Silesian
University
of Technology

MASTER THESIS

Selected techniques of rendering shadow models in interactive computer graphics

Radosław Jacek RZECZKOWSKI
Student identification number: 290776

Programme: Informatyka
Specialisation: Interaktywna Grafika Trójwymiarowa

SUPERVISOR
Dr hab. inż. Agnieszka Szczęsna
DEPARTMENT Wydział Automatyki, Elektroniki i Informatyki
Faculty of Automatic Control, Electronics and Computer Science

Gliwice 2024

Thesis title

Selected techniques of rendering shadow models in interactive computer graphics

Abstract

(Thesis abstract – to be copied into an appropriate field during an electronic submission – in English.)

Key words

(2-5 keywords, separated by commas)

Tytuł pracy

Wybrane techniki renderowania modelu cieni w interaktywnej grafice komputerowej

Streszczenie

(Streszczenie pracy – odpowiednie pole w systemie APD powinno zawierać kopię tego streszczenia.)

Słowa kluczowe

(2-5 słów (fraz) kluczowych, oddzielonych przecinkami)

Contents

Chapter 1

Introduction

- introduction into the problem domain
- settling of the problem in the domain
- objective of the thesis
- scope of the thesis
- short description of chapters
- clear description of contribution of the thesis's author

Chapter 2

Problem analysis

2.1 The task of rendering

In this thesis, rendering is understood as the process of obtaining an image from a description of a three-dimensional scene. A scene can be rendered in a multitude of ways, with differences both in the specifics of the initial scene description and the rendered result. The rendered visuals can range from stylized to photorealistic. Rendering happens everywhere where a computer-generated imagery (CGI) is created for a viewer to see.

The wide spectrum of possible rendering results hints at the multitude of ways in which rendering itself is performed. There are various techniques employed during the rendering process, which can be utilized together to create a desired look and fit within performance constraints. In more complex processes, there are many techniques used to render a scene, each responsible for modeling the visual aspects of different real-life phenomena or artificial effects. A renderer could be capable of adding stylized edge detection and cell-shading to an image, rendering glossy and rough surfaces, simulating shadows, reflections, caustics and refraction, dealing with hair and fur or volumetric participating media such as fog, smoke and clouds. Each of the mentioned effects can be rendered using one of many techniques, and many of them are being actively improved upon and researched.

The fact that there are many techniques currently in use to render a single type of effect creates and advantageous situation, where the techniques used can be chosen for each application depending on its characteristic. Two main factors can be discerned as defining the needs of an application: the performance requirements and the desired visual style.

The desired visual style is partially just a matter of preference defined by the style of the project. More importantly however it is a matter of clearly and efficiently conveying visual information, in a way that is consistent with the rest of the application and with what the end user expects. This means that realism, often touted the pinnacle and goal of

computer graphics, is not necessarily always the best approach. A CAD (computer-aided design) application or a 3D modelling tool would be made less useful by including realistic reflections, highly contrasting full shadows and motion blur in the rendered viewport. These programs need to clearly convey information about the shape and design of 3D objects, without distractions and obstructions. On the other hand, when a player starts a modern action-adventure game, they expect a level of realism in the game's graphics that allows for immersion in the presented world. The intricate visuals also make for a more engaging experience and can mean a better reception of a game. At the same time, design choices should be made to ensure that the realism or intricacy of the presented graphics does not get in the way of enjoying the gameplay, which in the end should be the main attribute of a video game.

The performance requirements, or performance constraints, are usually better defined and divide rendering into two general categories: real-time rendering and offline rendering. When designing an offline renderer the performance constraints would most likely concern memory usage and possibly general time, or energy, efficiency, as with offline rendering the time it takes to render an image is of small importance. Most important are image quality and fidelity, often also realism. Techniques used in this context can spend as much time performing calculations as is necessary for the desired output. Offline rendering work can also easily be distributed between many machines, so-called render farms. Real-time rendering on the other hand works with very strict and small time budgets. Because a real-time application needs to be responsive to user inputs and give the illusion of continuous motion it is expected to render at least 30 frames per second (FPS), giving the time budget for a single frame of at most 0.03 seconds. This time cannot all be spent on rendering, as user inputs need to be handled and application logic performed in the same time frame. Because of that, real-time applications need to balance visual complexity and performance. They are also often created for more casual consumers than offline rendering programs, so the hardware on which the application will run is expected to be moderately powerful.

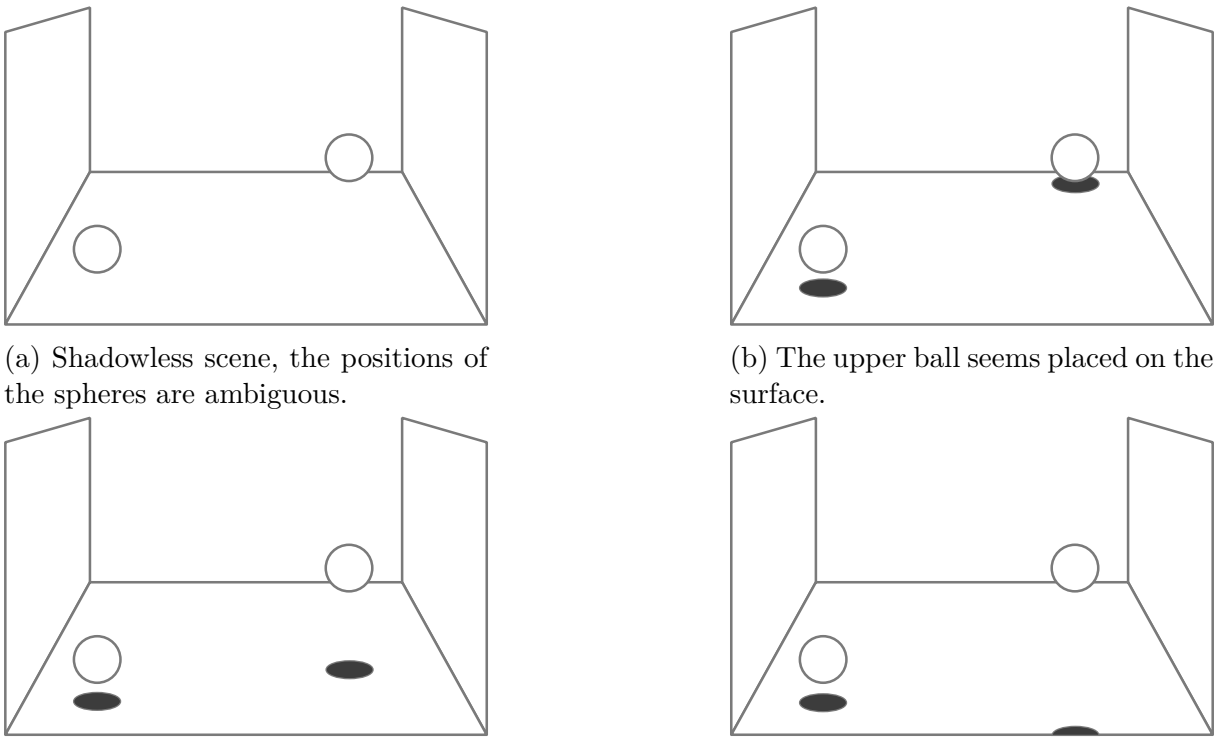
It is apparent that the choice of rendering techniques is complex and requires in-depth knowledge about each of them as early as the design stages of an application. This thesis aims to provide a guide through the rendering techniques used to render cast shadows in scenes, focusing on those applicable in real-time contexts. Some of the chosen techniques are the most widely adopted techniques, but some, while possibly less popular, are still noteworthy due to their promised outstanding performance or visual quality. This thesis introduces the techniques, gives detailed explanations of their algorithms, shows exemplary implementations and compares them in a series of tests. The comparison is made based on measures of performance (execution time), memory consumption, visual fidelity, realism and the amount of aliasing.

2.2 The role of shadows in computer generated images

Computer graphics usually aim to reflect our surrounding reality, with techniques for rendering phenomena modeled from physical laws and after observations made in the real world. When not aiming for realism, the displayed graphics need to be at least rooted in the common human observation experience for the information to be understandable and interpretable. Shadows are a phenomenon that is very commonly present in our daily lives, deeply rooted in the common experience. An 1997 article [6] shows that even simple shadows provide the viewer essential information on the position and movement of objects, especially movement in depth. In some cases this information even overrides other cues such as change of size due to perspective. This effect is illustrated by Fig. 2.1.

The importance of shadows for understanding the 3D composition of a scene is especially great when an observer is devoid of stereo visual information, such as in the case of observing an image displayed on a regular monitor. Shadows, lighting and perspective then become the sole sources of information about depth and the relative positions of objects in the scene.

These observations are also supported by a more recent 2018 review on the perception



(a) Shadowless scene, the positions of the spheres are ambiguous.

(b) The upper ball seems placed on the surface.

(c) The upper ball seems to hover halfway.

(d) The upper ball seems high in the air and closer to the observer.

Figure 2.1: The position and scale of the spheres do not change, yet the perceived locations in 3D space are dictated by the shadow.

of shadows [14]. It suggests more possible roles that shadows play in human perception of the world. Shadows help observers position objects in 3D space. Sometimes shadows are perceived as being a part of the object, so the lack of a shadow would make the object incomplete. Information about the shape of an object can also be derived from the shadow that it casts, as presented in Fig. 2.2.

