when creating the shadow map but we haven't addressed any resampling errors when we render the camera view, using the created shadow map to sample points.

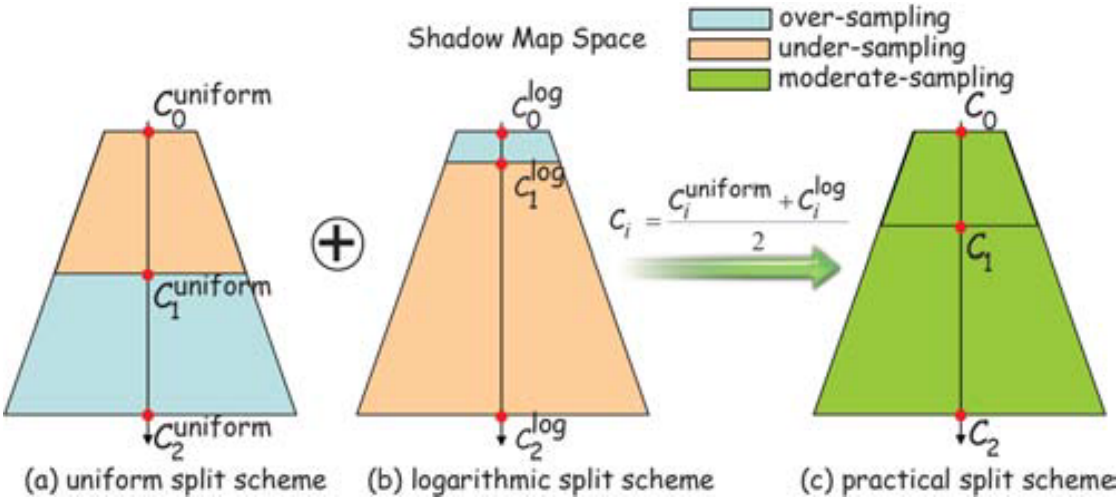

Figure 3.13: *Practical split scheme. Image courtesy of Zhang et al. [17]*

A typical approach to deal with these errors is called *percentage close fliter* (PCF) and was introduced by Reeves et al. [18]. Modern graphical hardware comes with a 2×2 bilinear PCF built in.

It is worth to mention that while filtered shadows give the illusion of soft shadows, they do not compare to physically based soft shadow techniques. That being said, they are a good trade off between computation and realism.

3.2.4.1 Poisson disk

In order to reduce perceived aliasing, Dippé and Wold [19], as well as Cook [20], and Mitchell [21] introduced methods to transform regular aliasing patterns into featureless noise. They found that sampling based on *Poisson disk distribution* yields one of the best result and makes aliasing patterns less noticeable.

For this work we decided to implement a Poisson disk filter with a precomputed distribution saved in the fragment shader file. Each shadow map is sampled a total of 16 times per fragment, leading to less aliased shadows in our models (cf. Figure 3.14).

3.3 Normal mapping

Although much of the focus for this work was put on enhancing shadows for Scenery3d, our overall goal was to improve rendering effects in general. One such effect that we implemented is *normal mapping* (also known as *bump mapping*).

