FAKULTÄT FÜR !NFORMATIK

Accurate Soft Shadows in Real-Time Applications

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computergraphik/Digitale Bildverarbeitung

eingereicht von

Michael Schwärzler
Matrikelnummer 0325222

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung:
Betreuer: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Betreuender Assistent: Univ.Ass. Dipl.-Ing. Mag.rer.soc.oec. Daniel Scherzer

Wien, 20.02.2009

_____	_____
(Unterschrift Verfasser)	(Unterschrift Betreuer)

Erklärung zur Verfassung der Arbeit

Michael Schwärzler
Anzengrubergasse 49, 2380 Perchtoldsdorf

"Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe."

Ort, Datum: _____ Unterschrift: _____

Abstract

In this thesis, the generation and use of *soft shadows* in real-time rendering is discussed. While *hard shadow* algorithms are already widely used in games and applications, the fast and correct calculation of soft shadows, which are not cast by a point light source but by an area light source, is a complex task and still an area of active research. The simulation of soft shadows is worth the increased effort, though: Nearly every shadow in reality has soft boundaries, so using soft shadows in rendering applications significantly increases the realism of the generated images.

After giving an explanation on where soft shadows come from and what problems occur during their computation, current real-time soft shadow algorithms are presented and discussed. Nearly all of them are based on either the *shadow mapping* or the *shadow volumes* algorithm, which are extended in various ways to work together with area light sources. Still, none of them is able to compute *physically correct* soft shadows in real-time for arbitrary scenes: Either the calculation takes too long, or the soft shadow is only approximated.

Thus, we suggest a new approach, which uses *Temporal Coherence* between frames in order to generate physically accurate soft shadows: By generating only a single shadow map each frame from a different, random sampling position on the area light source and storing the shadowing information in a screen space *shadow buffer*, it is possible to compute exact shadows while achieving frame-rates as high as in single-sample fake approaches.

Kurzfassung

In dieser Diplomarbeit wird die Berechnung und Verwendung von weichen Schatten in Anwendungen der Echtzeitgraphik behandelt. Während harte Schatten bereits häufig in Spielen und Programmen eingesetzt werden, ist die Verwendung von weichen Schatten, die anstelle von Punktlichtquellen von Flächenlichtquellen erzeugt werden, aufgrund der viel höheren Berechnungskomplexität noch eher selten und stellt deshalb auch immer noch ein aktives Forschungsgebiet dar. Der Mehraufwand, den die Verwendung von weichen Schatten mit sich bringt, lohnt sich aber dennoch: Beinahe jeder Schatten, den wir in unserem alltäglichen Leben vorfinden, ist weich, weshalb wir computergenerierte Bilder mit weichen Schatten als sehr viel realistischer wahrnehmen.

Nach einer Erklärung, was weiche Schatten sind, warum sie auftreten, und welche Schwierigkeiten sich bei ihrer Berechnung ergeben, werden aktuelle Algorithmen vorgestellt, die in der Lage sind, weiche Schatten in Echtzeit zu berechnen. Sie basieren stets auf einem der beiden bekanntesten Algorithmen für harte Schatten, *shadow mapping* oder *shadow volumes*, und wurden auf verschiedenste Arten und Weisen erweitert, um die Generierung von weichen Schatten zu ermöglichen. Dennoch ist es mit keinem der vorgestellten Methoden möglich, für beliebige Szenen *physikalisch korrekte* weiche Schatten in Echtzeit zu erzeugen: Sie sind entweder zu langsam, oder approximieren die Schatten nur.

Wir stellen deshalb einen neuen Ansatz vor, der mithilfe von *Temporal Coherence* die Berechnung von physikalisch korrekten weichen Schatten ermöglicht: Durch die Generierung einer einzigen *shadow map* pro Frame, die jeweils von einer anderen, zufälligen Stelle der Flächenlichtquelle erzeugt wird, sowie der Verwendung eines sogenannten *shadow buffers*, der die bisherigen Schattenergebnisse speichert, ist es möglich, exakte weiche Schatten zu berechnen, und dabei überzeugende Bildwiederholraten zu erreichen.

Acknowledgements

Many thanks to all the people at the *The Institute of Computer Graphics and Algorithms* at the *Vienna University of Technology* who assisted and helped me during my studies; in particular to Michael Wimmer and Daniel Scherzer, who both supervised my work on this master thesis and gave me lots of useful hints and comments.

Many thanks also to the members of the *Computer Graphics Club* for their support, ideas, help, criticism and all the parties we had together! :-) Particularly Alexander Kusternig helped me a lot during the development of the DirectX 10 framework and with many enlightening conversations.

Special thanks to all the university colleagues who helped me on my way through all the exams and exercises. I would like to primarily thank Heinrich Fritz and Martin Knecht for all the discussions, the excellent collaboration and – most important – the great friendship! :-)

For financial support, which made it possible to acquire the needed hardware, I thank the *Faculty of Informatics* and its professors at the *Vienna University of Technology*.

Finally, i thank my family and my girlfriend Susanne Schöffmann for their support, help and patience! :-)

Contents

A false friend and a shadow attend only while the sun shines.

(*Benjamin Franklin)*

Chapter 1

Overview

This chapter provides a first glance at the Chapters and outlines their contents. The thesis is structured in five main parts, in which the following topics are discussed:

Chapter 2 of this thesis gives an *introduction* into the fields of real-time graphics, shadow algorithms and rendering methods - the "basics" needed to understand the following Chapters.

Chapter 3 will describe what so-called *soft shadows* are, why and how they occur, how they differ from *hard shadows*, and what difficulties arise during their implementation.

In Chapter 4, an overview of today's *real-time soft shadow* algorithms is given: A vast amount of such algorithms have been published during the past few years. Most of them are based on either the shadow mapping or the shadow volumes algorithm, but they still differ a lot: While some focus on generating physically accurate but computationally costly soft shadows, other techniques use "fake methods" to achieve perceptually plausible, but faster results.

In Chapter 5, we present our own real-time soft shadow algorithm, called *Real-Time Soft Shadows using Temporal Coherence*. It combines the speed of single-sample fake methods with the physical accuracy of approaches where multiple samples of the area light source are taken. We do this by exploiting the *Temporal Coherence* of consecutive frames.

Chapter 6 summarizes and concludes this thesis.

Shadow reveals the true shape of a body.

(*Leonardo Da Vinci)*

Chapter 2

Introduction

In this chapter, an introduction into the fields of shadow generation and real-time rendering is given. First of all, it is explained why shadow generation in computer graphics application is needed at all (Section 2.1). In Section 2.2, the terms used in this thesis are defined and explained:

- What are the reasons for shadows to be cast, and what are their characteristics (2.2.1)?
- What is real-time rendering, what are interactive frame rates (2.2.2)?
- How are images rendered at all? Which techniques can be used (2.2.3)?

Finally, some real-time shadow generation algorithms are presented and explained in Section 2.3, with focus on the two mostly used: shadow mapping and shadow volumes.

2.1 Why shadows?

Shadows play an important part in the visual perception of the world, since they appear in nearly every situation or scene of our daily life. They are in fact so common that we usually notice them only unconsciously. Most of the time we only realize the impact of shadows in a scene when they are missing: Images created by artists, on which no Shadows are visible, do usually seem unnatural and artificial, no matter whether they are hand-drawn or created digitally with computer graphics methods.

The reason for this is that the absence of shadows reduces the usual information a human perceives when looking at a scene: Shadows provide visual cues which help the human brain to understand the geometry of the objects. Whenever they are missing, it is harder or even impossible for the human visual system to determine the correct three dimensional relationships:

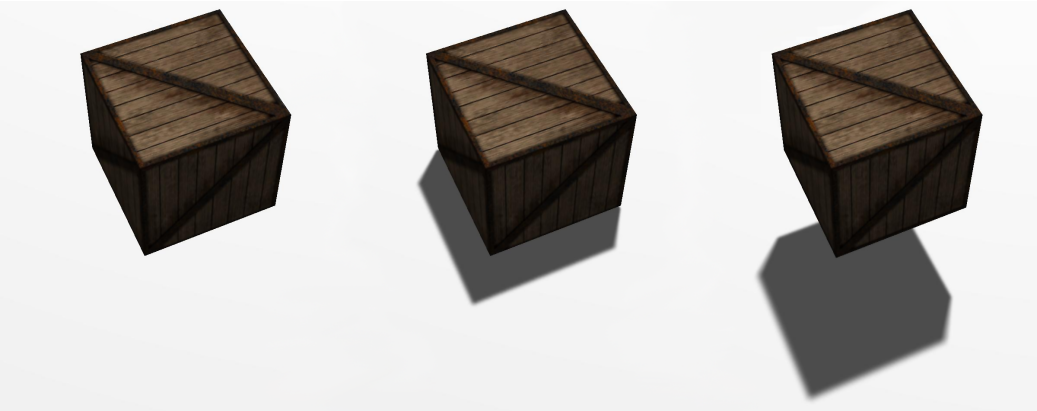

Fig. 2.1: Shadows provide information on the relative position of an object in space. On the left image, the crate's position can't be determined. In the middle and right images, this is different due to the cast shadows.

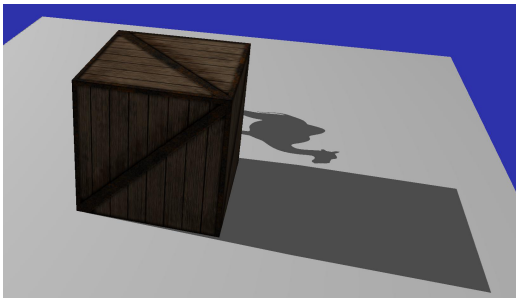

Fig. 2.2: The camel is completely hidden behind the crate, but the shadow reveals its presence.

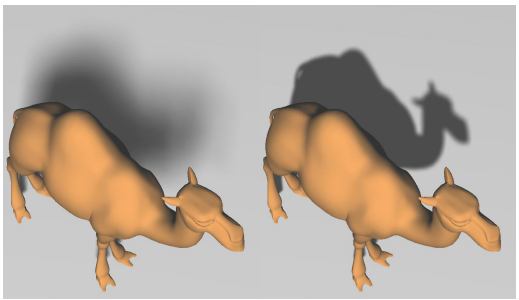

Fig. 2.3: Shadows provide information on the light source size: The larger the source, the softer the shadow.

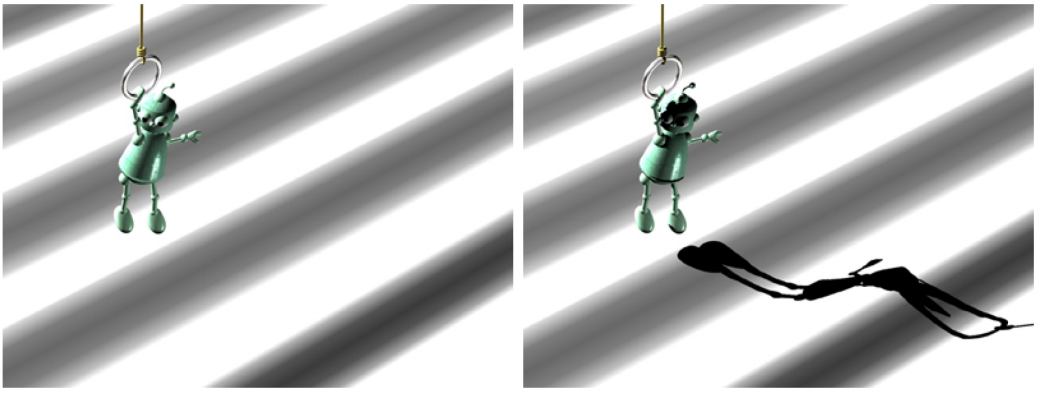

Fig. 2.4: The geometry of the receiver can be estimated by the cast shadow [HLHS03].

- Shadows give information on relative position and size of scene objects - they "anchor" objects in a scene. They tell you if the object touches the shadow receiver or (otherwise) the distance to it. Without shadows, an object just "floats" (Figure 2.1).
- Through the projection of the shadow caster onto the shadow receiver, shadows provide geometrical information on object parts of the occluder which are not visible from the observers eye position (Figure 2.2).
- Shadows also give you details about the light source which is responsible for their occurrence, like for example its extents or its distance (Figure 2.3).
- Depending on their projected shape, shadows also reveal much of the geometry of the shadow receivers. Complex structures of certain surfaces only become visible when a shadow is cast on them (Figure 2.4).

Simulating shadows in Computer Graphics applications can therefore help the perceiver to gain a better sense of the three dimensional scene, increases the realism of the generated images and adds a certain "atmospheric" effect to them.

2.2 Basics and definitions

In this section, the basics and definitions which are necessary to understand this thesis are given. First, the term *shadow* itself is introduced, followed by a definition of *real-time/interactive* frame rates and the explanation of current *rendering methods*.