(a) The left shape could be a sphere, the right shape could be flat, like a coin.

(b) The left shape could be a cube, and the right a shape with top and bottom faces not being parallel.

Figure 2.2: The shadow can convey information on the shape of the object. It additionally describes whether an object is placed on a surface or above it.

The review also mentions that shadows in art, even if depicted in a way that is impossible in the real world, are an important element for an image to be perceived as realistic and of high quality. This also applies to computer graphics, making rendering shadows a necessary element for any application that requires elevated realism or sophistication of the visual presentation. The difference between a scene with and without cast shadows is presented in Fig. 2.3 It is worth noting that since observers are not great at spotting physically incorrect shadows, as is stated in the studies, so graphics programmers can utilize this fact and take liberties with the realism of their shadows. This can lead to simplifying the algorithms producing them and improving their performance, with negligible impact on the apparent visual quality.

2.3 Shadow rendering techniques

2.3.1 Naming conventions

In this thesis the term ‘shadows’ is used interchangeably with term ‘cast shadows’, not to be confused with shading on the surface of objects, which stems from the orientation of the surface with regard to the light source.

As presented in the “Real-Time Rendering” book [1], a shadow is cast by a shadow caster, also called an occluder, onto a shadow receiver, when the caster occludes a line of sight from a point on the surface of the receiver to a light source.

Shadows can be differentiated into hard and soft shadows. Perfectly hard shadows are not observed in reality, as such a shadow would require an infinitely small light source. Hard shadows however are often found in computer-generated images, as they are relatively simple and cost-efficient to render. Soft shadows require more complex solutions, but

(a) The scene rendered with no cast shadows, despite the high quality of models and materials, looks unrealistic and flat.

(b) The scene rendered with cast shadows is realistic and reads well.

Figure 2.3: The same scene rendered twice using Blender's Cycles renderer, with and without cast shadows. *Scene source: "Classroom" by Christophe Seux, Blender demo files.*

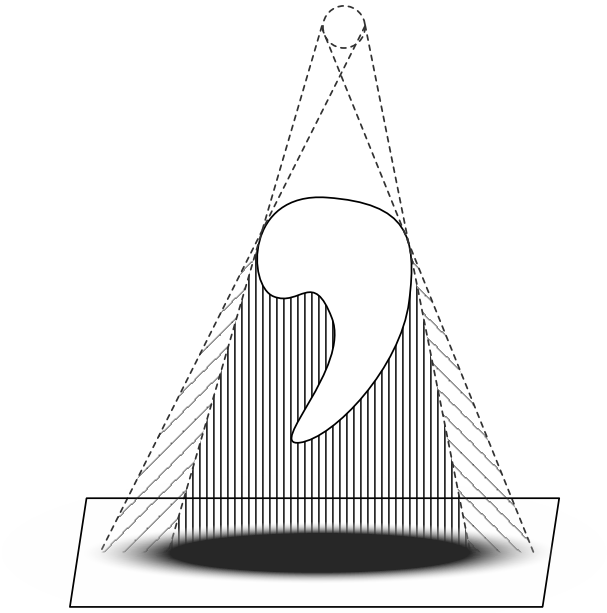


Figure 2.4: The umbra (under vertical lines) and penumbra (under slant lines) create a soft shadow.

enhance the realism of the rendered scene. A soft shadow, as shown in Fig. 2.4 consists of an *umbra*, being the fully shadowed region, and *penumbra*, meaning the region that is partially in shadow. The appearance of a soft shadow, the sizes of the umbra and penumbra defining its softness, depend on the relative distances between the shadow caster, the shadow receiver, the light source and also the size of the light source.

2.3.2 Coarse distinction of shadow rendering techniques

In this thesis a coarse distinction of the shadow rendering techniques is proposed, dividing them into three groups. The division is decided based on the main mechanism utilized in a technique. Additionally, techniques in each group share similar issues that researchers have attempted and are still attempting to resolve with newer iterations of the original algorithms.

The first group consists of shadow map-based, or image-based, techniques. They render the scene from the viewpoint of the light and store how far away scene geometry is in a depth buffer. This is then used when rendering the actual view of the scene to determine which parts are in shadow and which are lit. A common issue for these techniques is aliasing, which occurs when signals are sampled at different rates, causing artifacts. In the case of computer graphics, spatial aliasing is most often observed, where the source signal, being geometry, a texture, or a depth buffer as in the case of shadow mapping, is sampled in discrete locations in space at varying sample densities. It is not trivial to counteract aliasing in shadow map techniques and there are many approaches to solving this problem. On the other hand, it is relatively simple to achieve approximate soft shadows, with more

advanced techniques allowing for more realistic results.

The second group contains geometry-based techniques. These approaches modify the scene geometry to create shadow volumes, which contain areas where light from a source does not fall. The shadow volumes are then rendered with a clever use of a stencil buffer, producing the information whether a surface point is in light or not. Such techniques suffer less aliasing problems than image-based ones, producing perfectly sharp shadow information on a per-pixel basis. They however introduce a lot of new geometry to the scene, the rendering time of which can be large and difficult to predict. It is also more difficult to achieve the effect of soft shadows with this approach.

The third group are ray-based techniques. They have only recently started being utilized in real-time scenarios, as they can be computationally costly and require powerful, modern hardware to run at interactive rates. This group of techniques is possibly the most intuitive, as it does not really rely on clever utilization of the rasterization hardware of a graphics card. To determine if a surface point is in shadow, a ray is cast in a straight line from the point to a light source. If any geometry is encountered along the ray, then the light source is occluded and the point is in shadow, otherwise it is lit. These techniques, apart from relying on powerful hardware, utilize specialized scene description data structures that accelerate these ray queries. They do not suffer from aliasing and can be extended to give photorealistic and physically-correct results.

The following sections introduce the techniques based on the publications which first described them. The sections attempt to mention them in logical order, highlighting the improvements and additions made to each group over the years. More detailed descriptions of the algorithms used to implement these methods will be given in chapter 3. Additionally, the next section presents a now largely unused approach, that does not really fit into any of these categories, but is useful to build a geometric intuition of how shadows form.

2.3.3 Planar shadows

Planar shadows are one of the first techniques used for implementing shadow rendering and are not very robust, so they are mentioned in this thesis for historic reasons. They can be a help with understanding the geometry involved with shadows and their shape. That is because planar shadows are rendered using basic geometric projection.

Introduced by James Blinn in 1988 [3], planar shadows are created by projecting the shadow caster onto a flat surface of the shadow receiver. The caster vertices are projected along a vector between the vertex and a light source onto a planar surface of the receiver. This is achieved with a projection matrix, which multiplied with the vertex positions of the caster gives a new mesh, flattened onto the planar surface. This mesh can then be rendered on top of the shadow receiver with a black material, giving the illusion of a shadow. The technique is described in more detail in section 3.2.1.

While very simple and intuitive, this approach has many shortcomings that nowadays render it obsolete. The application needs to keep track of shadow casters and receivers separately, to know which objects should be projected onto which surfaces. Issues occur when a light source is between the caster and receiver. In such a situation an incorrect anti-shadow is created, as the operation of projection through a point is still valid. The receivers have to be planes, and special care needs to be taken to only draw the shadow on top of a receiver. Since the shadow is a separate mesh, it could be rendered beyond the mesh of a receiver. Additionally, objects cannot shadow themselves. On the other hand planar shadows give a lot of artistic control over the appearance of shadows and even their shape, as they can be manipulated just as any other mesh.

2.3.4 Basic shadow mapping

The general algorithm for shadow mapping was initially described by Lance Williams in 1978 in the article "Casting curved shadows on curved surfaces" [18]. The title immediately hints at the biggest strengths of this technique at the time: it is robust, allowing any scene objects to cast hard shadows onto others, regardless of their shapes and without the need for special treatment of casters and receivers as with planar shadows. In fact, similar shadow mapping techniques that utilized the z-buffer were already known before this article, but they could only be used for geometry consisting of planar polygons. This technique can be used to render shadows anywhere where any scene rendering can be obtained at all, which includes smooth, curved parametric surface patches.

The article presents the general method of shadow mapping, along with its advantages and disadvantages. The rationale of the technique is based on the fact that all surfaces 'seen' by a light source from its point of view are lit by it, while all others are in shadow. To determine surface visibility for a light source, the scene is simply rendered from the viewpoint of the light source. This rendering is called the shadow map. Such an intermediate rendering, which will not be presented on screen but is only a step in the process of creating the final render of a scene, is often called a render pass. This rendering does not need to produce a color image, but only a depth- or z-buffer. This buffer is used in the standard rasterization pipeline to perform depth culling, so no special implementation is needed. Moreover, the existing highly optimized and hardware-supported rasterization architecture is utilized. The z-buffer stores the distances from the observer to the closest surface point in the scene per-texel. Once this view is obtained, the scene can be rendered again, in the usual manner from the point of view of the observer. When deciding whether a surface point is in shadow or not, the distance between the point and the light source can be calculated and compared to the value stored in the previously obtained shadow map. If the distance of the surface being rendered to the light source is greater than the depth read from the shadow map, then the surface is in shadow, and some other object that

is closer to the light source occludes it. Otherwise, the surface is lit and further shading computations can take place, such as diffuse and specular lighting calculations.