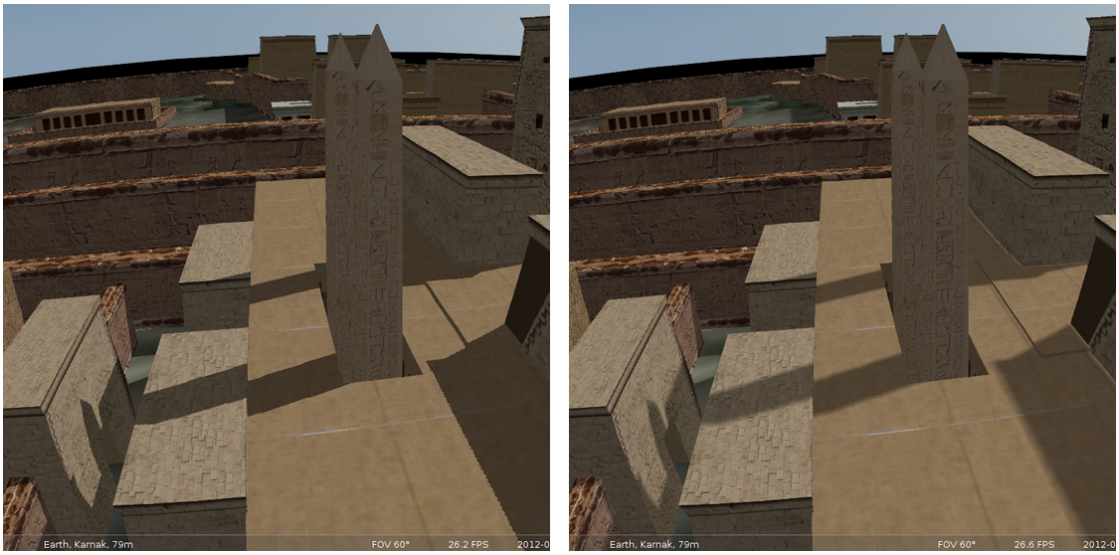

Figure 3.14: *Left: Hard shadows, no filtering. Right: Soft shadows through Poisson disk sampling*

At this time, rendering polygon meshes is the current standard in most real-time computer graphics applications. While it is possible to render many meshes with a very high polygon count − and thus high detail − it is often not recommended to do so.

In order to reduce polygon count but preserve details at the same time we make use of *normal maps* when determining the shading of a point. This allows us to alter the lighting of pixels on our textured object by looking the surface normal up in a RGB texture in which r, g and b map to the normal's coordinates x, y and z. By doing this, we can simulate the lighting of surface properties, e.g. bumps.

3.3.1 Lighting

It is important to know how we determine the intensity of a pixel in order to understand normal mapping. The following is a simplified version of a lighting equation, it covers only the diffuse part of the equation and is also known as Lambert reflection:

$$I = L_d * M_d * cos\theta. \tag{3.6}$$

L_d is the light source's diffuse color, M_d is the object's material diffuse color, θ is the angle between the normal and the light's direction vector. In order to compute $cos\theta$ we can use the dot product $N \cdot L$ where N is the normal vector and L is the light's direction vector.
Now, if we reduce a given model's polygons, we take away details because we smooth the models surface. In order to increase perceived details but keep the polygon count down, texture

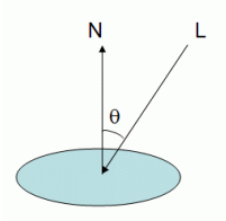

Figure 3.15: *A point's intensity depends on the angle between the surface normal and the light's direction vector. Image courtesy of Lighthouse3d.com [22]*

mapping is used. We place a simple *flat* texture onto a polygon to achieve faked detail.

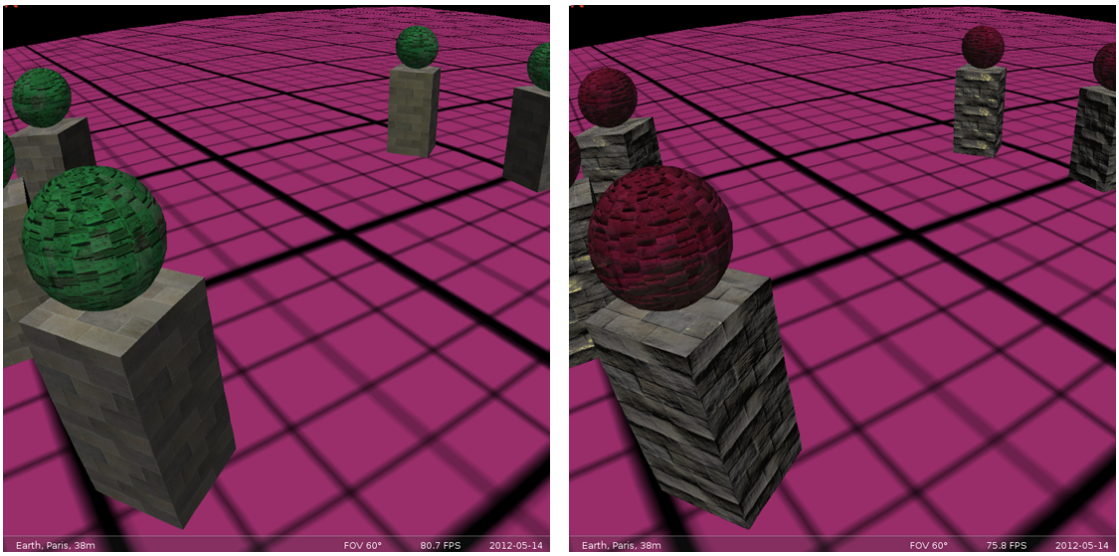

Figure 3.16: *Left: Standard texture mapping. Right: Normal mapping greatly enhancing the illusion of structure.*

However, the achievable perceived detail is very limited. This is especially visible when the textured models are close to the camera.