2.2.1 What are shadows?

A point lies in shadow whenever it is not visible from the light source's position (and vice versa). This definition holds of course only for point light sources, which have no extents and exist only theoretically. They produce so-called *hard shadows*: Every single point in a scene is either in shadow or not, which is leading to a high contrast at the shadow borders, making them very noticeable for the human eye. This leads to the following problems:

- Artifacts at the borders are extremely noticeable as well.
- Hard shadows can in some occasions be interpreted as separate objects.

So-called directional light sources (light sources that are infinitely far away) emit parallel light rays and produce hard shadows as well. In computer graphics, the sun light is often approximated and modeled as a directional light source.

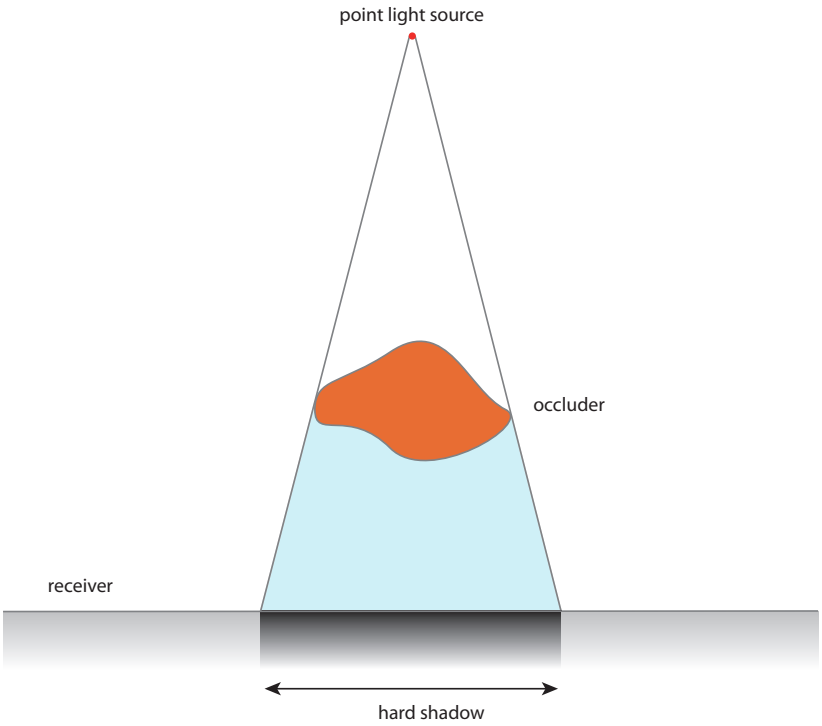

Fig. 2.5: Hard shadow cast by a point light source: An occluder blocks the light from the receiver.

Apart from the light source, every shadowed scene consists of several objects, which is illustrated in Figure 2.5: The object causing the shadow by blocking the light is called shadow caster, shadow creator, occluder or blocker. The object shadowed by this occluder is called the shadow receiver or occludee. A special case called *self-shadowing* occurs whenever an occluder casts a shadow on itself due to its special geometry. The object is then shadow caster and shadow receiver at the same time (see Figure 2.6).

In many computer graphics applications, especially when real-time frame rates have to be achieved, lights are modeled as point or directional sources due to the fact that calculating a hard shadow is comparably easy and fast. In the "real world", every light source has its own area (or even volume), leading to so called *soft shadows*, which are discussed in detail in Chapter 3.

2.2.2 Real-time vs. interactive frame rates

Shadow calculation in a computer graphics application is a global effect and computationally expensive. Depending on the rendering method, the used shadow

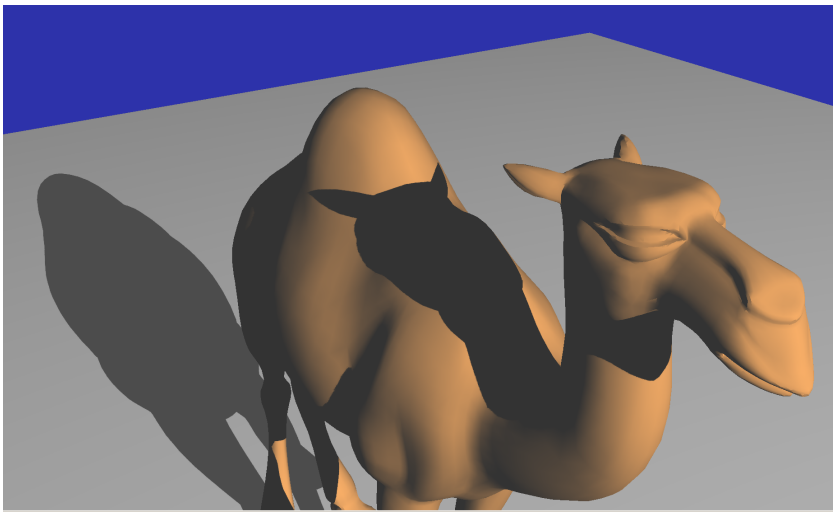

Fig. 2.6: An example for self-shadowing: The object casts shadows on itself.

algorithm, the available hardware, the quality of the shadow, the scene complexity and the size of the rendered frame, the generation and rendering of a shadowed image can be a very time-consuming task: It can take from several milliseconds up to several minutes or even hours.

In this thesis, the focus lies on methods and algorithms which allow shadows to be rendered and used in *real-time* applications. Real-time means, that at least 30 frames per second (FPS) can be calculated and presented on the screen (or twice the rate for stereo vision). It is important that the frame rate does not drop below this value and is as constant as possible, because the human visual system is quite sensitive to substantial frame-to-frame changes. Frame rates higher than 30 Hertz are perceived as continuous animations. An application which manages to output "only" 2 to 15 FPS renders at so-called *interactive* speed.

Today's hardware is capable of rendering various shadow techniques in real-time. Especially programmable graphics processor units (GPUs), which speed up *polygon rasterization* rendering methods (see Section 2.2.3) enormously and are by now available in many home computer systems and gaming consoles, make it easy for developers to include real-time shadows with increasing quality in their applications and games. Since no shadow algorithm can fulfill the requirement to render *any* arbitrary scene with guaranteed real-time frame rates, though, it is necessary for an algorithm to be parameterizable to perform as needed. Such an altering of a parameter in order to gain more speed often comes at the cost of visual quality.

Although extremely realistic shadows can also be rendered with various techniques like *ray tracing*, *photon mapping* or *radiosity*, interactive or even real-time

frame rates cannot be achieved with them today with consumer hardware. This is especially true in the case of soft shadows. Therefore, algorithms for these techniques are not handled in detail in this thesis - the focus lies on *polygon rasterization* methods. An in-depth overview on real-time rendering can be found in [AMHH08].

2.2.3 Rendering methods

In computer graphics, rendering is the process of calculating an image out of mathematically described scene objects by a computer program. Such an object description can for example consist of three dimensional vertices, which are connected by edges, which in turn form a triangulated wireframe model. Another possibility is to describe them by mathematical functions. Furthermore, a view point, from which the scene is looked at, is specified, and the light sources are set. The challenge in rendering is now to create color information (a digital image) out of the given mathematical definitions. This problem can be solved by several so-called *rendering methods*.

Each of the approaches has of course its own advantages and disadvantages, but they are all capable of rendering shadowed images. In order to understand the shadow algorithms which are presented later, it is necessary to know some details about the most important underlying rendering methods (polygon rasterization, ray tracing and radiosity).

Polygon rasterization

The most common way to render three dimensional objects in an application or game is to use *polygon rasterization*. The methods works by transforming the polygons the objects are made of into screen space and color their corresponding pixels there ("rasterization"). The calculation of a pixel color is done by using approximate illumination models and the polygon's surface properties.

A big advantage is the fact that areas where no polygons appear can easily be determined and ignored in the rendering process. Problems arise whenever two polygons lie behind each other: Only the fragment of the polygon with the smallest distance to the eye should be drawn, otherwise the visibility of the scene would be wrong. *Fragment* is the term used for a pixel with additional data like the depth or a normal.

This difficulty is solved by using a so-called z-buffer, which is a buffer with exactly the same dimensions as the render window and saves the z-value (the distance from the eye) for each drawn fragment. Whenever another polygon part is transformed to this screen space pixel location, its depth is compared to the

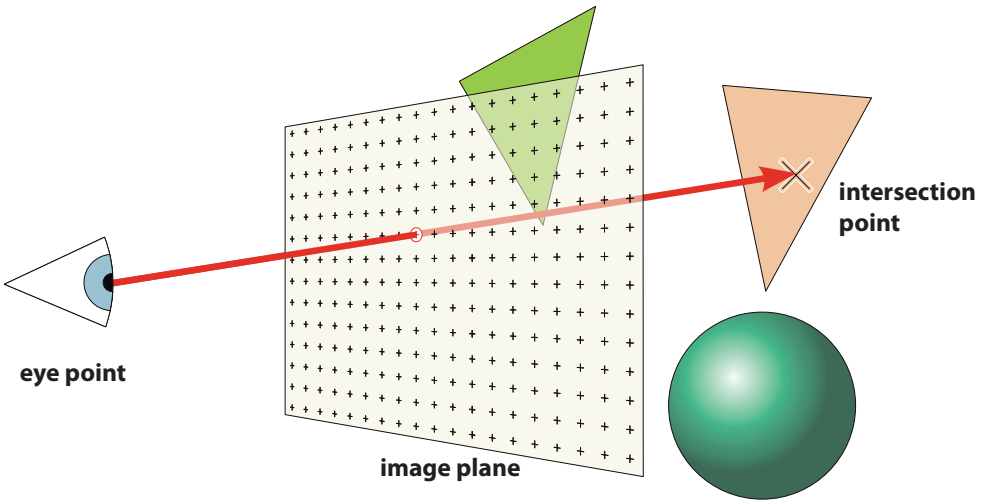

Fig. 2.7: Ray tracing basics: A ray is shot from the eye point into the scene. The color of the object at the intersection point is projected onto the corresponding pixel in the image plane [ray07].

value in the buffer: If the depth is smaller, it gets drawn and the new depth value is stored - otherwise, the fragment is ignored.

The introduction and the following huge success of specialized graphics card around 1995 were responsibly for a drastic improvement of this method in terms of speed and visual quality, and graphics acceleration hardware soon became a standard in today's home PCs and consoles. In 2000, it became even possible to program the whole rendering pipeline on the GPUs using small programs called *shaders*. They allow developers to control every single step during the transformation and rasterization and have been responsible for the invention and improvement of many new algorithms - especially in the area of real-time shadow generation.

Ray tracing

Ray tracing is an algorithm which uses a data structure called ray to generate images out of the scene description. For each pixel, a ray is shot from the eye point into the scene, intersecting the objects there. The intersection point with the smallest distance to the eye is used to determine the pixel color of the rendered image (see Figure 2.7).

For this calculation of the pixel color, the material properties and the point normal as well as the *bounced* rays are used to calculate the shading: In contrast to *Ray Casting*, the rays are traced further recursively, which simulates the physical

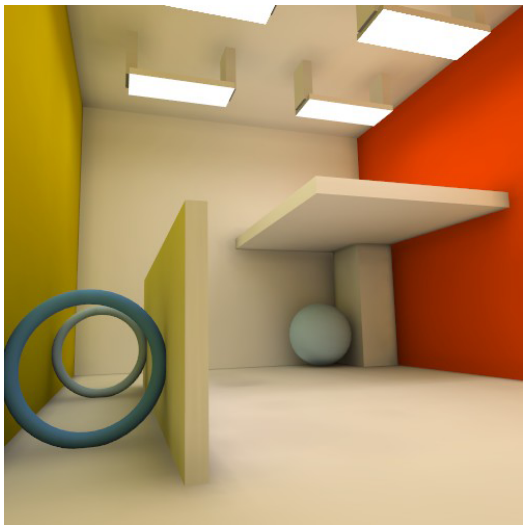

Fig. 2.8: A test scene rendered with the radiosity technique [rad05].

effects of reflection and refraction. The more bounces a ray is allowed to make, the more realistic the result is.

The simulation of shadows in ray casting is straight-forward: A point on an object lies in shadow if an occluder lies between it and the light source. This occluder-test can simply be performed by using a ray again, which is shot at the object point from the light source. Whenever it is intersected by another object, the point is in shadow, otherwise it is lit.

Even though ray tracing can generate very realistic images out of all intersectable and mathematically described scene objects, its performance is quite bad compared to polygon rasterization. This is due to the fact that it is a global rendering method, which means that all scene objects have an influence on every single scene point. This makes it impossible to be used in real time on today's graphics hardware, but recent developments have shown that real-time ray tracing applications could soon become reality.

Radiosity

The *radiosity* algorithm only works on ideal diffuse object surfaces, which are divided in a finite number of small patches. Each of these patches act as indirect light source whenever they are illuminated directly, since they reflect all incoming light they cannot absorb in all directions. First, it is necessary to compute the energy-radiant interactions between all those patches. This is done by using so-called form factors or view factors, which describe how much the patches influence each other (for example, large distances, oblique angles or blocking objects

between two patches will reduce the form factor value).