Variants of shadow mapping have been described over the years, and techniques based on this approach are currently most widely utilized for real-time rendering. This is not surprising when considering the advantages of this method. The issues present in planar shadows such as the need to separate shadows casters and shadow receivers, difficulty casting shadows onto non-planar surfaces, impossibility of obtaining self-shadowing and possibility for shadows to appear in empty space beyond any receivers are all solved with shadow mapping. The technique is also relatively simple to implement and utilizes the rasterization pipeline in its full potential. The cost of the algorithm per light source is roughly twice the cost of rendering without shadow mapping, as scene objects need to be transformed and rendered into the z-buffer. The cost of rendering the shadow map will however be almost always less than rendering the scene, as there are no per-pixel lighting computations done to obtain color. In modern rendering pipelines the color render target buffer can be omitted altogether, making it also possible to not run the pixel shader at all.

The disadvantages of shadow mapping are outweighed by its advantages, but are not trivial to combat. One disadvantage is that basic shadow mapping generates only hard shadows, which are not realistic and are prone to visible aliasing in the final image. Multiple specializations of this technique exist that make it possible to render soft shadows with varying levels of realism and correctness. Another significant issue is the fact that this technique introduces a second level of aliasing to the shadows. This is because the shadow map itself is an image representation of the scene, making it a discrete view of continuous scene geometry. This creates a few problems that need to be addressed. For one, numerical representation resolution errors can be introduced into the shadow map. This can be easily minimized by rendering into a 16-bit or more z-buffer and setting the near and far clipping planes of the shadow map view to tightly encompass the scene. The second problem being self-shadowing, also called surface-acne, is much more prominent and immediately observable in a naive implementation. As the samples of depth values stored in the shadow map are texels, they represent a non-zero area for a point location on a surface. When the exact depth values calculated when rendering the main view of the scene are compared with these spatially discretized samples, a surface can erroneously shadow itself, creating high-frequency, high-contrast patches of shadow, which often create intrusive moiré patterns on screen. This problem can be reduced by using a higher resolution shadow map, resulting in more dense depth sampling and decreasing the size of self-shadowing patches. This however is expensive and will not fully resolve the problem. Another method to deal with surface-acne is to use a depth bias when rendering the shadow map. A depth bias is a value that is added to the actual depth values before storing them in the z-buffer. By adding a positive depth bias, the depth of a surface can

be pushed further away from the light source, under the actual surface. This way the actual distance between the light source and the surface being rendered in the final pass will always be less than the depth stored in the shadow map. This however can cause issues with light-leaking, or peter-panning, where contact shadows get detached from their casters. Methods to counteract this, as well as an example implementation will be shown in chapter 3.2.2. More issues with shadow mapping will be described in the following sections, along methods to circumvent them.

2.3.5 Fitting shadow maps

Fitting shadow maps is the simplest technique that can be used to reduce shadow map aliasing problems, by better utilization of the available shadow map resolution and precision. The main idea of fitting is to avoid rendering into the shadow map objects that are not necessary for the current scene and viewpoint. Initially introduced by Brabec [4], the method was developed over time and can be implemented in different levels of sophistication, more complex ones giving better results.

The simplest way to fit a shadow map is to tightly encompass the entire scene with the view frustum of the light source. This can work well enough for small scenes, but will be problematic for large or densely populated scenes.

One can set the light frustum width and height to encompass just the frustum of the viewer. Additional care needs to be taken to actually include all shadow casters that may cast shadows onto the scene objects present in the view frustum, as they may not necessarily be within it. This concerns only the near and far clipping planes of the light, and can be simply achieved by setting them to the extents of the scene.

There are more advanced techniques, including using the depth information of the scene observer to analyze the distribution of future shadow map samples in space and produce light frusta that tightly bind the rendered geometry [8]. This requires a depth map of the scene from the observer's point of view to be available when rendering the shadow map, which can introduce additional performance cost but can also be hidden if the rendering pipeline already requires a depth pre-pass.

Fitting, since it depends on the view position and direction, can be temporally unstable. When the viewport changes there can be noticeable flickering on the edges of the shadows, caused by temporal aliasing. This can be minimized by quantizing the offset of the shadow map that is used when fitting to texel increments.

2.3.6 Warping shadow maps

Warping shadow maps was introduced by Stamminger [15] and aims to counteract perspective aliasing. Perspective aliasing means that more samples of the shadow map are taken closer to the viewer due to perspective foreshortening, meaning that a higher con-

centration of shadow map samples is required there than further away. A warped shadow map accomplishes exactly that with minimal performance impact by using a specific parametrized perspective transformation on the scene geometry before rendering into the shadow map. The mentioned technique is called perspective shadow maps (PSM). More approaches were build basing on this proposition, namely light space perspective shadow maps (LiSPSM) [19] and trapezoidal shadow maps (TSM) [11]. Both these techniques are very similar and give better results than PSM. It has been proven that LiSPSM actually gives best results out of the three as it leads to the best distribution of error over the shadow map area. [10]. These warping techniques can actually be used in tandem with shadow map partitioning approaches.

2.3.7 Filtering shadow maps

While the former techniques were mostly meant to counteract initial sampling aliasing by utilizing the resolution of the shadow map in the best possible way, shadow map filtering methods are mainly meant to help reduce aliasing during the resampling stage. Resampling happens when the shadow map is actually used to determine shadowed and lit areas in the final render, when the appearance of the shadows is determined. Usually to counteract such aliasing various filtering methods are employed. This is also the case in many areas of computer graphics, including sampling of different types of textures. Filtering can get rid of high frequency components from processed signals, such as pixelated edges in textures. Color textures can be filtered in simple and intuitive ways, for example by using linear filters or mipmaps.

A shadow map is special in the context of filtering, because of its specific content. It actually cannot be filtered directly. When shadow map filtering is considered more carefully, it becomes apparent that it is not the depth values that need to be filtered, but the actual shadow information obtained based on these depths from the shadow map and depths of points in view. A shadow map filtered by, for example, linear interpolation would contain depths of non-existing surfaces, witch is not the desired outcome. Additionally, the shadows themselves would still be sharp and aliased, because the depth comparison function has binary output: a pixel is either in shadow or not.

Filtering shadow mapping has been approached in a few ways. One of them actually focuses on filtering the shadow information itself in the final render. The percentage-closer filtering (PCF) [13] algorithm does exactly that. The shadow computation is done in the regular way, but it is performed multiple times per pixel. A filter window and a kernel are used to sample the shadow map at offsets around the point that is being shaded, and the binary comparison results are averaged to obtain a filtered shadow. More robust techniques based on PCF improve the visual quality and performance, such as using specialized kernel weights and sampling patterns, for example a Poisson disk. The

Poisson disk sampling pattern allows using fewer samples and covering a larger area. It can also be randomly rotated to avoid banding artifacts. Another one of such approaches was described by Yury Uralsky in the GPU Gems article "Chapter 17. Efficient Soft-Edged Shadows Using Pixel Shader Branching" [12]. It uses specifically generated arrays of sample offsets that are arranged in such a way that samples further away from the origin are first in the array. This way, when performing PCF, only the outer samples can be tested first. If their values agree, meaning the point is either fully in shadow or fully in light, the calculations can stop, allowing to omit most of the actual sample count. This early test-out provides significant performance gains. The technique also deals very well with banding, which is obvious in basic PCF. These techniques can be further improved to give smoother results by utilizing percentage-closer filtering that is built into the graphics hardware. By setting an appropriate sampler for the shadow map, bilinear PCF, with a $2x2$ kernel, can be achieved at virtually no additional performance cost. PCF is further described in section 3.2.4.

Another way to approach filtering shadow maps is to utilize mathematical and statistical principles to describe the shadow signal in a way that can be precomputed and filtered. One of the first such techniques were variance shadow maps (VSM) [5]. Mean and variance of the shadow map values are computed and later used to calculate the upper bound of the fraction describing how much of the receiver is occluded. This can lead to light leaks near object edges, which introduce high variance in the depth values, but significantly improves performance by allowing traditional filtering methods to be carried out on the depth map. Many improved techniques have been developed basing on this achievement. One of them are layered variance shadow maps (LVSM) [7], which deal with light leaks by modifying the stored depth values to introduce less variance.

2.3.8 Rendering soft shadows with shadow maps

Here techniques for obtaining soft shadows are described.

2.3.9 Basic shadow volumes

Here geometry/stencil based techniques begin, starting with stencil-based, hard-shadow, per-triangle shadow volumes.

Chapter 3

Subject of the thesis

3.1 Test application

3.2 Implemented shadow rendering techniques

Below, all the shadow rendering techniques that have been implemented during work on this thesis are described, along with explanations of their algorithms.

3.2.1 Planar shadows implementation

Planar shadows, as described in section 2.3.3, are a very simple technique to achieve hard shadows, introduced by Blinn [3]. The idea for the algorithm stems from the geometric understanding of how a shadow is formed. The shape of a shadow is basically the shadow caster projected onto the surface of a shadow receiver along rays from a light source, as shown in Fig. 3.1.