Lighting plays a major role in real-time rendering and as we have discussed above, a pixel's intensity depends on the angle between the light vector and the normal vector. While we can see surface details depicted in the texture, the computed light intensities contradict these notions often because the surface normals do not match the image information.

To achieve light intensities that simulate the given image information better, we supply a second texture – the normal map – to the shader. Each texel in the normal map represents a normal vector faking the direction of the surface at this point.

In our work, normal mapping helped us to improve realism at small additional cost (cf. Figure 3.16), however it remains a faking technique.

CHAPTER 4

Conclusion

Throughout this thesis we have discussed improvements added to *Scenery3d* – a plugin for Stellarium.

We started off with introducing *Stellarium*, a free open source planetarium software founded by Fabien Chéreau, followed by *Scenery3d* itself.

In chapter 2 we reviewed the state of the plugin before our improvements were added. We briefly discussed the plugin's architecture, the way it loads a scene from `.OBJ` files and how it renders the loaded scene. As mentioned before, the previous authors of Scenery3d added *shadow mapping* as an experimental feature to the plugin. However, it lacked a lot of refinement to be really usable.

We overhauled the plugin architecture, for example it is now possible to use shader programs. The main focus of chapter 3 – or in fact the entire thesis – was to improve the shadowing technique. For this, we looked at the existing technique in order to determine its reusability. However, not much was of use and we ended up rewriting the entire algorithm. That being said, we kept the original idea of using shadow mapping.

First, we implemented the basic algorithm for traditional shadow mapping as discussed in section 3.2.2. Then we identified the common problems related to this technique. Especially perspective aliasing was a key topic of interest for this thesis. In order to fight it, we first focused the shadow map to a convex body encompassing all shadow receivers as well as all potential shadow casters to guarantee correct shadowing even when shadow casters are outside of the camera view frustum.

Most of our test scenes were large outdoor scenes and as such, focusing alone was not enough to reduce perspective aliasing to our satisfaction. After some consideration, we decided to implement *Cascaded Shadow Maps* due to its simplicity and robustness. We allow up to four frustum

splits and the standard setup uses 1024×1024 sized shadow maps per subfrusta. This approach lead to decent hard shadows with little aliasing.

In order to reduce the remaining aliasing even further, we implemented Poisson disk filtering as mentioned in section 3.2.4.1. Our shadows appear less aliased and softer, however they do not represent physically modeled soft shadows.

Finally, to enhance the realism of our plugin, we implemented normal mapping at little extra cost.

4.1 Future work

Our overall goal was to improve the plugin's realism by implementing new and improving old real-time rendering effects. Future potential work should continue in this direction. To aid the expansion of effects, however, the plugin's architecture should be overhauled first.

Further reducing the shadow mapping aliasing is always desirable and worth looking into. Boosting realism should stay a priority and physically based soft shadows could help achieve higher realism. Shadows are still a popular research field and thus there is a lot of potential work in this direction, however, it is important to keep performance in mind.

Lighting is another major topic of relevance. A simple improvement would be to implement *Screen Space Ambient Occlusion* in order to approximate global illumination more. *Parallax mapping* could take the already implemented normal mapping to the next level with little additional cost, however *displacement mapping* might be worth looking into for an even better effect.

Finally, performance could be gained by implementing a culling algorithm like *CHC++* [23].

Bibliography

[1] Stellarium.org - free open source planetarium software, "http://www.stellarium.org/." Accessed: 2012-07-14.

[2] ASTROSIM. *Simulation of Astronomical Aspects of Middle Neolithic Circular Ditch Systems*, "http://astrosim.univie.ac.at/." Accessed: 2012-07-14.