For each patch, a linear equation system is set up using the corresponding form factors, which is solved numerically. The result is the *radiosity* (or brightness) for each patch. Soft shadows are automatically computed using this approach. The algorithm can be expanded to other materials, but it is limited in precision by the amount of patches and the resulting memory requirements.

Once the radiosity values are computed and stored, the scene can be rendered in real-time, as the lighting is independent of the viewing direction and does not change during camera movement. This method is therefore a good solution for the rendering of static scenes in real-time, if the preprocessing time does not matter. See Figure 2.8 for an example image rendered with radiosity.

2.3 Shadowing in real-time

Shadowing in real-time is a non-trivial challenge, and there is no general solution which can be applied on any arbitrary scene. Sometimes it is sufficient to calculate static shadow information in a preprocessing step and store it in a special data structure or texture, which can later be used to illuminate the corresponding objects in real time. But as soon as dynamic elements are involved, other techniques have to be chosen.

Many older games use (blurry) dark textures, placed underneath the objects, as approximate shadows (see Figure 2.10). Even though this technique is of course no real shadow calculation but only a very rough estimation, it is quite cheap in terms of computation, and it enhances the visual perception of the three dimensional scene. Another method called *projection shadows* [Bli88] uses a matrix transformation to project the shadow caster object flattened onto the ground plane. The scene is first drawn without shadows, followed by a second pass on which the transformed version of the occluder is drawn onto the plane as a shadow. Even though it is possible to calculate correct shadows with this approach, it can't handle self-shadowing and is only applicable for scenes with very few, large objects. An example is given in Figure 2.9.

The two most robust shadowing algorithms are presented in the following sections: *Shadow mapping*, an image based approach, and *shadow volumes*, an object based approach, can both handle self-shadowing and are able to solve the visibility problem caused by the shadow calculation in real-time. For more information on real-time shadows, we recommend [AMHH08] and [HLHS03].

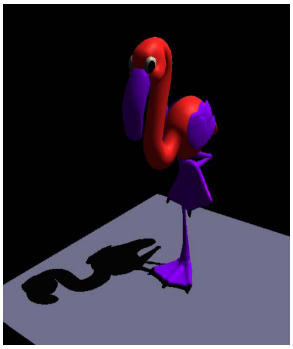

Fig. 2.9: Projective shadows as proposed in [Bli88].

Fig. 2.10: Approximate shadows: Simple geometry is used to "fake" a shadow (Screenshot from the game *NFS V: Porsche*, ©Electronic Arts).

2.3.1 The shadow mapping algorithm

Shadow mapping is an image based algorithm which was first introduced by [Wil78] in 1978. The basic idea of this method is to view the scene from the position of the light source in a first pass, and store the depth values of the fragments in a texture (called the *shadow map*). The shadow map therefore contains the distances to all sampled surface points which are illuminated by the light source. Depending on the type of used light source, a perspective (for point lights) or an orthographic (for directional lights) projection has to be used. The shadow map has to be updated whenever an object in the scene or the light source moves.

In the second pass, the scene is rendered from the camera's point of view. Every fragment is transformed into light space, where its distance to the light source is compared to the corresponding value in the shadow map. If the distance to the current fragment is larger than the shadow map value, it lies in shadow; otherwise it has to be illuminated by this light source. Figure 2.11 illustrates the basics of the algorithm.

The shadow mapping algorithm can be implemented in a fully hardware accelerated way on modern GPUs and is therefore a comparatively fast and often used method to simulate hard shadows in real-time applications and games. Furthermore, it can handle any geometry due to the fact that it is an image based approach, its speed is independent of the scene and object complexity, and self-shadowing is handled automatically.

Shadow mapping has some considerable drawbacks, though: It is not possible to use only one shadow map when using an omni-directional light source, as no view frustum can represent such a spherical view. Moreover, due to the sampling of the scene during the creation of the shadow maps, aliasing and undersampling artifacts are likely to occur:

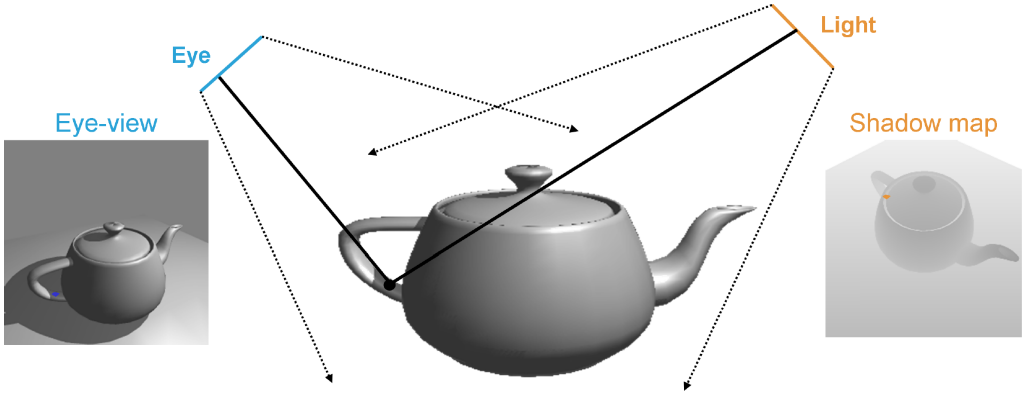

Fig. 2.11: The shadow mapping algorithm: The depth values as seen from the light source are stored in a shadow map, and are then used in a second pass to generate shadows on the objects [Sch05].

- Perspective aliasing occurs whenever the shadow map resolution is insufficient for regions near the camera due to the fact that a perspective view shows nearby objects larger than distant ones (see Figure 2.12). This problem can for example be reduced by using perspective shadow maps [SD02] or light space perspective shadow maps [WSP04], where the shadow map resolution near the camera is increased.

- Projection aliasing can be found on surfaces which are nearly orthogonal to the shadow map plane (so the surfaces lie nearly parallel to the light direction). It is possible that only very few shadow map texels are projected onto such an area, leading to an insufficient resolution there (see Figure 2.13). By using the standard openGL shading model, such artifacts can be nearly completely hidden.

- The limited precision of the depth values in the shadow map or resampling errors can lead to incorrect self-shadowing or *shadow-acne* (see Figure 2.14). This can be avoided by using a depth bias, which is added to the depth value in the shadow map before the shadow comparison takes place.

Sampling artifacts can also be reduced using a technique called percentage closer filtering (PCF) introduced by [RSC87] which softens the shadow borders (see Section 4.3). This filtering method is meanwhile included in current graphics hardware and can be used without performance hits. Soft shadows generally help to hide the artifacts which occur in shadow mapping.

More details on solving the problems in shadow mapping can be found in [Sch05].

Fig. 2.12: Perspective aliasing artifacts

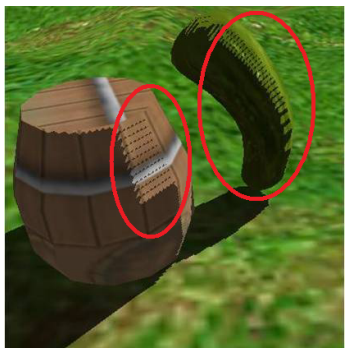

Fig. 2.13: Projection aliasing artifacts

Fig. 2.14: Shadow acne artifacts

2.3.2 The shadow volume algorithm

The *shadow volume* algorithm, first introduced by [Cro77], is a completely different approach, as it is an object based method. It works by extruding the object silhouettes as seen by the light source to infinity - the light and the occluders "cast out a shadow volume", which is used for the shadow query. The silhouette extraction has usually been done on the CPU, but with the introduction of so-called *geometry shaders* on newer GPUs, it has become possible to calculate the volumes on the graphics hardware as well.

The shadow test is performed by using a counter: For each pixel, the volume intersections which occur from the eye point to the object point are investigated. Every time a volume is entered, the counter is increased by one, and every time a volume is left, the counter is decreased by one (see Figure 2.15). Whenever the object point is reached and the counter is larger than zero, it lies in a shadow volume, otherwise, it is illuminated. This can be implemented by using a stencil buffer:

1. Reset stencil, draw the scene with ambient lighting only, fill z-Buffer.

2. Draw the shadow volume twice with z-test on:

   - The first time, render the front faces only & increment the stencil buffer.
   - The second time, render the back faces only & decrement the stencil buffer.

3. Draw the scene with stencil test on, update & illuminate only pixels whose stencil value is zero.

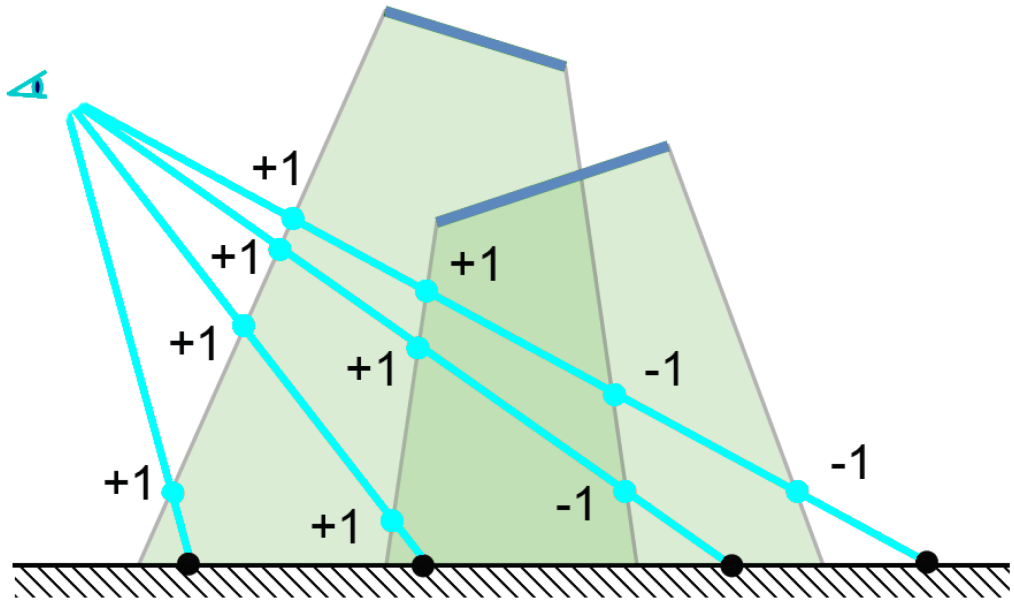

Fig. 2.15: The shadow volumes algorithm: By entering a shadow volume, the counter is increased, and by leaving it, the counter is decreased [Sch05].

The big advantage of the shadow volume algorithm is its object precision: the shadow volumes represent the exact shadow, and not a sampled version as in shadow mapping, so no aliasing artifacts can appear. Moreover, it supports omni-directional point light sources and handles self-shadowing. A famous example for a game which has shadow volumes implemented is *Doom 3* (see Figure 2.16).

The drawbacks of the shadow volume algorithm appear immediately when it comes to the rendering of polygon-rich scenes: A huge fill-rate is necessary to render all the volumes, and the silhouette extraction is another costly factor. Another disadvantage is the limitation to polygonal objects, as the determination of the silhouettes has to be fast.

2.4 Summary

This chapter gave an introduction to the area of shadowing in computer graphics, and described how the visual quality and realism of artificially generated images can be improved by it. The terms necessary to understand the basics of shadowing and real-time rendering were defined, followed by an explanation of the most important rendering methods and real-time shadowing algorithms.

More information on the topics real-time rendering and shadows in computer graphics can be found in Möller and Haines' *Real-Time Rendering* book [AMHH08].

Fig. 2.16: Screenshot from the game *Doom 3* (©id Software), which uses the shadow volumes algorithm.

Soft and quick as shadows, we must be.

(*Gollum in "The Lord of the Rings"*)

Chapter 3

Soft shadows

3.1 What are soft shadows?

As already explained in Section 2.2.1, light sources in real-time applications are often modeled as point lights with no physical extent. Such point light sources only exist in theory, though: In the "real world", light sources consist of an area or even a volume, causing so-called *soft shadows* to be cast. The shadow definition given in Section 2.2.1 does not hold for them anymore, since we are not dealing with a point-to-point, but an area-from-point visibility problem in such a case. The calculation of soft shadows is therefore far more complex, but due to the recent graphics hardware developments and the invention of new algorithms, it has become possible to render them in real-time. Still, we are far away from rendering realistic and physically correct (!) soft shadows in any arbitrary scene, and a lot of research effort is put into this topic, which can also be seen by the large number of recently published papers (see Chapter 4 for an overview).

In Section 3.1.1, an explanation on how and why area light sources cause soft shadows as well as the definition of the terms *umbra* and *penumbra* are given. Section 3.1.2 compares soft to hard shadows and lists their assets and drawbacks. Special scene configurations in connection with soft shadows are treated in Sections 3.1.3 and 3.1.4.