Using this geometric understanding and similar triangles, marked in Fig. 3.1 in dashed lines, the following equations and a projection matrix can be derived. Assuming l to be the position of a light source, v to be the position of a vertex of the shadow caster and p the position of the projected shadow vertex, a projection onto the $y = 0$ plane is described by:

$$\frac{p_x - l_x}{v_x - l_x} = \frac{l_y}{l_y - v_y} \iff p_x = \frac{l_y v_x - l_x v_y}{l_y - v_y} \tag{3.1}$$

The z component is derived in analogous way:

$$\frac{p_z - l_z}{v_z - l_z} = \frac{l_y}{l_y - v_y} \iff p_z = \frac{l_y v_z - l_z v_y}{l_y - v_y} \tag{3.2}$$

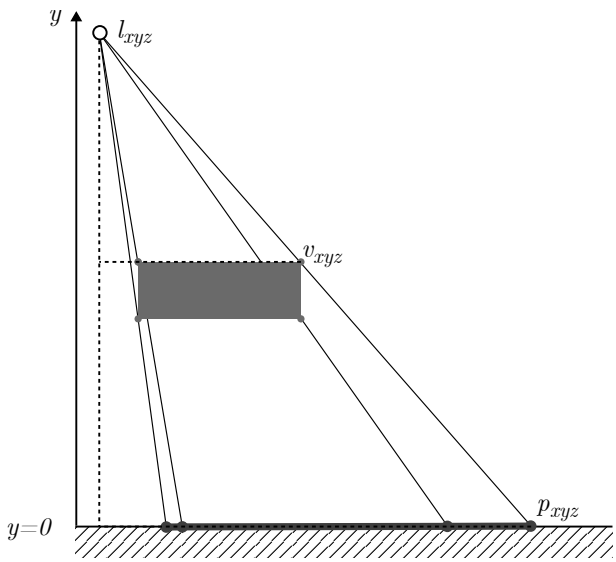


Figure 3.1: Vertices of a shadow caster projected onto $y = 0$ plane create a planar shadow. Dashed lines visualize the similar triangles.

The y component will be $y = 0$ since the projection happens onto the xz plane. From that the projection matrix can be constructed:

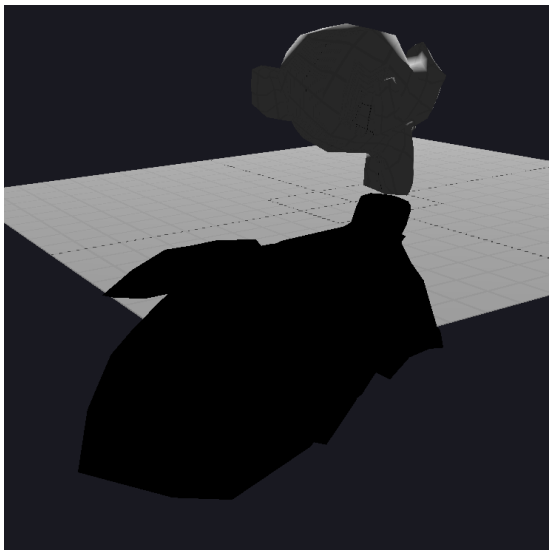
$$\mathbf{M} = \begin{pmatrix} l_y & -l_x & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & -l_z & l_y & 0 \\ 0 & -1 & 0 & l_y \end{pmatrix} \tag{3.3}$$

The $l_y - v_y$ factor in the denominator is obtained by utilizing the property of homogenous coordinates, where all components of a vector $\mathbf{v} = \begin{bmatrix} x & y & z & w \end{bmatrix}^T$ get divided by the w component, which gives $\mathbf{v} = \begin{bmatrix} x/w & y/w & z/w & 1 \end{bmatrix}^T$.

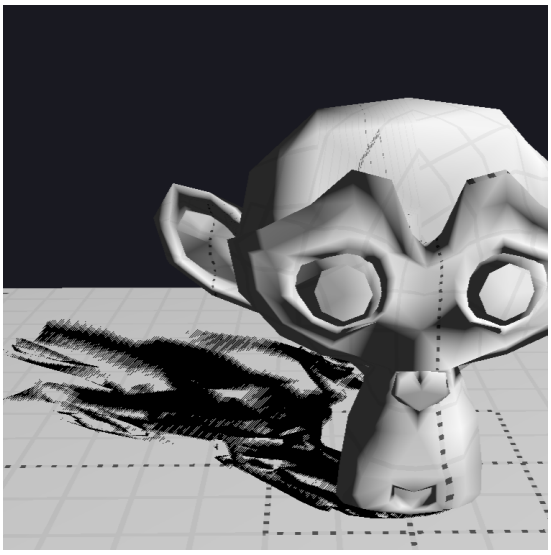
This matrix can be generalized to project points onto any plane, but this was not implemented.

In the implementation the matrix is created in the vertex shader based on the position of the light source. The vertices of the caster, after being transformed into world space, are multiplied by it, projecting them onto the $y = 0$ plane. Then they are transformed with the view and perspective projection matrices as usual. The resulting mesh is rendered in black, with a pixel shader that just returns the black color. Special care should be taken when rendering the shadow mesh, since it perfectly overlaps the ground plane at $y = 0$. To avoid z-fighting, which is shown in Fig 3.2a, this mesh needs to be rendered after the receiver and before the caster, with z depth testing turned off in the rendering pipeline, or, as is the case in the implementation, a negative depth offset needs to be added to the mesh to bring it above the ground plane.

Multiple issues arise with this technique. The receiver must be a flat plane, objects



(a) The projected shadow mesh lies beyond the ground plane.



(b) Severe z-fighting caused by the fact that both the ground plane mesh and shadow mesh lie in the exact same plane.

Figure 3.2: Some of the issues that can arise when using projection shadows.

cannot shadow themselves, casters and receivers need separate treatment. Additionally, care needs to be taken to only render the shadow meshes on top of the geometry of a receiver. Since the shadow is a separate mesh, it could be cast beyond the actual surface of the receiver, creating an impossible shadow in the air. This is illustrated in Fig 3.2b There is another possibility here for an impossible shadow. If the light source is between the caster and receiver, an anti-shadow will be cast as shown in Fig 3.3. This creates another special case that needs to be checked for by the application. Soft shadows are also difficult and costly to achieve, created by projecting the caster geometry multiple times using different discrete locations sampled over the area of the light source. The shadow meshes can then be blended, producing a darker color where more shadows are present. This gives very accurate results with a high number of samples, but is not a viable option for real-time rendering. All these problems make this technique virtually unused in modern rendering engines.

3.2.2 Implementation of basic shadow mapping

As mentioned in chapter 2.3.4, shadow mapping for a single light source is performed using two render passes. The first one generates the shadow map itself, the second and final pass renders the scene with shadows determined using the shadow map. The shadow map is a z-buffer, which means that, for each texel, it stores the values of the z coordinate of the scene surface point being rendered. These coordinates are stored after transforming the vertices of the scene into the NDC space, or normalized device coordinate space, of a given light source. This transformation is achieved in the vertex shader, by using the

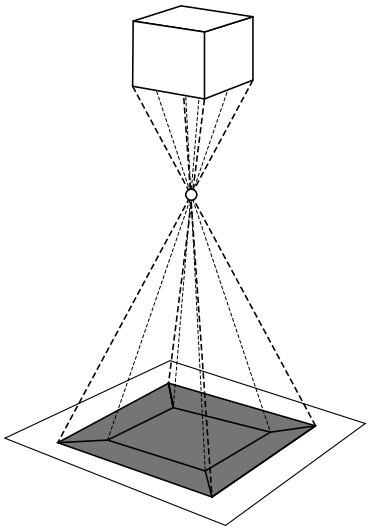


Figure 3.3: The shadow caster projected through a light source, creating an anti-shadow.

regular world, view, projection (WVP) transformation matrices, with the distinction that the view and projection matrices are not of the scene observer but of the light source. In truth, multiplying the vertex positions by the WVP matrix will result in clip space homogenous coordinates, which then will be transformed automatically by the pipeline into NDC during the perspective division, when the homogenous coordinates are divided by their w component. The stored depth values will then be in the $0 - 1$ range, usually 0 being at the near clipping plane and 1 at the far clipping plane. This z-buffer can then be used as a texture shader resource in a pixel shader to render the final scene. To read the depth values from the shadow map, the NDC coordinates of each surface point need to be calculated again. The x any y components will then be used to access a texel in the shadow map, the z component will be compared with the contents of the shadow map. Do note that the actual Euclidean distance between the shaded point and the light source is not used, as these values would be in different coordinate spaces. When the shadow map value is accessed, it is compared with the z value of the point that is currently being shaded. If the shadow map value is greater, then the point being shaded is farther away from the light source than some other surface seen at this point from the point of view of the light. This means that it is not illuminated and is in shadow. Otherwise, the point is lit by the light source. The process is illustrated in Fig. 3.4. Fig. 3.5 shows a rendered view of a scene with shadow mapping and the shadow map itself.