[3] Wavefront Technologies OBJ geometry format specification, "http://www.martinreddy.net/gfx/3d/OBJ.spec." Accessed: 2012-07-14.

[4] Georg Zotti. *Scenery3d - Walkable 3D Models in Stellarium*, "http://bazaar.launchpad.net/~stellarium-scenery3d/stellarium/scenery3d/view/head:/plugins/Scenery3d/doc/Scenery3d.pdf." Accessed: 2012-07-14.

[5] L. Williams, "Casting curved shadows on curved surfaces," *SIGGRAPH Comput. Graph.*, vol. 12, pp. 270–274, Aug. 1978.

[6] paulsprojects.net - A collection of open source projects and tutorials., "http://www.paulsprojects.net/tutorials/smt/smt.html." Accessed: 2012-07-14.

[7] Eleni Maria Stea. *Socisplanet*, "https://code.launchpad.net/~hikiko/stellarium/socisplanet." Accessed: 2012-07-14.

[8] Lighthouse3d. *View Frustum Culling*, "http://www.lighthouse3d.com/tutorials/view-frustum-culling/." Accessed: 2012-07-15.

[9] K. Tadamura, X. Qin, G. Jiao, and E. Nakamae, "Rendering optimal solar shadows using plural sunlight depth buffers," in *Proceedings of the International Conference on Computer Graphics*, CGI '99, (Washington, DC, USA), pp. 166–, IEEE Computer Society, 1999.

[10] W. Engel, *Cascaded Shadow Maps, in Advanced Rendering Techniques*, vol. 5 of *ShaderX*. Charles River Media, Boston, Massachusetts, 2006. pp. 197-206.

[11] J.-M. Hasenfratz, M. Lapierre, N. Holzschuch, and F. X. Sillion, "A survey of real-time soft shadows algorithms," *Comput. Graph. Forum*, vol. 22, no. 4, pp. 753–774, 2003.

[12] M. Wimmer, D. Scherzer, and W. Purgathofer, "Light space perspective shadow maps," in *Rendering Techniques*, pp. 143–152, 2004.

[13] E. Eisemann, M. Schwarz, U. Assarsson and M.Wimmer, "Real-time shadows." A K Peters/CRC Press, 2011. 384 pages.

[14] S. Brabec, T. Annen, and H. peter Seidel, "Practical shadow mapping," *Journal of Graphics Tools*, vol. 7, pp. 9–18, 2000.

[15] M. Stamminger and G. Drettakis, "Perspective shadow maps," *ACM Trans. Graph.*, vol. 21, pp. 557–562, July 2002.

[16] M. Wimmer and D. Scherzer, "Robust shadow mapping with light space perspective shadow maps," in *ShaderX 4 – Advanced Rendering Techniques* (W. Engel, ed.), vol. 4 of *ShaderX*, Charles River Media, mar 2006.

[17] F. Zhang, H. Sun, L. Xu, and L. K. Lun, "Parallel-split shadow maps for large-scale virtual environments," in *Proceedings of the 2006 ACM international conference on Virtual reality continuum and its applications*, VRCIA '06, (New York, NY, USA), pp. 311–318, ACM, 2006.

[18] W. T. Reeves, D. H. Salesin, and R. L. Cook, "Rendering antialiased shadows with depth maps," in *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '87, (New York, NY, USA), pp. 283–291, ACM, 1987.

[19] M. A. Z. Dippé and E. H. Wold, "Antialiasing through stochastic sampling," in *Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '85, (New York, NY, USA), pp. 69–78, ACM, 1985.

[20] R. L. Cook, "Stochastic sampling in computer graphics," *ACM Trans. Graph.*, vol. 5, pp. 51–72, Jan. 1986.

[21] D. P. Mitchell, "Generating antialiased images at low sampling densities," in *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '87, (New York, NY, USA), pp. 65–72, ACM, 1987.

[22] Lighthouse3d. *Per pixel lighting*, "`http://www.lighthouse3d.com/tutorials/glsl-tutorial/directional-lights-i/`." Accessed: 2012-08-04.

[23] O. Mattausch, J. Bittner, and M. Wimmer, "CHC++: Coherent hierarchical culling revisited," *Computer Graphics Forum (Proceedings Eurographics 2008)*, vol. 27, pp. 221–230, Apr. 2008.