An excellent survey on soft shadows can also be found in [HLHS03].

3.1.1 Umbra & penumbra

In case of a point light source (and the resulting hard shadow), shadowing is a binary decision: The point is illuminated whenever the source is visible from its position, and otherwise not. For area light sources, things are more complicated: If the light source is partly visible from the position of the affected point (which means that this point is only partly hidden from the light source), it is neither completely lit, nor completely shadowed, but in the so-called *penumbra*. The penumbra is the region in which the fading from the unshadowed area to the completely

shadowed area (the so-called *umbra*) takes place (see Figure 3.1).

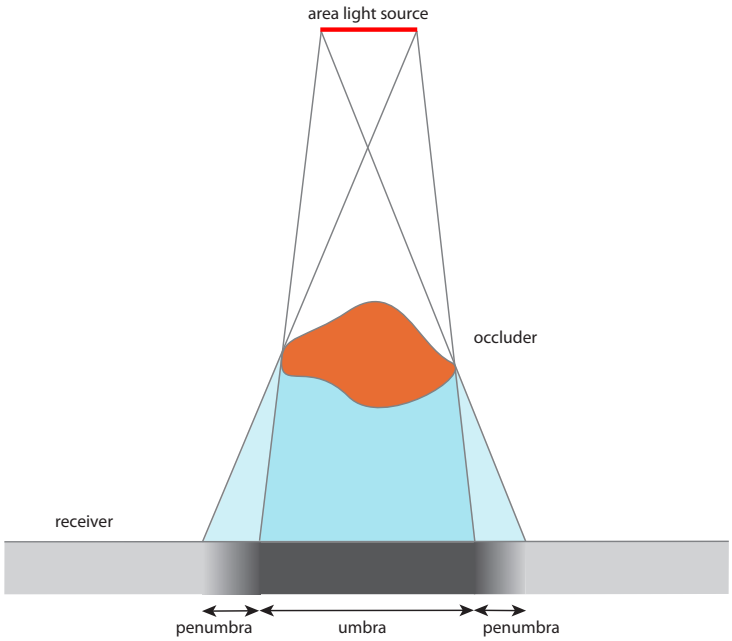

Fig. 3.1: An area light source leads to a soft shadow, which consists of umbra and penumbra.

So, in case of an area light source, the shadow consists of the union of umbra and penumbra, and a point is shadowed, whenever some area of the light source is not visible from its position. Apart from that, the terms already defined in 2.2.1 are the same.

The exact calculation of umbra and penumbra is a quite complex task as it implicates solving a three-dimensional visibility problem. Their extents depend on the light source area size, on the distance from occluder to receiver and on the distance from light source to occluder. For example, the umbra is not just the hard shadow region which would be generated by a point light - the larger the light source is in respect to the occluder, the smaller the umbra region is.

Since the umbra is defined as the area where the light source is not visible at all, it would have to be completely black in theory. In practice, the umbra is of course illuminated due to light reflections on scene surfaces. This behavior is usually simulated by using an ambient term which contributes to the final object color no matter if it is in shadow or not.

3.1.2 Hard shadows vs. soft shadows

The decision on whether to implement hard or soft shadows in a real-time rendering application relies heavily on its demands:

- Soft shadow algorithms are a lot more difficult to understand and evaluate due to the increased complexity of the visibility problem. This can be seen by the fact that nearly all real-time soft shadow algorithms are enhanced versions of basic hard shadow methods, namely shadow mapping and shadow volumes.
- As a result, the rendering time for soft shadows is usually significantly longer, and varying frame rates are more likely to occur. On the opposite side, robust hard shadows can easily be implemented in real-time applications.
- Concerning the visual image quality though, soft shadows provide obviously far more realistic and convincing results, since they are caused by light areas with finite extent. Observers are used to such shadows from the "real world". Even the sun cast shadows that consist of a penumbra and an umbra due to its significant angular extent, whereas light points and hard shadows only exist in theory and are often perceived as unrealistic in images. See Figure 3.2 for a comparison.
- Soft shadows are inherently blended into the scene because of the gradual shifting of the penumbra color from light to dark, avoiding them to be perceived as separate objects. Moreover, this reduces visible artifacts and aliasing problems that are likely to occur in hard shadow approaches drastically.

In conclusion it is to say that as long as there are enough resources available in a computer graphics application, the implementation of soft shadows is to be preferred, as they greatly improve the visual quality and the realism of a rendered image. Light sources should only be modeled as point lights when resolution of the framebuffer is too low to properly render the penumbra. This can be the case whenever the distance from light source to occluder is significantly larger than the distance from occluder to receiver.

3.1.3 Shadows from several light sources

Soft shadows for more than one area light source can be rendered quite easily: Once we know how to calculate a soft shadow for a single area single source, we can do the same for all area sources in a scene and sum up their contribution (for each wavelength or color band; see Figure 3.3 for examples).

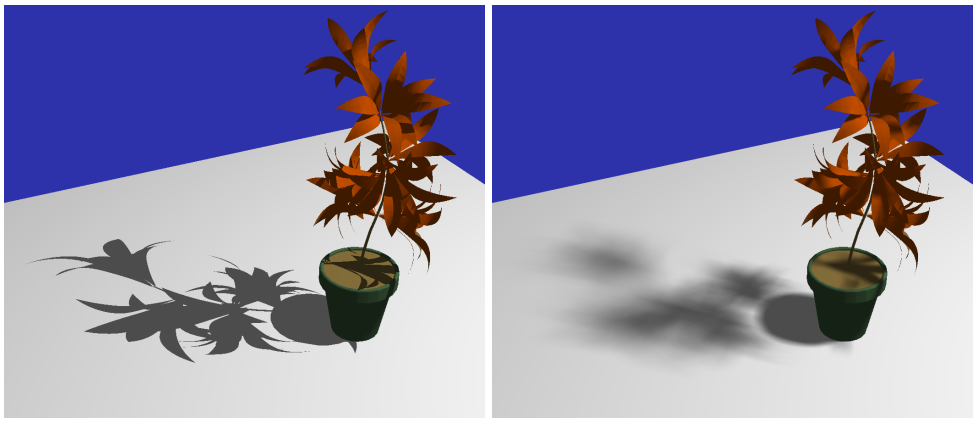

Fig. 3.2: Soft shadows look much more realistic than hard shadows.

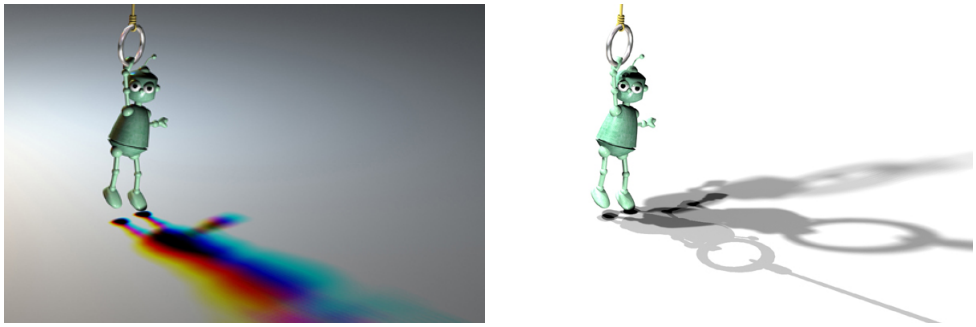

Fig. 3.3: Soft shadows from several light sources [HLHS03].

3.1.4 Shadows from several objects

In contrast to the calculation of combined shadows from several occluders for a point light, the evaluation for an area light source can be quite complicated. Whereas the union of the individual hard shadows is equivalent to the overall shadowed area, this does not have to be true for soft shadows: For example, a point where the light source is not completely blocked by the occluders taken separately can still be completely occluded by the union of these shadow casters. The umbra region of the whole set is therefore larger than the union of all separate umbra regions. Such a configuration is displayed in Figure 3.4.

A big problem for soft shadow algorithms is therefore the fact that the individual partial visibility functions can't be combined to a partial visibility function of the whole set of occluders, making its prediction a very hard task, particularly in real-time. Hence it is approximated or bounded in most approaches, especially in single sample methods (see Section 3.2). Especially in animated scenes, this approximation works quite well, and even though not all artifacts can be removed,

they are usually not very noticeable.

In Chapter 5, a new real-time algorithm is presented which makes it possible to render physically correct soft shadows, including the capability to solve the visibility problem associated with several occluders.

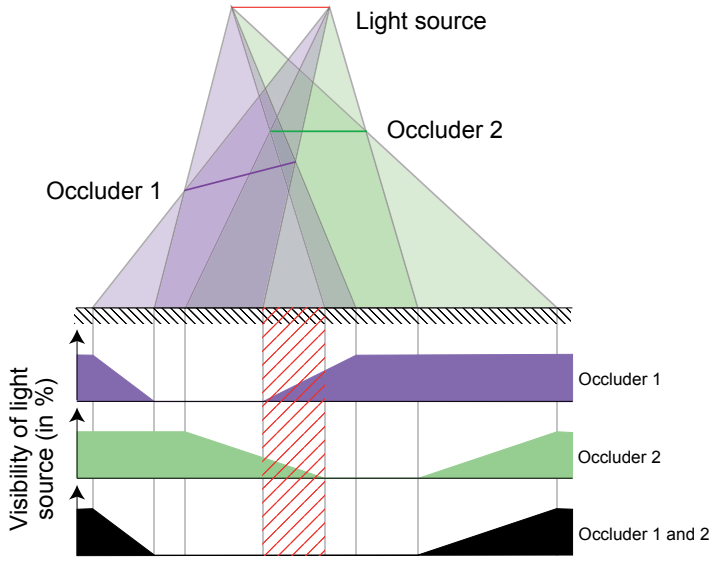

Fig. 3.4: Soft shadows from several overlapping occluders. Notice that the indicated area in the middle lies in the umbra, although it is not completely blocked by a single occluder [HLHS03].

3.2 Physically correct vs. fake soft shadows

When calculating soft shadows cast by an area light source, we are dealing with a difficult visibility problem: Due to the extents of the source, it is not sufficient to identify the shadow caster from a single point. Theoretically, all parts of the occluder which are visible from at least one point of the light source have to be identified in order to guarantee a physically exact shadow to be cast. This is of course hard to achieve in real-time.

For this reason, most real-time soft shadow approaches estimate the visibility by calculating it from only one point of the area light source, and simulate the effects which are typical for soft shadows using approximative heuristics. These algorithms produce so-called fake soft shadows, which often look quite realistic, but cannot guarantee physical correctness. In fact, they are enhancements of hard shadow techniques combined with cleverly used border softening methods. Still, they can be absolutely sufficient for many purposes - for example, in animated

scenes (which occur in most games), it is often nearly impossible to perceive any differences between physically exact or fake soft shadows.

3.2.1 Problems with fake soft shadows

Fake soft shadows may work well in many cases, but in some situations, the differences to an exact soft shadow solution become visible. Especially when the light source area is quite large in relation to the shadow caster, the approximation does not deliver satisfying results. For an example, see Figure 3.5: The large light source is very close to the blocker, illuminating different sides of the object and leading to a very small umbra region. Fake approaches do usually have difficulties with this scene configuration.

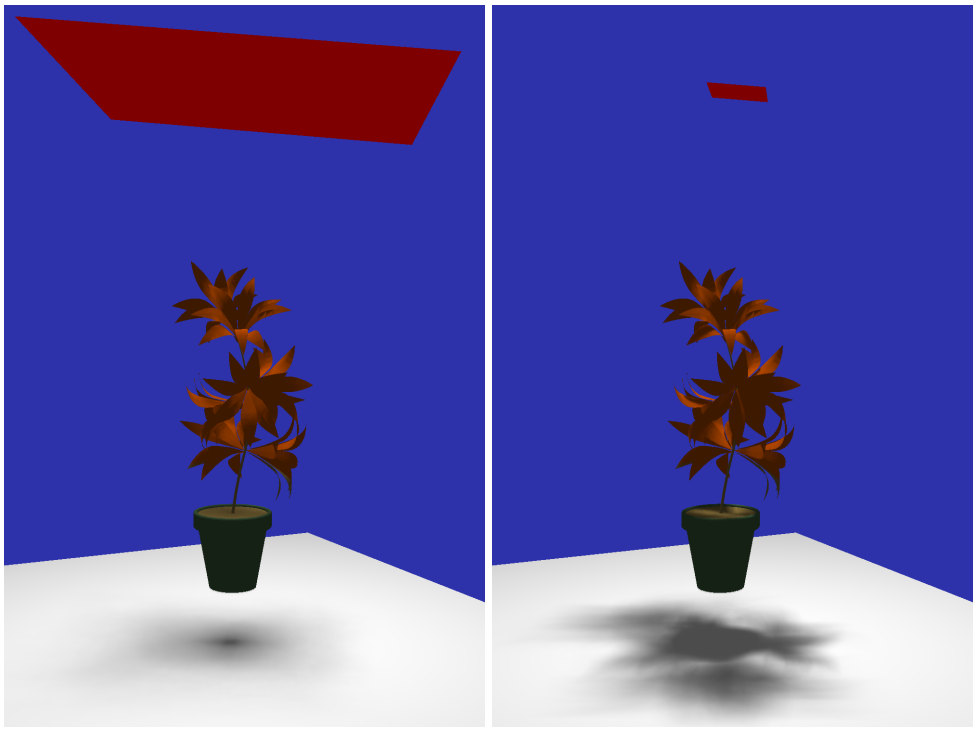

Fig. 3.5: If the light source is close and significantly larger than the receiver, the generated shadow differs extremely from a shadow generated by a single sample fake shadow approach - the umbra nearly disappears.