To implement this technique a resource is needed that will be used both as an additional depth buffer and a texture. This can be easily achieved in an application programming interface (API) like DirectX 12 by creating one resource and two resource views, one being a Depth Stencil View (DSV) and another a Shader resource view (SRV). Once the rendering of the shadow map is finished using the DSV, the resource can be transitioned in preparation to be used as an SRV during rendering the main view using a resource barrier. Caution needs to be taken to transition the resource back to a DSV-compatible

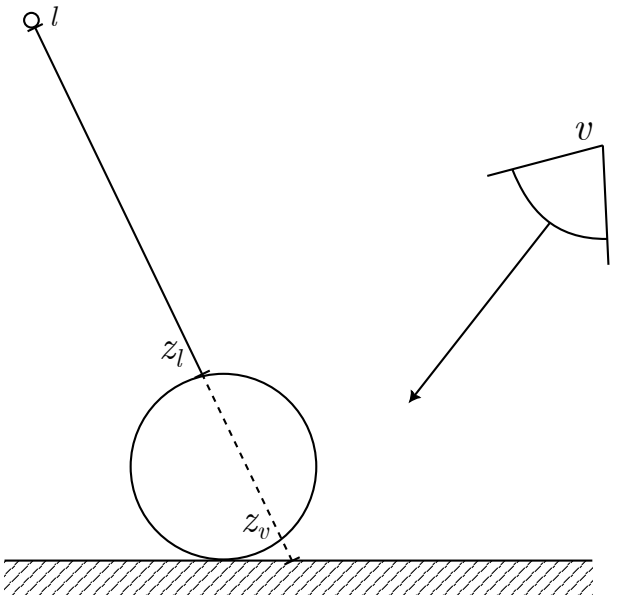
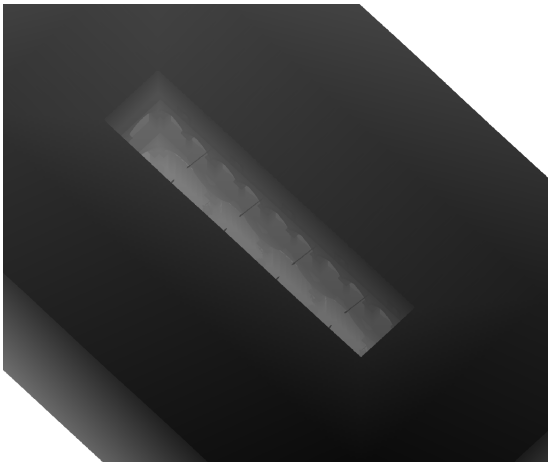


Figure 3.4: The light source l has stored the depth value z_l in its shadow map. When rendering the scene from the viewpoint of the observer v, the stored depth is compared to the depth z_v. This value will be larger than the one in the shadow map, so the point will be in shadow.

(a) A scene rendered with the use of shadow mapping.



(b) Section of the shadow map rendered for this scene.

Figure 3.5: An example scene rendered with a shadow map, which is shown in black and white.

state before rendering the shadow map of the next frame. In this case, a resource barrier will also ensure that all write operations have finished before continuing with rendering, meaning that the rendering of the shadow map will be complete before it is used in the final pass. The shaders responsible for rendering the main scene view will evidently need access to the texture resource containing the shadow map, but also it will need to access the view and projection matrices of the light view pass. This is to calculate the corresponding NDC coordinates of the point being shaded in the shadow map. It is optimal to calculate light clip space coordinates in the vertex shader and let the pipeline interpolate them across the triangle for each fragment. Since the perspective division will not happen automatically here, it is needed to divide the coordinates by the w component in the pixel shader to obtain correct NDC values. Since the shadow map is sampled as a texture, the xy coordinates with range $[-1:1]$ need to be converted into uv texture coordinates with range $[0:1]$. Additionally, the y component needs to be flipped to address the texture correctly. This can be performed as a simple transformation in the pixel shader as $\begin{bmatrix} x & y \end{bmatrix}^T / \begin{bmatrix} 2 & -2 \end{bmatrix}^T + \begin{bmatrix} 0.5 & 0.5 \end{bmatrix}^T$ or incorporated into the MVP matrix of the light view. This can be achieved by multiplying the MVP matrix for the light source by the offset matrix as shown in equation 3.4.

$$\mathbf{L_{offset}} = \begin{pmatrix} 0.5 & 0 & 0 & 0.5 \\ 0 & -0.5 & 0 & 0.5 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \mathbf{MVP} \tag{3.4}$$

When sampling the shadow map as a texture, it is important to do so with a sampler that does not perform any filtering on the texture. Typically, texture filtering helps avoid different kinds of aliasing. Regular color textures however can be filtered and interpolated as they only store color data. The shadow map, which stores depth values, cannot be filtered in the same way as the produced values would be senseless in this process. They would create non-existing apparent surfaces especially noticeable between sharp depth changes, for example between a close object and the far clipping plane.

This observation brings forth one of the issues with shadow mapping: they cannot be filtered to avoid aliasing. One could also attempt to blur a shadow map to achieve soft shadows, but this is not a viable option for the same reason. Because of this basic shadow mapping can only be used to create hard shadows that suffer from aliasing issues, creating shadows that can appear blocky when up close, as in Fig. 3.6. Aliasing can be mitigated by rendering the shadow map into a higher resolution target, but only to a degree. Special techniques allowing for smoothing and filtering shadows generated with shadow mapping will be presented in the following sections.

When rendering the shadow map an appropriate projection matrix has to be construc-

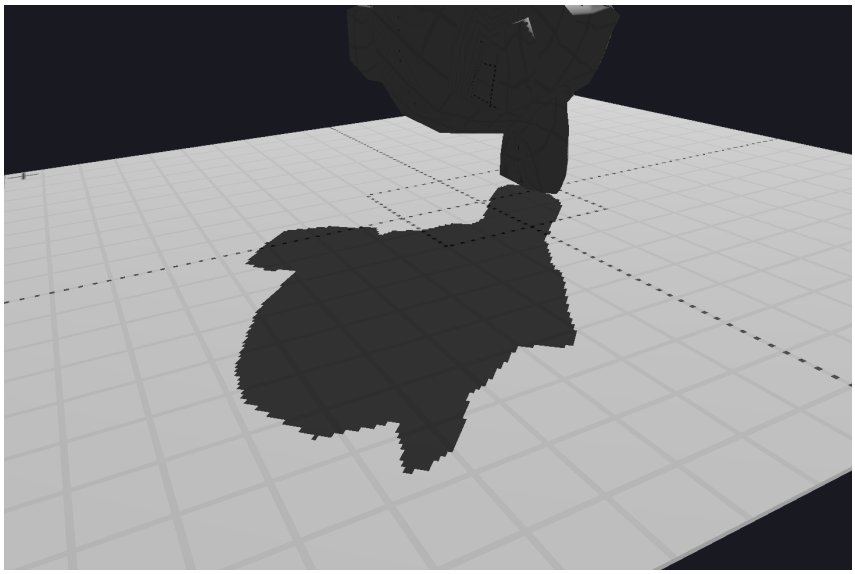
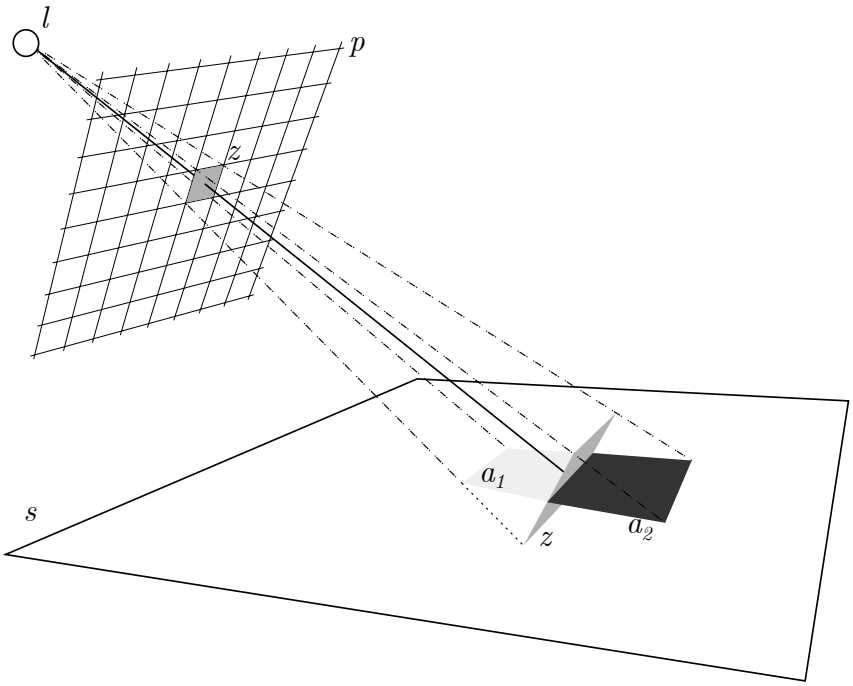


Figure 3.6: Aliased shadow appearance due to no filtering and low resolution of the shadow map.

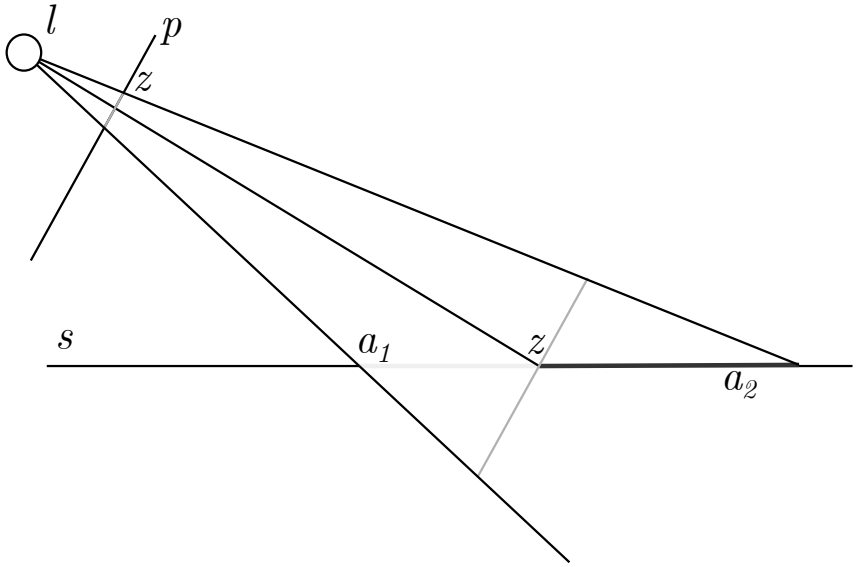
ted. For a far away directional light, such as a sun light, an orthographic projection should be used. When rendering a spotlight a perspective projection can be used. Point lights can also use a single shadow map with a perspective projection matrix, but only in situations when the light source is positioned outside the scene, or at least is not surrounded by shadow casters. In the opposite case, the most common technique is to use cubemaps and improved derivative techniques as described by Liang Wan [16]. A cubemap consists of six shadow map renders, each covering a face of a cube constructed around the light source. This way an omnidirectional shadow map is created.

When creating a projection matrix as well as the view matrix for a shadow map, care has to be taken to capture all relevant shadow casters. This is especially true for large scale scenes and directional lights, as rendering the entirety of a scene into the shadow map would most of the time be unnecessary and suboptimal. Most of the rendered map would not be used in a given frame, and the distribution of shadow map resolution would be poor. It would be enough for the light view to tightly encompass the view frustum of the observer to obtain correct results. This is called fitting and will be presented in the following sections. This idea can be taken further as shown by Jiří Bittner [2] to allow for an approximate FPS gain of 1.5 times. Their method uses a custom advanced algorithm to cull shadow casters and render only those that contribute to the current main viewpoint.

Another issue of shadow maps is self-shadowing, or surface acne. This is also a result of aliasing, since the continuous scene geometry and its depth data is sampled in a discrete manner into an image when rendering the shadow map. If the shadow map texel does not correlate one-to-one with the final view samples, which happens very often and is highly likely, the shadow map samples will cover a non-zero area on the geometry surface in the final view of the scene, which is illustrated in Fig. 3.7. This can cause the surface



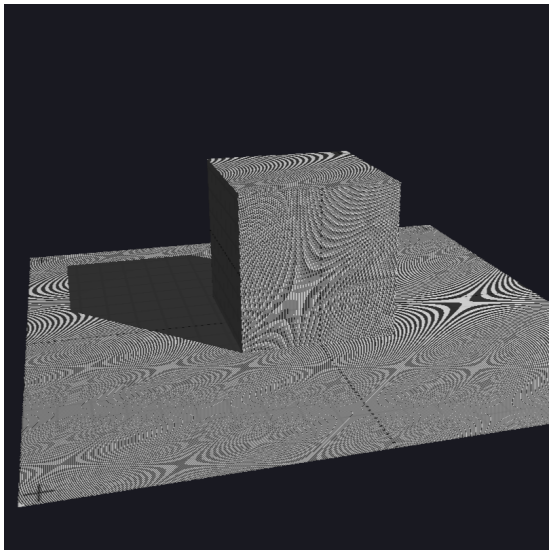
(a) Depth samples being compared with depth values over an area of the actual mesh lead to aliasing and shadow acne.



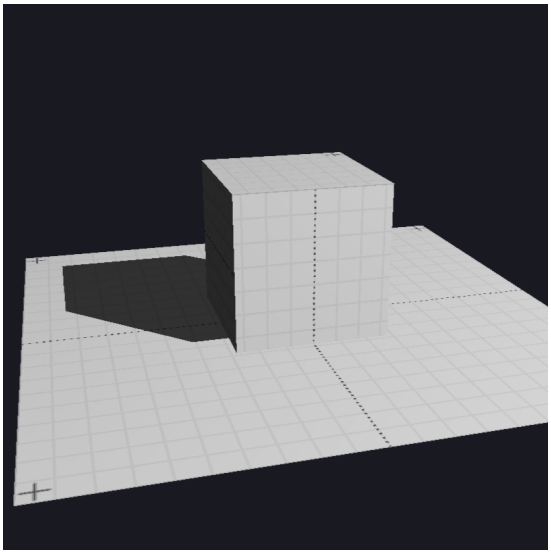
(b) The aliasing issue present in shadow maps illustrated in two dimensions from the side.

Figure 3.7: The cause of shadow acne. Light source l renders the shadow map onto its projection plane p, having some finite resolution. The sampled depth value z is stored in a texel. This creates a virtual surface orthogonal to the light direction on top of the actual geometry surface s. When rendering the scene, depth values in the area a_1 will have a lower depth value stored in the shadow map texel than their actual depth, causing them to be in light. Samples in the a_2 area will have greater depth, causing them to be evaluated as being in shadow.

to erroneously shadow itself, as shown in Fig. 3.8a. This issue gets more pronounced the higher the angle between the light source view ray and the surface normal. It can be resolved by adding a depth bias when rendering the shadow map, which will ensure the samples are below the actual surface. In modern rendering APIs the depth bias can be configured for a pipeline and applied automatically when rendering. Especially useful is the combination of a constant depth bias and a slope-scaled depth bias. The second bias helps eliminate the problem when surfaces appear more edge-on in the light view by scaling the depth bias with the angle between the light direction and surface. Too high depth bias can cause peter-panning, a situation shown in Fig. 3.8b where a shadow caster gets detached from its shadow. Because of this, depth bias values often need to be hand-tuned on a per-scene basis. Other approaches have also been proposed, such as using only the back faces of objects to render the shadow map [17] or utilizing depth peeling to store average depths between the back and front faces of an object [20], but they both still require some bias to avoid shadow-acne in areas such as silhouette edges or concavities.



(a) Shadow acne due to using not enough bias.



(b) The cube is disconnected from its shadow due to too much depth bias.

Figure 3.8: Some of the issues that can arise when using projection shadows.

The z-buffer of a rendered image can be actually non-linear in its $[0-1]$ range. Perspective projection matrices are sometimes built in such a way that the precision closer to 1 is lower than closer to 0. This is purposefully done to gain more precision closer to the viewer, where it is more needed when performing hidden-surface removal than far away, where objects will become very small due to perspective foreshortening. This can be an issue with shadow mapping, since objects close to the light source are not necessarily close to the viewer. In fact the opposite can often be true. As shown by Stefan Brabec [4], the z-buffer for the use in shadow mapping can be linearized using a linear transformation dependent on the projection matrix used. The following perspective projection matrix

present in the DX Math library can be linearized with equation 3.7.

$$\begin{pmatrix} W & 0 & 0 & 0 \\ 0 & H & 0 & 0 \\ 0 & 0 & \frac{f}{f-n} & \frac{-fn}{f-n} \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} Wx \\ Hy \\ \frac{zf}{f-n} - \frac{wfn}{f-n} \\ z \end{pmatrix} \Rightarrow \begin{pmatrix} Wx \\ Hy \\ \frac{zf}{f-n} - \frac{fn}{f-n} \\ z \end{pmatrix} \quad \text{for} \quad w = 1 \tag{3.5}$$

$$\begin{pmatrix} Wx \\ Hy \\ \frac{zf}{f-n} - \frac{fn}{f-n} \\ z \end{pmatrix} /z = \begin{pmatrix} \frac{Wx}{z} \\ \frac{Hy}{z} \\ \frac{f}{f-n} - \frac{fn}{z(f-n)} \\ 1 \end{pmatrix} \tag{3.6}$$

After the perspective division the final form of the z-value equation is obtained. Below, in Fig. 3.9, the z-values are plotted for exemplary near and far clipping planes equal $n = 2$, $f = 15$.

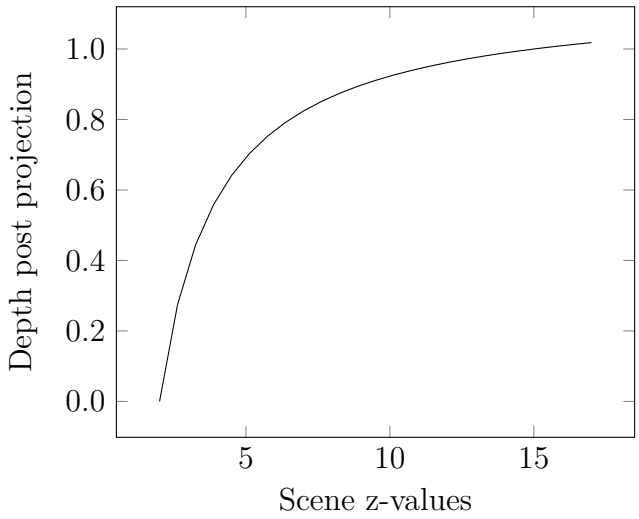


Figure 3.9: The z-values after a perspective projection become non-linear.