Even though such a scene configuration can usually be avoided in real-time applications and games by the use of smaller area lights and clever positioning of the objects, the same problems arise whenever an elongated objects lies along the axis of the light source (one end of the object points to the light source, the other

end is close to the receiver). The shadow quality can in such a case be improved by the following solutions:

- The visibility calculation can be performed using the whole extent of the light source. This is usually too complicated to be performed in real-time.
- The area light source is split into several smaller areas, for which the visibility gets estimated separately. This is geometrically closer to the previous approach and reduces some artifacts. Of course, the calculation time compared to a single sample approach is multiplied by the number of separate light sources.
- Another possibility is to "cut" the object into slices and compute soft shadows for each slice separately. The combination of these individual shadows is usually quite difficult, though. The calculation time of the single sample approach is multiplied by the number of slices plus the time needed to combine the shadows correctly.

3.2.2 Ways to approximate the penumbra region

As already explained before, fake soft shadow approaches compute the visibility using extended hard shadow algorithms which approximate the penumbra region. This can be done in several ways:

- The penumbra is computed by extending the umbra outwards. This leads to shadows which always consist of an umbra and a penumbra, which can be false in some scene configurations and significantly darkens the scene (see Section 3.2.1).
- The opposite way is to compute the penumbra by shrinking the umbra region. This can lead to light leaks between neighboring objects when their shadows overlap and influence each other.
- Of course, soft shadows can also be approximated by calculating both inner and outer penumbra.

3.2.3 Shadows from several objects

In Section 3.1.4, the difficulties involved in calculating the soft shadows from several occluders was explained: The umbra caused by the union of the individual shadows can be smaller than the umbra caused by the shadow of the whole set of the occluders. The exact computation is a hard task, as it requires the visibility to be known from every single point of the light source.

Since fake soft shadow algorithms cannot estimate the visibility from all the light source points due to the fact that most of them use only a single point to compute the visible surfaces, they cannot generate physically exact soft shadows from several objects and produce artifacts like in Figure 3.6. Even though these effects are clearly visible on images, they are hardly noticeable in dynamic scenes. Especially in games, clever object placement and animation can hide most of these problems, making soft shadows preferable over a hard shadow solution.

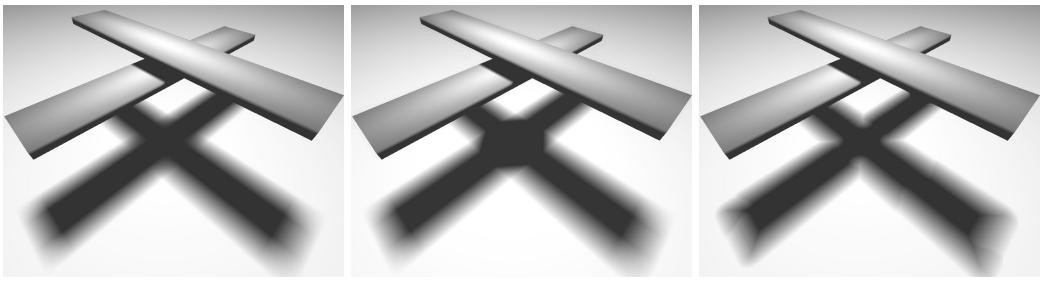

Fig. 3.6: Soft shadows generated by overlapping occluders, created with different soft shadow algorithms. Left: Reference image. Middle: Penumbra wedges (Section 4.12). Right: Flood fill (Section 4.5). Notice the artifacts which occur by using these single sample fake approaches (Images from [GBP06]).

3.3 Summary

In this chapter we have explained where soft shadows come from and why they improve the quality of rendered images: Since hard shadows are only cast by theoretical point light sources, they do not occur in reality. Therefore we are used to soft shadows, which are caused by area light sources with a geometrical extent, and consist of umbra and penumbra regions.

Generating physically exact soft shadows is a very complicated task, since the visibility computation is more complex than for point lights. In order to achieve real-time frame rates in games and applications, it is therefore usual to use fake approaches to render visually plausible shadows. Of course, these simplified visibility calculations do not provide exact solutions and do generate artifacts, but in many cases (especially in dynamic scenes), the results are sufficient to convince the casual observer.

- “Dark the other side is... very dark...”
- “Shut up and eat your toast, Yoda!”

(*Unreleased dialogue from “Star Wars”)*

Chapter 4

Real-Time Soft Shadow Algorithms

In this chapter, the most widely used real-time soft shadow algorithms are presented: All of them are either based on shadow mapping as presented in Section 2.3.1 or on the shadow volume algorithm (see Section 2.3.2). Some approaches do even combine ideas from both methods. The main idea of all of them is to enhance these standard hard shadow approaches by a penumbra.

An excellent survey on real-time soft shadows can be found in [HLHS03], but since it was published in 2003, it does not cover the latest algorithms and methods like for example backprojection techniques (Section 4.8).

4.1 Multiple shadow maps per light

The probably most intuitive approach to generate soft shadows is to generate hard shadows from several sampling points on the area light source and accumulate this information. Theoretically, if an infinite number of samples could be taken, this idea would lead to the exact, physically correct soft shadow solution. Since this is not possible, [HH97] suggests using only a few regularly distributed samples for the calculation in order to simulate the penumbra.

This method can be implemented in an image based version, as described by [Her97]. For each shadow receiver, a so-called *attenuation map* is computed:

- The light source is sampled N times, hence N binary occlusion maps are generated for each shadow receiver. Such an occlusion map is generated by rendering the scene from the light view and contains a 0 for the receiver and a 1 for every other object.

- All the binary occlusion maps are then combined into the attenuation map. Each of its pixels contains the number of times the receiver was occluded after N light source samples (See Figure 4.1).

- Since it is necessary to calculate such an attenuation map for the whole set of receivers P, it takes $P * N$ passes to calculate all of them.

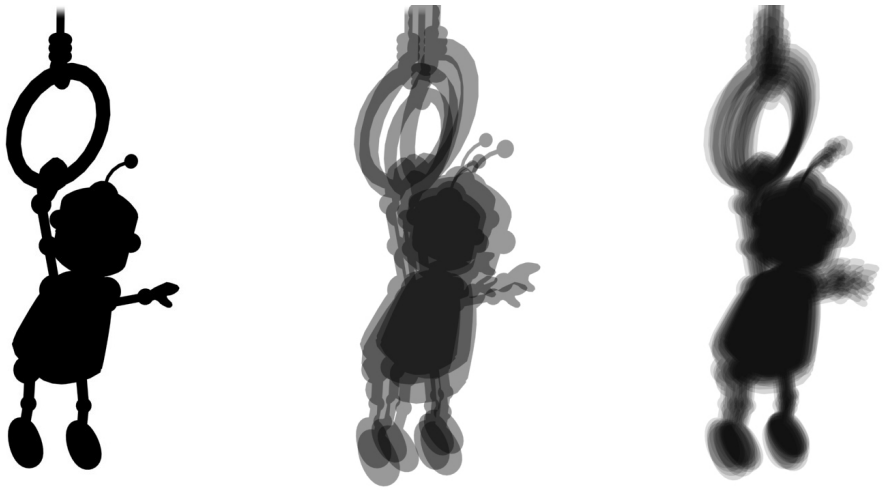

Fig. 4.1: Several occlusion maps are combined to generate an occlusion map. Left: a single occlusion map. Middle: attenuation map with 4 samples. Right: attenuation map with 64 samples [HLHS03].

- The attenuation map is then used in a final rendering pass to modify the illumination of the scene objects.

Even though the idea is simple and not too complicated to implement, the problems which can occur in this approach are obvious: Depending on the size of the penumbra, enormous amounts of binary occlusion maps have to be calculated, which is usually computationally too expensive to be performed in real-time. Of course, the algorithm can easily be fastened by using fewer samples, but this drastically reduces the soft shadow quality: For example, by taking less than 9 samples in a simple scene with a moderately large light source, our visual system does not perceive a soft shadow anymore, but several hard shadows. For large area light sources, it can be necessary to calculate more than 256 binary occlusions maps in order to gain high quality soft shadows. Other ways to speed up the rendering process are to limit the scene to a single shadow receiver or to parallelize the computation by using several computers/CPUs/GPUs.

Despite its speed problems, the algorithm has an important advantage: It generates physically correct soft shadows (under the assumption that enough light source samples are taken), which is not the case for the other real-time algorithms. Of course, also the shadow volumes algorithm can be used to generate such an attenuation map. The basic idea of this method has also been used in the new algorithm we present in Chapter 5.

An enhancement of this algorithm was presented in [ARHM00]: Instead of calculating and using an attenuation map for each receiver, a single *layered atten-*

uation map for the whole scene is created:

- Like in the basic method, the light source is sampled, and different scene views from these sample points are generated.
- These generated images are then transformed to a central reference view, which lies at the center of the light source.
- By using the z-Values of the images, the object with the smallest distance to the light source can be estimated and makes the first layer.
- The number of samples seeing this object is counted and stored in the layered attenuation map together with the distance, giving an estimate for the occlusion percentage.
- The same is done for any other visible objects, forming further layers.

In the final rendering pass, the layered attenuation map is used like a standard attenuation map, with the difference that it is used for the whole scene. When the scene is rendered from the camera view point, a look-up in the layered attenuation map takes place for every screen pixel: If it is in the attenuation map, the occlusion percentage is used to modify the illumination. Otherwise, the pixel is completely occluded and stays dark.

Using this approach still requires the area light source to be sampled N times - but for the whole scene, and not for every single shadow receiver! Furthermore, it is not necessary to differentiate between shadow casters and receivers, and self-shadowing is handled automatically. Of course, speed and quality still depend heavily on the number of samples taken. An enhanced version of this algorithm can be found in [SAPP05].

Another way to render soft shadows with several light source samples is to generate a visibility channel encoding the percentage of the light source which is visible, as suggested by [HBS00]. In the initial version of the algorithm, only linear light sources could be used, but due to the research of [ZYD02] it was extended to polygonal light sources.

After the creation of comparatively few (for small linear light sources, only 2) shadow maps, depth discontinuities are detected in each map using image analysis techniques. Such discontinuities indicate a shadow boundary, separating a caster and a receiver. As a next step, a polygon is created on each found boundary, linking both objects in the front and in the back. These polygons are then rendered from the other sample point, whereas the color value of the polygon is faded from 0 on the closer points to 1 on the farthest points using Gouraud shading. The resulting image is the visibility channel as shown in Figure 4.2, with values

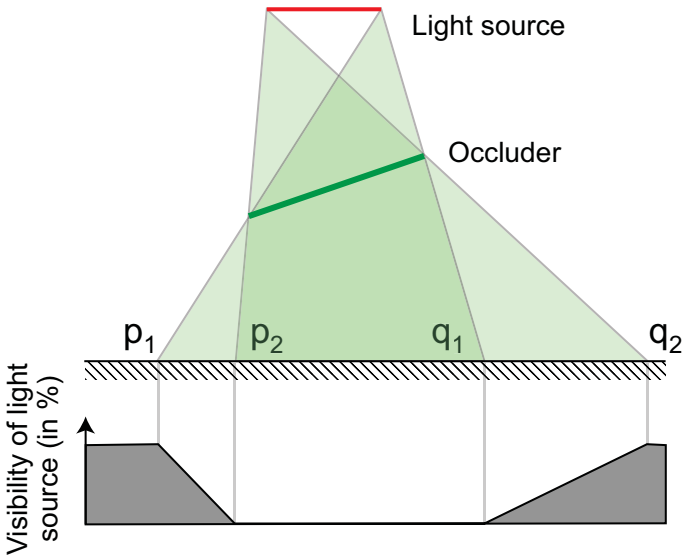

Fig. 4.2: The visibility channel (bottom) encodes the percentage of a linear light source that is visible [HLHS03].

between 0 (the light source is completely hidden) and 1 (the corresponding point is completely lit).

In the final rendering process, the shadow maps are used together with the visibility channel in order to render the soft shadows:

- If the point is hidden from the light source for all area light sample points, the point stays dark, as it lies in the umbra.
- If the point is visible from the light source for all area light sample points, the point gets illuminated.
- If the point is partly visible from the area light (it is visible from at least one sample point, and hidden from at least another sample point), it lies in the penumbra. The information found in the visibility channel is then used to calculate the according illumination.