The values can be linearized back again simply by multiplying the clip-space z-value by $\frac{z}{f}$, which gives the following equation.

$$\left(\frac{f}{f-n} - \frac{fn}{z(f-n)}\right) * \frac{z}{f} = \frac{z-n}{f-n} \tag{3.7}$$

In the application z-buffer linearization was not implemented, as after investigation it turned out that an orthographic projection, which is used for the directional light, does not introduce nonlinearity to the z-buffer range. This can be seen by inspecting how the matrix will affect the w component. The orthographic projection matrix as implemented in the DX Math library is presented below.

$$\mathbf{M_o} = \begin{pmatrix} W & 0 & 0 & 0 \\ 0 & H & 0 & 0 \\ 0 & 0 & \frac{1}{f-n} & \frac{-n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.8}$$

This matrix does not modify the w component and the z values will remain linear. In fact the formula for depth values after using this matrix is exactly the same as the perspective one after linearization.

3.2.3 Fitting shadow maps

An attempt to implement LiSPSM was made, but was not successful due to poor coverage of the implementation details both in the original paper and on the internet.

Only a part of the algorithm was implemented, basically providing simplified fitting of the shadow map view. The shadow map camera has certain extents set up by hand for each scene, such that the whole scene fits within view. Since the entire scene is not necessarily viewed at once by the observer, only the part that is within the observer view frustum can be rendered to the shadow map. Therefore, the shadow map post-projection space can be adjusted to include the area viewed by the observer.

This can be achieved by building a fitting transformation matrix $\mathbf{F}$ that includes translation and scaling, that is applied after the light view and projection matrices, resulting in $\mathbf{F}\mathbf{P}_L\mathbf{V}_L$. To obtain $\mathbf{F}$, first the observer view frustum extents need to be found in world space, which can be obtained by applying the inverse of observer view and projection matrices to an NDC cuboid. This can be a cube with ranges $[-1:1]$ in x, y, z in APIs like OpenGL, or a cuboid with ranges $[-1:1]$ in x and y, and $[0:1]$ in z in DirectX. Once the vertices of this cuboid are moved into world space, they can be multiplied by the view and projection matrices of the light source, $\mathbf{P}_L\mathbf{V}_L$, and their components divided by w to bring them into the post-projection space of the light source. Once there, a bounding box can be found enclosing these points. To not move the contents of the shadow map beyond the original extent specified for the light source, the components of the vertices of this bounding box can be clamped to the NDC range. The whole process is shown in Fig. 3.10. Finally, the minimum and maximum x and y values of these bounding vertices

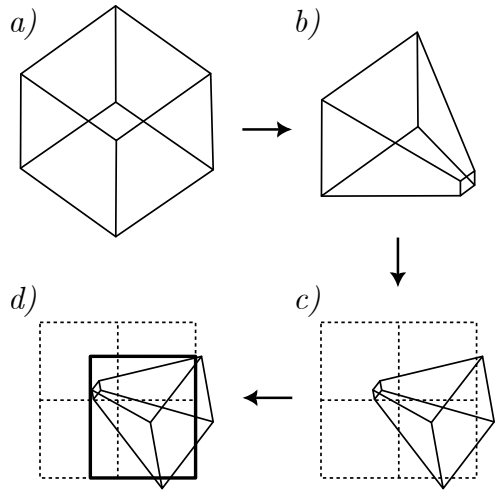


Figure 3.10: The steps involved in fitting the shadow map. The NDC cuboid in a) is in observer post-projection space, or clip space. It is moved to the state presented in b) using inverse matrices, resulting in a world space representation of the viewing frustum. It is them transformed into c), the light view clipping space. Finally, a clamped bounding box is found, shown in d).

can be found and used in the following fitting matrix.

$$\mathbf{F} = \begin{pmatrix} s_x & 0 & 0 & o_x \\ 0 & s_y & 0 & o_y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \text{where} \tag{3.9}$$

$$s_x = \frac{2}{x_{max} - x_{min}} \quad \text{and} \quad o_x = \frac{-s_x(x_{max} + x_{min})}{2} \tag{3.10}$$

The calculation of s_y and o_y is analogous.

As was implemented, the fitting focuses on an approximate convex body of $\mathbf{L} \cap \mathbf{V}$, where $\mathbf{L}$ stands for the light view frustum and $\mathbf{V}$ for the viewer frustum. The shadow map is then fitted only in the clip space x and y dimensions, ignoring depth z. A more robust solution would be to find the convex hull of $\mathbf{L} \cap \mathbf{V} \cap \mathbf{S}$, with $\mathbf{S}$ meaning the scene bounding box. Then the shadow map would only focus on areas within the viewer frustum where there exists geometry to render. The near plane of the light frustum can then be set to the upper scene extent and the far plane to the farthest viewer frustum point. The shadow map would then require no hand adjustments to its frustum and would adapt to different scenes and viewer orientations.

3.2.4 Filtering shadow maps with PCF

As established in section 2.3.7, percentage-closer filtering performs filtering of the shadow information to avoid aliasing artifacts. Basic shadow mapping performs a depth

comparison giving a binary result on whether a scene point is in shadow of some other occluder or is directly in view of the light source. This comparison can be described with a formula $s(t, z) = H(d(t) - z)$, where function $s(t, z)$ is the shadow function that takes texture coordinates t and the depth value z of a scene sample in light space. Function $d(t)$ is the depth lookup into the shadow map and H is the Heaviside function. This binary result can be obtained for positions near the initial scene sample and averaged, to get a value in the $[0 : 1]$ range describing how much light the point receives. This makes the technique relatively simple to implement in a renderer that already uses shadow mapping.

Possibly the simplest way to get PCF results is to use the hardware implementation made available by modern graphics APIs. This is accessible via the settings of the sampler that will be used with the shadow map. In DirectX 12 one can specify a linear comparison filter operation and a comparison function when creating a sampler. When later used in an HLSL shader, the SampleCmp function should be used. It takes the value to compare against as one of its arguments. It will perform comparison according to the comparison function with four nearest texels and bilinearly interpolate the results. This is done at practically no additional performance cost.

Unfortunately, hardware bilinear PCF is not going to significantly improve the shadow mapping visuals. In most cases there will still be visible aliasing, which is due to the fact that the kernel used is very small, being only 2×2. A larger kernel can be built manually and used in the same manner to get smoother shadows. A simple box filter with an $n \times n$ kernel can cover a larger area in the shadow map. The larger the kernel, the smoother the shadow and the larger the penumbra. This can be implemented by generating an array of offsets either in the shader or, to preserve computing power, on the CPU and later sent to the GPU via a uniform buffer. The offsets should be in texel-aligned increments, which is easily achieved by expressing them in multiples of $1/s$, where s is the shadow map size. These offsets are then added to the scene sample position expressed in light space. It is important to note, that even though different texels are sampled, the initial scene sample z value remains the same. Unfortunately, the visual quality of such filtering is visibly sub-par, as pixelated banding artifacts occur. This is due to the fact that only $n^2 + 1$ shades can be obtained in the penumbra. The kernel size could obviously be raised, but sizes even as relatively low as 11 have a significant impact on performance. Hardware PCF can be used with this method to smooth out the banding, and an offset scale can be utilized to control the size of the penumbra regardless of the size of the kernel.

A more robust and efficient approach allows using fewer samples while giving good visual results. Instead of using a box filter it creates sets of randomly jittered offsets that are used when performing PCF. Most importantly, these offsets are stored in memory in such a way that allows to decide whether a point is fully lit or fully in shadow without having to test all samples. The offsets are first generated on the CPU in a jittered grid pattern of $n \times n$ size. Their coordinates are normalized, so all samples lie within the $[0 : 1]$

range in the x and y axes. Then, all the samples are moved from the grid into a disk domain using the following equations.

$$u = \sqrt{y} \times \cos(2\pi x) \tag{3.11}$$

$$v = \sqrt{y} \times \sin(2\pi x) \tag{3.12}$$

The angular position of the sample on the disk is then determined by x and the y defines the radial distance from the center. If the loop that generates these points and stores them into an array is constructed in a smart way, the highest y coordinates are generated first, so the offsets the furthest away from the center are first in the array. This is then utilized in shader code when looping over the sample offsets and performing PCF: only the first n samples are tested at first, and if they all result in either 0 or 1, meaning the point is entirely in shadow or in light, further PCF processing for this point can stop. This delivers a significant performance boost, saving processing power where computations do not contribute to the visual outcome. The full set of samples will only be used in penumbras. The fact that all offsets have been normalized means that they can be multiplied by a scalar scale factor to control the amount of smoothing, or the size of the penumbra.

Unfortunately, the problem of banding in penumbra regions observed in simple box filtered PCF still persists. Hardware PCF can again be used to smooth the results further, but there is another solution to this problem. Instead of using a single set of jittered sample offsets, $m \times m$ such sets can be generated and stored in a three-dimensional texture. In the shader this texture is accessed using the screen pixel coordinates, to choose a set of offsets, and then the set is indexed to perform PCF. This way banding is stochastically smoothed out in the final result. Creating an offset set for each pixel of the final image would consume a lot of memory and not provide significant gains, so small sets, such as 16×16 are used and tiled over the whole render area. This is enough to generate sufficient noise in the penumbra areas to create a smooth appearance.

As an argument against this technique one could point out the fact that GPU hardware does not handle dynamic branching well, and that it should be avoided to maintain the best performance. While this statement is generally true, it is not fully accurate. On modern GPUs the parallel threads that perform the computations run in groups, partially determined by their screen space positions, called warps or wavefronts. This is well explained and visualized by Nvidia in their developer article on the logical structure of the rendering hardware [9]. Since the branching in this algorithm depends on whether a pixel is in the penumbra region, it is dependent on screen space position and penumbra points will likely be in warps together, making the optimization effective.

A substantial problem of PCF algorithms is surface acne, which gets worse the larger the kernels and offset scales are used. This is because the sampled area of the shadow map increases and read depth values may vary more, while the depth of the scene point being

shaded does not change. This can be again combated with a depth bias. An offset cone can be also used by applying higher bias the further from the original sample a point is. These however still need to be hand-tuned on a scene by scene basis. Additionally, it is important to note that, while PCF approaches do generate smooth shadows, they are not physically correct soft shadows. This is both due to the fact that the amount of smoothing is constant and that scene surface points are sampled. For an accurate shadow the area of the light source would need to be sampled for visibility from a given scene point, not the other way around.

Chapter 4

Experiments

4.1 Methodology

One of the objectives of the thesis was to compare multiple shadow rendering methods both in a qualitative and quantitative ways. To achieve this a test renderer application was created, which implements the rendering techniques and allows a user to observe their results as well as profile the performance the program.

The tests were performed in a semi-controlled environment. Care was taken to avoid other processes interrupting and affecting the test results, but their impact could not be entirely eliminated in. Results of repeated tests at different moments in time or on different machines with the same hardware will differ, but that difference should not be large enough to impact the overall comparison results. The machine used for performing measurements had the following specification:

- OS: Windows 10, 22H2
- CPU: AMD Ryzen 5 1600
- GPU: Nvidia GTX 1080, 8 GB VRAM
- RAM: 16 GB

4.1.1 Created test application

The test application allows the user to choose a rendering mode and observe the results in real time. It also allows to change the viewpoint and observe how the shadows behave in motion. Most rendering modes implement different shadow rendering techniques and some include debug views like wireframe mode or mesh colored by normals. The application is also instrumented with profiling commands which make it possible to connect to an external profiler and collect data in real time, as well as save and load existing profiling data.

The application was written in the C++ programming language. It uses the Win32 platform to create windows in a Windows operating system. It uses DirectX 12 to utilize the GPU for rendering graphics. DirectX 12 was chosen because it is the newest of DirectX versions, allowing for granular control over the rendering process and boasting advanced features. It is also very widely adopted for all sorts of computer graphics tasks, making it worth specializing in. Additionally, the DirectX 12 API is much more verbose and involved as compared to, for example, OpenGL, which forces the programmer to gain more in-depth knowledge about graphics programming and all the processes involved.

Two notable open-source libraries were utilized in the project. Dear ImGui is a library simplifying the creation of graphical user interfaces. It is compatible with all the major graphics programming APIs, including DX12, making it fast and easy to integrate into a project, providing am intuitive and smooth user experience at the same time. The other library is the Tracy Profiler. The instrumentation library is just a part of the Tracy ecosystem, with separate applications like the profiler to gather and analyze performance data. It is simple to incorporate into an existing project and very performant, adding almost no overhead. Code zones and frames that will be captured when profiling are marked manually by the programmer, giving them full control over what data is captured. This also improves the readability of the outputs, which stands in contrast with other profilers that automatically gather data and can easily overwhelm the user and clutter the presentation with the amount of information provided.

4.2 Data sets

The data sets consist of multiple scenes used for testing.

4.3 Results

The results of testing each implemented shadow rendering method are presented in the following sections. Each set of results is complemented by a description and discussion, highlighting possible causes and, if appropriate, possible fixes. The techniques might undergo different tests based on their characteristic.

4.3.1 Planar shadow mapping

4.3.2 Basic shadow maps

Chapter 5

Summary

- What problem have I solved?
- How have I solved the problem?
- What are pros and cons of my solutions?
- Can I state some recommendations?
- synthetic description of performed work
- conclusions
- future development, potential future research
- Has the objective been reached?

Bibliography

[1] Thomas Akenine-Moller, Eric Haines and N. Hoffman. *Real-Time Rendering, Fourth Edition*. Boca Raton: Taylor & Francis Ltd, 2018. ISBN: 978-1-1386-2700-0.

[2] Jiří Bittner, Oliver Mattausch and A. Silvennoinen. 'Shadow caster culling for efficient shadow mapping'. In: *Symposium on Interactive 3D Graphics and Games*. 2011, 81—88.

[3] J. Blinn. 'Me and My (Fake) Shadow'. In: *IEEE Comput. Graph. Appl.* 8.1 (1988), 82—86.

[4] Stefan Brabec, Thomas Annen and H.-P. Seidel. 'Practical Shadow Mapping'. In: *Journal of Graphics Tools* 7.4 (2002), pp. 9–18.

[5] William Donnelly and Andrew Lauritzen. 'Variance shadow maps'. In: *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*. I3D '06. Redwood City, California: Association for Computing Machinery, 2006, 161–165. ISBN: 159593295X. DOI: 10.1145/1111411.1111440. URL: https://doi.org/10.1145/1111411.1111440.

[6] Daniel Kersten, Pascal Mamassian and D. Knill. 'Moving Cast Shadows Induce Apparent Motion in Depth'. In: *Perception* 26.2 (1997). PMID: 9274752, pp. 171–192. DOI: 10.1068/p260171. eprint: https://doi.org/10.1068/p260171. URL: https://doi.org/10.1068/p260171.

[7] Andrew Lauritzen and Michael McCool. 'Layered variance shadow maps'. In: *Proceedings of Graphics Interface 2008*. GI '08. Windsor, Ontario, Canada: Canadian Information Processing Society, 2008, 139–146. ISBN: 9781568814230.

[8] Andrew Lauritzen, Marco Salvi and Aaron Lefohn. 'Sample distribution Shadow Maps'. In: Feb. 2011, pp. 97–102. DOI: 10.1145/1944745.1944761.

[9] *Life of a triangle - NVIDIA's logical pipeline*. URL: https://developer.nvidia.com/content/life-triangle-nvidias-logical-pipeline (visited on 14/10/2024).

[10] Brandon Lloyd, David Tuft, Sung-eui Yoon and Dinesh Manocha. 'Warping and Partitioning for Low Error Shadow Maps'. In: Jan. 2006, pp. 215–226. DOI: 10.2312/EGWR/EGSR06/215-226.

[11] Tobias Martin and Tiow-Seng Tan. 'Anti-aliasing and Continuity with Trapezoidal Shadow Maps.' In: Jan. 2004, pp. 153–160. DOI: 10.2312/EGWR/EGSR04/153-160.

[12] Matt Pharr and Randima Fernando. *GPU Gems 2: Programming Techniques For High-Performance Graphics And General-Purpose Computation*. Addison-Wesley Professional, 2005.

[13] William T. Reeves, David 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: Association for Computing Machinery, 1987, 283–291. ISBN: 0897912276. DOI: 10.1145/37401.37435. URL: https://doi.org/10.1145/37401.37435.

[14] Paulo E. Santos, Roberto Casati and P. Cavanagh. 'Perception, cognition and reasoning about shadows'. In: *Spatial Cognition & Computation* 18.2 (2018), pp. 78–85. DOI: 10.1080/13875868.2017.1377204. eprint: https://doi.org/10.1080/13875868.2017.1377204. URL: https://doi.org/10.1080/13875868.2017.1377204.

[15] Marc Stamminger and George Drettakis. 'Perspective shadow maps'. In: *ACM Trans. Graph.* 21.3 (2002), 557–562.

[16] Liang Wan, Tien-Tsin Wong and C. Leung. 'Isocube: Exploiting the Cubemap Hardware'. In: *IEEE Transactions on Visualization and Computer Graphics* 13.4 (2007), pp. 720–731.

[17] Yulan Wang and Steven Molnar. *Second-Depth Shadow Mapping*. Tech. rep. USA, 1994.

[18] Lance Williams. 'Casting curved shadows on curved surfaces'. In: *SIGGRAPH Comput. Graph.* 12.3 (1978), 270—274.

[19] Michael Wimmer, Daniel Scherzer and W. Purgathofer. 'Light space perspective shadow maps'. In: *Proceedings of the Fifteenth Eurographics Conference on Rendering Techniques*. EGSR'04. Norrköping, Sweden: Eurographics Association, 2004, 143–151. ISBN: 3905673126.

[20] Andrew Woo. 'The shadow depth map revisited'. In: *Graphics Gems III*. USA: Academic Press Professional, Inc., 1992, 338—342. ISBN: 0124096719.

Appendices

Technical documentation

List of abbreviations and symbols

CGI	computer-generated imagery
CAD	computer-aided design
FPS	frames per second
PSM	perspective shadow maps
LiSPSM	light space perspective shadow maps
PCF	percentage-closer filtering
VSM	variance shadow maps
LVSM	layered variance shadow maps
NDC	normalized device coordinates
WVP	world–view–projection
API	application programming interface
DSV	Depth Stencil View
SRV	Shader resource view
N	cardinality of data set
μ	membership function of a fuzzy set
$\mathbb{E}$	set of edges of a graph
$\mathcal{L}$	Laplace transformation

List of additional files in electronic submission (if applicable)

Additional files uploaded to the system include:

- source code of the application,
- test data,
- a video file showing how software or hardware developed for thesis is used,
- etc.

List of Figures

List of Tables