This method is a lot faster than the previous approaches, as it requires significantly less sample points in order to calculate visually pleasing soft shadows. Undersampling leads to artifacts, though. The generated soft shadows are usually not physically accurate, but perceptually convincing. The rendering speed is mainly dependent on the complexity of the shadow casters.

4.2 Single sample soft shadow approaches with occluder search

A completely different approach to calculate soft shadows was first proposed by [PSS98] and later extended to be used in real-time with shadow mapping on graphics hardware by [BS02]. The basic idea is to use a single shadow map sample only, which is usually taken at the center of the area light source. As already explained in Section 3.2, single sample soft shadow approaches cannot compute physically accurate results, as the visibility is calculated for a single point only (like in hard shadow calculation).

The region of partial visibility of the light source is divided into an inner and an outer penumbra at the hard shadow border (see Figure 4.3) - an assumption which is physically incorrect. While [PSS98] only computes the outer penumbra (which darkens the scene significantly), [BS02] generates both an inner and an outer one. An enhanced version of the algorithm by [KD03], which is explained later, calculates only the inner penumbra.

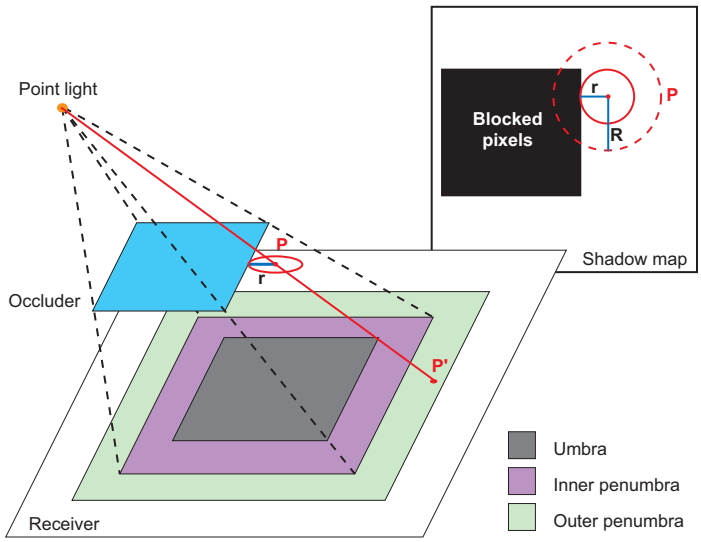

Fig. 4.3: The hard shadow is extended by an inner and an outer penumbra. For each pixel P', the corresponding shadow map texel P is estimated. Within a search radius R, the nearest blocked pixel in the shadow map is searched from there, and an attenuation coefficient is calculated based on the distance r between the hard shadow boundary and P [HLHS03].

The algorithm works by first creating a shadow map from the center of the light source like in standard shadow mapping. In the second rendering pass, the

shadow map is examined:

- Whenever the shadow map lookup returns that the current pixel is in shadow, we search the shadow map for the next illuminated pixel within a radius R.
- Otherwise, if the current pixel is visible from the light source center, we search the shadow map for the next shadowed pixel within a radius R (see Figure 4.3).

Since the penumbra size varies according to the distances between light source, occluder and receiver, the search radius R is modified to that effect.

Whenever the search is successful, the pixel is assumed to lie in the penumbra: Depending on the relative distance between the hard shadow boundary and the current fragment, an attenuation factor is calculated and used during the illumination process. The search gets more robust by using object ids as introduced in [HN85], which however makes self-shadowing impossible.

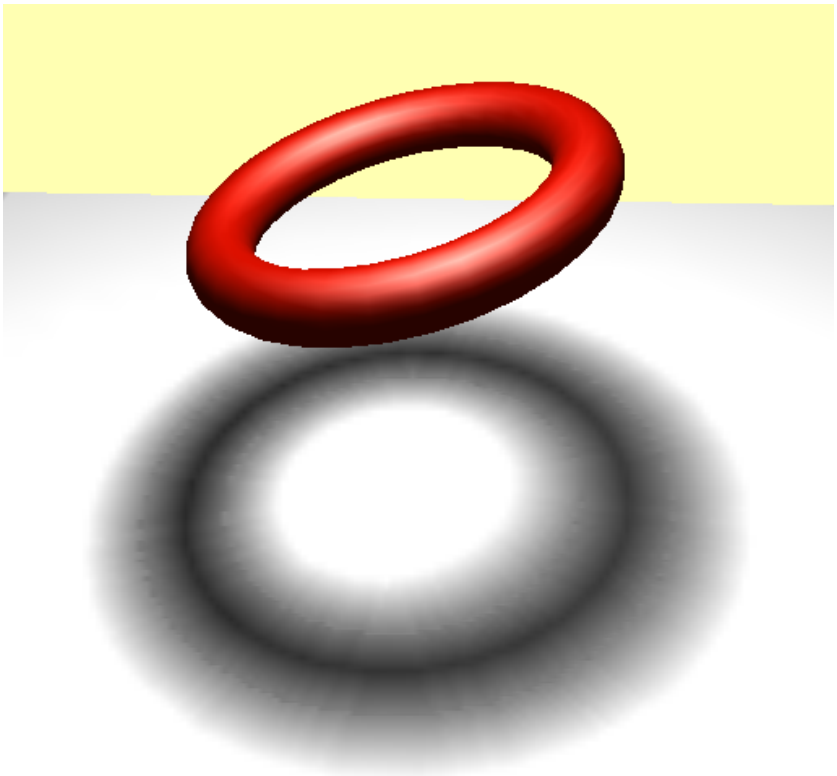

Fig. 4.4: A screenshot showing the soft shadows generated with the algorithm presented in [BS02].

The method produces visually plausible fake soft shadows at a comparatively high speed, since only a single shadow maps needs to be computed (See Figure 4.4). Due to the adjustable search radius R, the effect of smaller penumbrae at regions where an occluder is close to a receiver can be simulated quite well. The search in the shadow map causes the biggest performance hit, though: Especially for large penumbrae, it is by far the computationally most expensive part of the algorithm.

This problem can be avoided by using the approach proposed in [KD03]: Instead of the expensive search, a so-called *shadow-width* map, which contains the distance to the nearest illuminated point for each point in shadow, is precomputed. This is done by applying a smoothing filter on an inverted binary occlusion map several times (see Figure 4.5). This way the distance to the hard shadow boundary can be easily determined by a texture lookup in the shadow width map instead of the costly search.

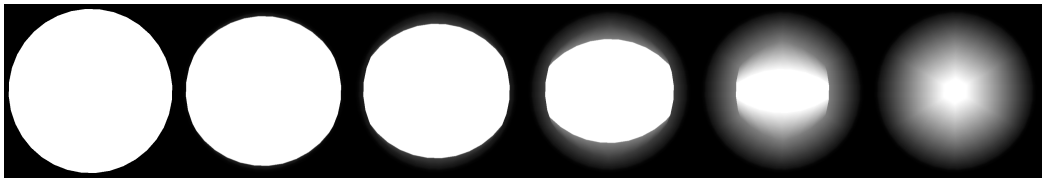

Fig. 4.5: An example shadow-width map generated by texture compositing [KD03].

This method speeds up the algorithm significantly, so that real-time frame rates can be achieved. Moreover, since it is an image-based approach and no geometrical edge detection is necessary, it is independent of the scene complexity. Unfortunately, the approach suffers from artifacts (for example due to overlapping occluders in light space) and physical incorrectness: Since only the inner penumbra is computed, the shadow is significantly smaller than the correct result.

4.3 Percentage closer filtering

Percentage Closer Filtering (PCF) was first introduced by [RSC87] in order to reduce the aliasing artifacts which appear during shadow mapping at the hard shadow boundaries. It works by comparing the depth of the processed point not only to its corresponding pixel in the shadow map, but also to its neighboring pixels there. The percentage of successful shadow tests specifies the shadow intensity. A slightly modified version of PCF with a 2x2 filter kernel is implemented in hardware on today's consumer GPUs (the 4 shadow test values are bilinearly interpolated according to their shadow map texture coordinates).

Since PCF softens the shadow boundary, it can also be seen as a penumbra caused by an area light source. This effect can be intensified by using a larger filter kernel, which increases the blurring effect and therefore the size of the penumbra. Due to the vast amount of texture lookups needed, this has a major impact on the performance, though. Furthermore, the penumbra is far from being physically accurate, as it always has the same size (regardless of the distances between light, blocker and receiver).

A solution for this problem called Percentage Closer Soft Shadows (PCSS) was proposed in [Fer05]: By using an additional blocker search in the shadow map, the filter kernel can be adjusted according to the relation between light, blocker and receiver (see Figure 4.6). Even though this idea still gives no physically correct soft shadows, the generated penumbrae are visually far more plausible, as they vary in size. This technique has already been used in various 3D games, as it has no problems with dynamic scenes, is easy to implement and computationally fast (it can be implemented in a way that it operates on the GPU only).

The PCF filtering step in PCSS can be replaced by applying other techniques in which the shadow map is pre-filtered, like for example Variance Shadows Maps (Section 4.3.1) or Convolution Shadow Maps (Section 4.7).

Fig. 4.6: Screenshot from a PCSS sample application. Notice the varying penumbra size [Fer05].

Fig. 4.7: Light bleeding artifacts can occur when using *variance shadow maps* (original image from [DL06], contrast has been enhanced in marked area for visibility reasons).

4.3.1 Variance shadow maps

In [DL06], the observation is made that in the PCF algorithm (Section 4.3) a single depth value is in fact compared to a particular distribution of depth values. It is only necessary to know the percentage of values in the sampled shadow map region which are larger than the depth of the current screen space pixel - the individual shadow tests don't matter. The author therefore suggests the introduction of so-called *variance shadow maps* (VSM), which are able to represent a distribution of depths at each pixel.

A VSM differs from a regular shadow map by storing the squared depth in a second channel. The generated VSM is then pre-filtered and blurred. By sampling the map, we therefore obtain the mean of the depth and the squared depth from the corresponding filter region, called the moments M_1 and M_2. The mean μ and the variance σ^2 are then computed:

$$\mu = M_1 \tag{4.1}$$

$$\sigma^2 = M_2 - M_1^2 \tag{4.2}$$

These parameters describe the distribution of depth values in the filtered area. The variance is interpreted as an estimation of the width of a distribution.

In order to find out the percentage $P(x \geq t)$ of values in a filter region which would fail the depth comparison test with a fixed depth t, Chebychev's inequality is used:

$$P(x \geq t) \leq p_{max}(t) \equiv \frac{\sigma^2}{\sigma^2 + (t - \mu)^2} \tag{4.3}$$

Even though the value we can compute is only an upper bound, we can use it as a good approximation for the percentage of pixels failing the shadow test in PCF filtering.

By applying an additional blocker search as introduced in the PCSS algorithm [Fer05], VSMs can be used to generate soft shadows with varying penumbra size. The algorithm is easy to implement and fast, and is therefore widely used in real-time applications and games. A big advantage is that high-quality pre-filtering of the VSM is possible.

Unfortunately, the approach suffers from light bleeding whenever the shadow of two shadow casters, which are relatively far away from each other, overlap (see Figure 4.7). A way to reduce these artifacts as well as suggestions on how to improve the filtering of the VSM can be found in [Lau07].

4.4 Smooth penumbra transitions with shadow maps

This image-based algorithm presented in [dB04] introduces a data structure called *skirt*, which is associated with an occluder silhouette edge and contains information like an attenuation factor and a depth value (see Figure 4.8). It is possible to render visually plausible fake soft shadows with inner and outer penumbrae.

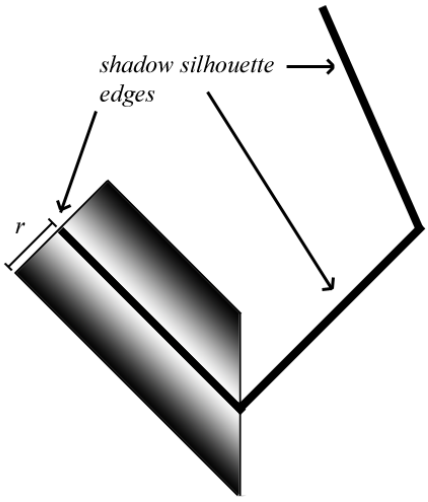

Fig. 4.8: The *skirt* data structure: It contains attenuation and depth information and is associated with a silhouette edge of the shadow caster [dB04].

The method consists of several steps (see Figure 4.9):

- A regular shadow map is generated from the center of a spherical light source.

- This shadow map is filtered with an edge detection kernel, leading to a representation of occluder edges in a skirt buffer. This buffer initially contains the depth values as well as an attenuation factor of 1 for all silhouette edges.
- By repeatedly applying a smoothing filter, the skirt is widened, leading to a soft decrease of the attenuation factors. The according depth values are estimated by taking the minimum depth of all pixels the filter is applied to. The number of iterations depends on the area light radius.
- In a final rendering step, the information stored in both the shadow map and the skirt buffer are used to illuminate the scene.

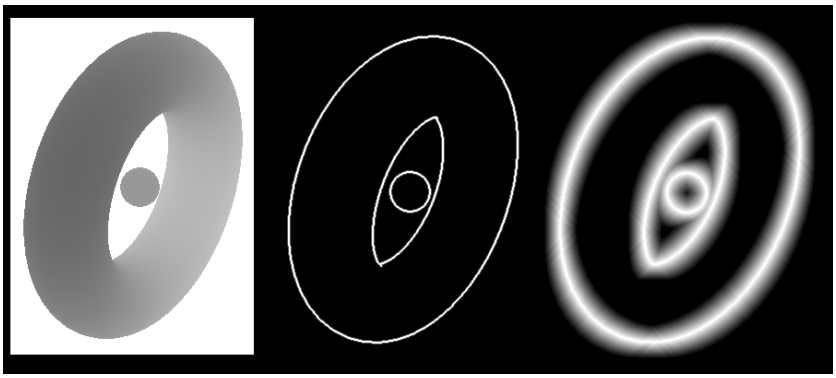

Fig. 4.9: The necessary steps to generate the skirts: The shadow map (left), its shadow silhouettes (middle) and the final skirts buffer (right) [dB04].

In order to calculate the inner penumbra, an additional search in the shadow map (comparable to [BS02] presented in Section 4.2) is necessary. Of course, the algorithm's performance depends heavily on the penumbra size: Both the search and the iterations needed to filter the skirt become computationally quite expensive in case the partly occluded area is large. The biggest advantage, on the other hand, is the fact that the approach is image-based, so the complexity of the scene objects does not have a big impact on the rendering speed.

4.5 Image-space flood-fill soft shadows

Another purely image-based algorithm is presented in [AHT04]. The main idea is to use a modified flood-fill algorithm in screen space to generate a penumbra out of hard shadows, which are created using standard shadow mapping in a first depth pass and a following shadowing pass. During this second pass, a pixel classification is performed: every pixel in screen space lying on a shadow boundary

is marked, and its shadow mapping coordinates are saved. The other pixels are either marked as inside or outside the shadow. Care must be taken whenever a lit object lies over a hard shadow cast by another object: In such a case, the screen-space shadow boundary should not generate a hard shadow (see Figure 4.10)! This can be avoided by checking if the pixel on the boundary is a silhouette pixel in the shadow map (this information can be extracted out of the shadow map by applying a simple edge detection filter).

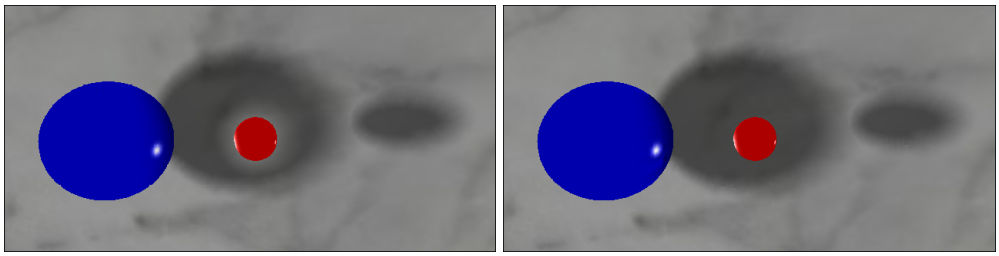

Fig. 4.10: The generation of false penumbra regions (left) can be avoided by boundary pixel verification (right). This is done by verifying that each classified boundary pixel is occluded by a silhouette pixel as seen from the light source [AHT04].

For each pixel which is lying in the penumbra (at the beginning, these are only the pixels classified as shadow boundaries), a visibility factor needs to be calculated. This is done by casting a ray from the processed pixel through the corresponding shadow map coordinate, intersecting the light source. There, a line, lying perpendicular to the connection between intersection point and area light center, is generated. This line "cuts" the light source into visible and invisible part. The visibility factor is then calculated by dividing the visible part by the total light source area (See Figure 4.11).

After the classification and initialization, several flood-fill render passes follow, in which the penumbra is spread: For each pixel in screen space, the 8-connected neighborhood is searched for shadow boundary pixels. Whenever the search is successful, the penumbra is widened, a visibility factor is calculated, and the shadow map coordinates from the neighbor pixel are stored. This is repeated until no more lines during the visibility calculations intersect the light source, or until a specified number of iterations has been reached.

It is possible to create visually plausible soft shadows with this approach, even though they are not physically correct. Artifacts can occur whenever umbrae overlap, though, and the rendering speed depends heavily on the penumbra size, which can be quite large for certain viewing positions and far extended light sources. Moreover, flood-filling is not really well-supported on today's graphics hardware, which makes the computation costly, the implementation complicated, and real-time frame rates hard to achieve.

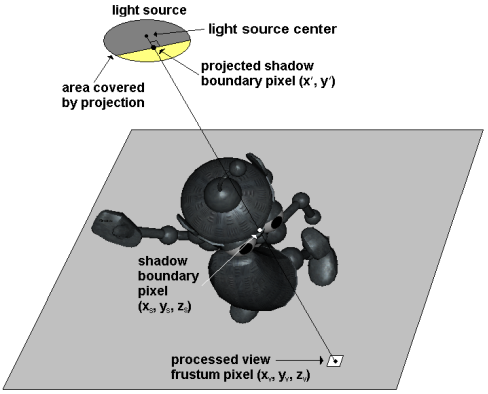

Fig. 4.11: Estimating the amount of occlusion by casting a ray through the corresponding shadow map coordinate [AHT04].

Fig. 4.12: Screenshot from a test application with soft shadows generated with the screen space flood-fill algorithm [AHT04].

4.6 Occlusion textures

In [ED06] the use of so-called *occlusion textures* is suggested in order to generate approximate soft shadows. These occlusion textures are generated by "cutting" the scene into 4 to 16 slices parallel to the light source, which are then projected onto separate binary textures. After pre-filtering, each texture is stored in a channel of a RGBA-texture, so that only 1 to 4 RGBA textures are needed to save all occlusion maps.

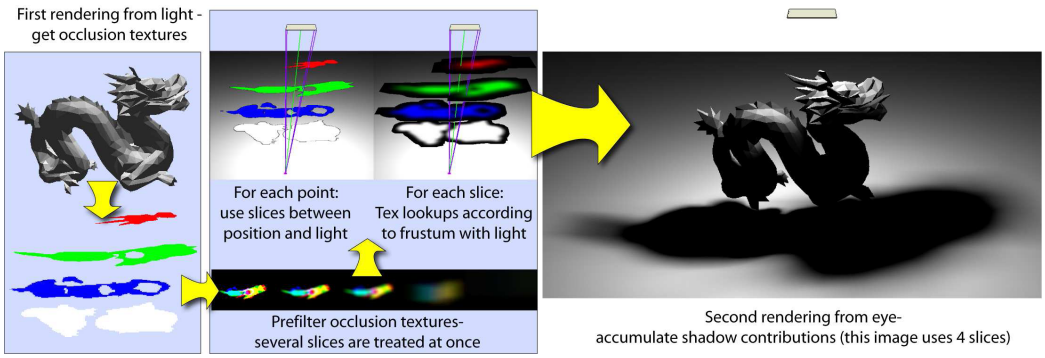

Fig. 4.13: Summary of the *occlusion textures* algorithm [ED06].

During the shadowing pass from the regular camera view, all slices between the processed pixel P and the light source are visited and sampled with a kernel size according to the projection of the area light on the slice as seen from P. The contributions of the different slices are then combined and generate a smooth soft

shadow. See Figure 4.13 for a summary of the algorithm.

Since the scene is split up into several slices in order to generate the occlusion textures, the algorithm cannot handle self-shadowing completely correctly, and the generated shadow is of course not physically exact. Still, the shadows are quite plausible, and since this algorithm is image-based and can be implemented in a way that nearly all work is done on modern programmable graphics hardware, real-time frame rates can be achieved.

4.7 Convolution techniques

Fig. 4.14: By convolving an image of the light source (left) with the receiver (middle), soft shadows (right) can be generated [SS98].

Using convolution for soft shadow generation was first proposed in [SS98]. Although purely image-based, the algorithm does not rely on shadow mapping. The main idea is to convolve an image of the area light with an image of the blocker. The resulting image is then used to modulate the illumination. See Figure 4.14 for an example.

At first sight, this method seems to be very fast and easy to use, especially since the calculation of the convolution can be done in a computationally inexpensive way on the GPU using the Fast Fourier Transformation (FFT). Unfortunately, valid soft shadows can only be generated whenever the area light, the blocker and the receiver lie in parallel planes. The author therefore suggests an error driven subdivision algorithm for a reasonable shadow quality in general scenes, which slows down the algorithm significantly. Moreover, it does not work in all cases, especially whenever elongated objects are lying parallel to the light direction.

Another approach using convolution is presented in [AMB+07]: It tries to combine shadow mapping with convolution techniques by introducing so-called *convolution shadow maps* (CSM). The idea is to pre-filter the shadow map, which

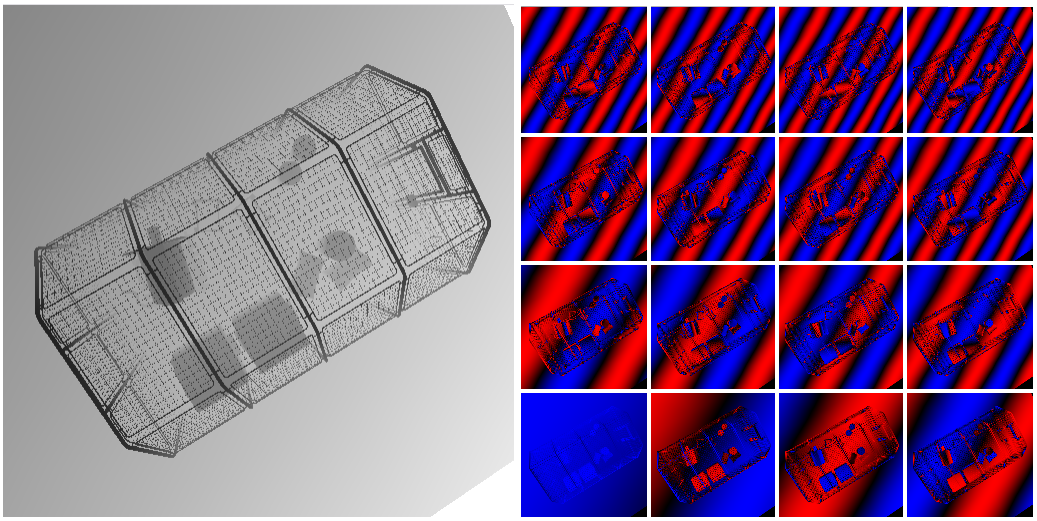

Fig. 4.15: The shadow map and the corresponding base images B_i, generated with Fourier expansion [AMB+07].

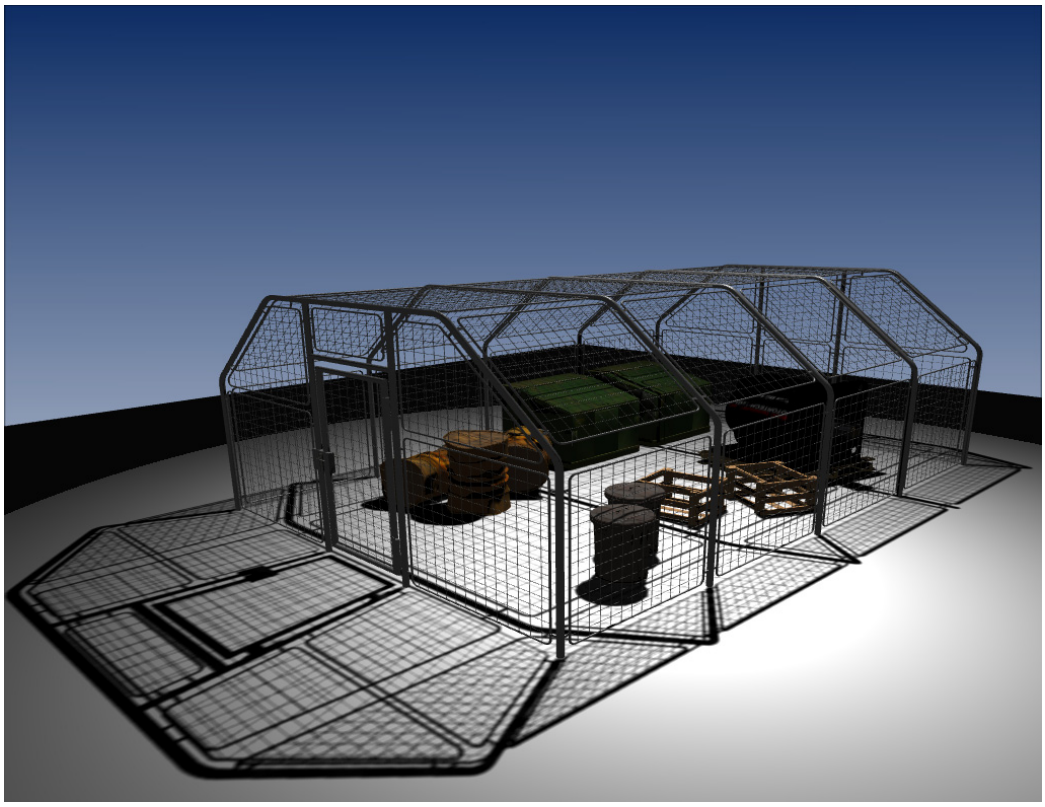

Fig. 4.16: Result image with smooth shadow borders, generated with the technique presented in [AMB+07].

requires to expand the shadow function $s()$

$$s(x) = f(d(x), z(p)) := \begin{cases} 1 & \text{if } d(x) \leq z(p) \\ 0 & \text{if } d(x) > z(p) \end{cases} \tag{4.4}$$

into a separable Fourier series

$$f(d(x), z(p)) = \sum_{i=1}^{\infty} a_i(d(x)) B_i(z(p)), \tag{4.5}$$

where $d(x)$ is the distance of the processed screen-space pixel x to the light source, and $z(p)$ is the depth value found for the corresponding shadow map pixel p. B_i are the corresponding basis images, and a_i are coefficients depending on the distance. Due to the linearization of the shadowing function, it can now be filtered by convolving it with a kernel w:

$$\begin{aligned} s_f(x) &= [w * f(d(x), z)](p) \\ &\approx \sum_{i=1}^{N} a_i(d(x))[w * B_i](p). \end{aligned} \tag{4.6}$$

In practice, the filtering is done by using mipmapping or by computing summed area tables [Cro84]. During the shadowing step, the basis images B_i are sampled, weighted by $a_i(d(x))$ and summed up in order to obtain soft shadow borders (see Figure 4.16).

The authors suggest using this approach only to avoid aliasing artifacts and not for the generation of soft shadows, since the penumbra size would always be the same (like in PCF, see Section 4.3). This problem has recently been solved in [ADM+08], where a blocker search (which is also based on convolution) helps to dynamically set a suitable filter kernel size and to generate visually plausible fake soft shadows.

4.8 Soft shadow mapping with backprojection

Several papers (e.g. [GBP06], [AHL+06], [ASK06], [SS07], [GBP07]) have recently been released, which all suggest to use a technique originally presented in [DF94] in order to generate soft shadows. The idea is to use a single shadow map (generated with the standard shadow mapping algorithm) not only for depth comparison, but to see it as a discretized representation of the scene. For each shadow map pixel, the shadow map coordinates (u, v) and the corresponding depth z are known, making it possible to project these so-called *micropatches* into the scene. In order to calculate the visibility factor v for a screen-space pixel p, each of these

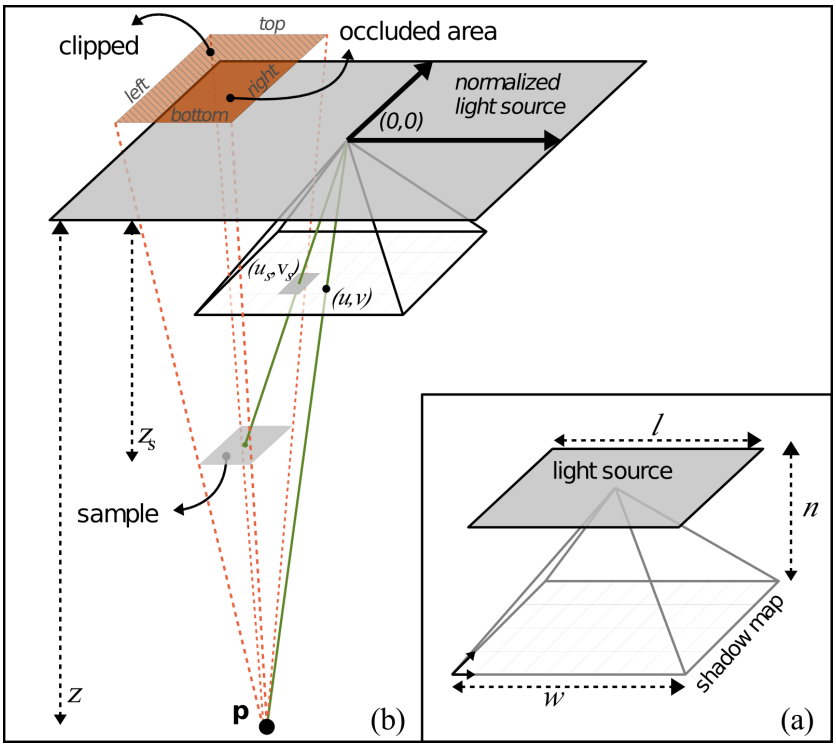

Fig. 4.17: Each shadow map sample is projected into the scene as a so-called *micropatch*. This patch is then backprojected onto the light source, where its percentage of occlusion is estimated [GBP06].

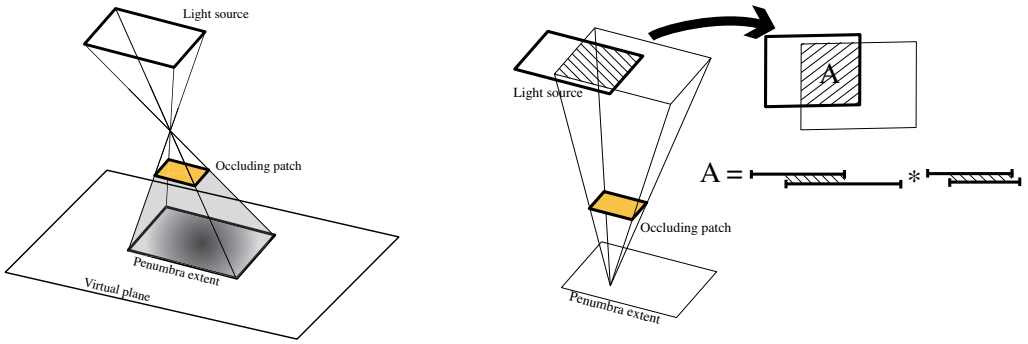

Fig. 4.18: Creation of a *soft shadow map* according to [AHL+06]: For each micropatch, a penumbra area is estimated (left). For all pixels in this penumbra region, the patch is backprojected and its percentage of occlusion is determined (right). This is done for all micropatches, and the results are summed up in the soft shadow map, which is used to illuminate pixels in the penumbra accordingly.

micropatches is backprojected from p onto the light source. The process of backprojecting pixels onto the shadow map is illustrated in Figure 4.17.

In case of an intersection, the fraction of occlusion is determined and subtracted from 1 (visibility factor $v = 1$ indicates a completely lit, $v = 0$ a completely occluded point p). The visibility factor v is then directly used to modify the illumination of the screen space pixels, generating very convincing soft shadows. Alternatively, the amount of occlusion of a micropatch can also be estimated by using the relative solid angle of its bounding sphere [ASK06].

Since every single micropatch is a potential occluder for every processed pixel p, several ways to optimize the backprojection and occlusion have been suggested to make this method usable in real-time applications:

- In [AHL+06], receivers and occluders are separated, and a so-called *soft shadow map* is generated. This is done by first estimating the penumbra extent of a single micropatch, and then backprojecting this patch for every single screen space pixel p which lies in this penumbra area (see Figure 4.18). The computed visibility amounts are stored in the soft shadow map. After looping over all micropatches, the map is used to render the penumbrae accordingly.

- Other approaches like [GBP06] or [SS07] don't use a soft shadow map, but calculate the backprojection directly for each point by considering only the micropatches which can have an influence on the visibility. This requires a search in the shadow map, which is computationally expensive. [GBP06] therefore suggests an approach in which the shadow map is converted into a *hierarchical shadow map* (HSM). It is comparable to a mip-map structure, but saves the minimal and maximal depth of the previous level, and can be used to decrease the amount of necessary backprojections. A similar data structure is proposed in [SS07].

- Instead of the search, [ASK06] suggests to sample the micropatches to be backprojected according to a Gaussian Poisson distribution.

Like any other single sample soft shadow algorithm, the backprojection method suffers from the fact that the visibility is computed for a single point only, leading to noticeable artifacts in certain cases. Other artifacts occur due to gaps and overlaps of the backprojected patches, as shown in Figure 4.19. While [GBP06] fixes the gaps by extending the patches, [SS07] places samples at the light source and uses a bitfield there to track which of them is occluded, and [GBP07] tries to solve the problems with an additional occluder contour detection.

Despite these limitations and drawbacks, soft shadows generated by backprojection are visually and physically very plausible. They benefit a lot from recent

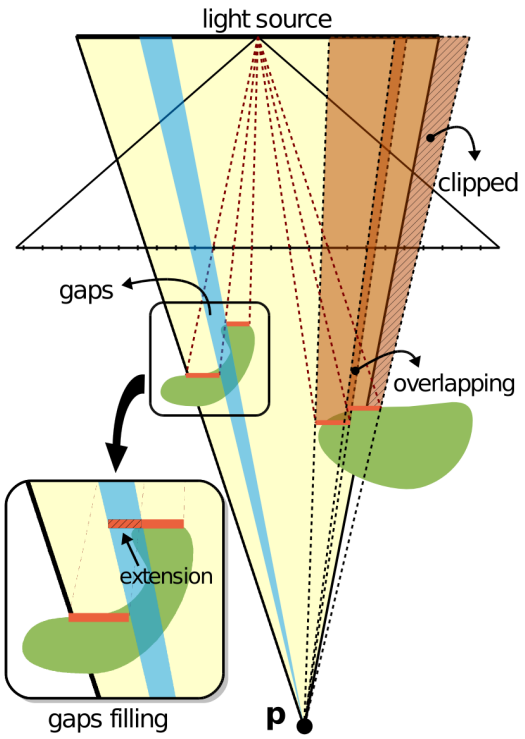

Fig. 4.19: The backprojection can lead to artifacts: Gaps and overlaps are likely to occur. In [GBP06], the micropatches are extended, so the gaps can be avoided.

hardware developments (for example, the bitfield technique proposed in [SS07] is only possible on GPUs which support the Shader Model 4), and it is possible that they will be used in real-time games and applications as soon as graphics hardware is fast enough.

4.9 Soft planar shadows using plateaus

The approach presented in [Hai01] does not rely on shadow mapping, but is an object-based method. It consists of two passes:

- First, an attenuation map is created. This is done by rendering hard shadows into a texture which is "lying" on the planar receiver. These hard shadows are computed using the shadow volume algorithm and form the umbra. In order to generate the penumbra, cones placed at the silhouette vertices as well as polygons connecting these cones at their outer tangents are rendered with decreasing shadow intensity from the umbra outwards (see Figure 4.20).

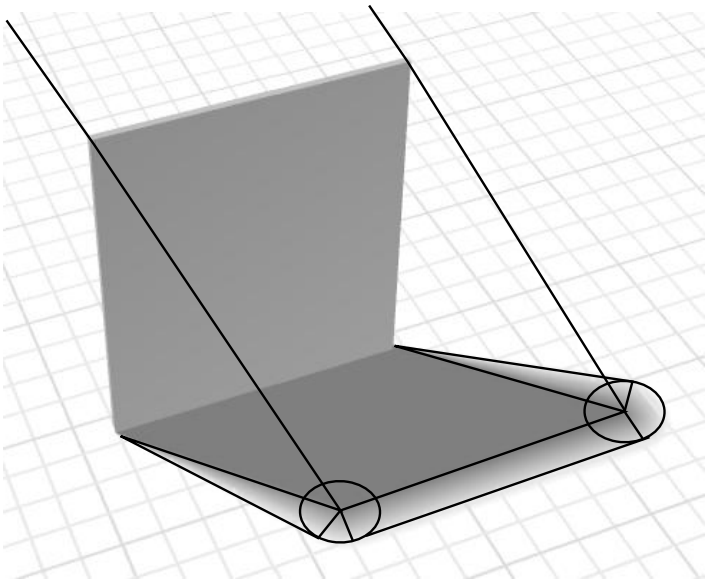

Fig. 4.20: The occluder shadow volume is extended by cones and planes, generating an outer penumbra [Hai01].

- In the second pass, the generated attenuation map is used to modify the illumination on the planar receiver.

Apart from the typical fill-rate problems which occur in all object based approaches, this method suffers from the fact that it is limited to planar receivers and a spherical light source. Moreover, only the outer penumbra is calculated, leading to a darker shadow than expected and a constant umbra size regardless of the area light source extents.

4.10 Penumbra maps

In [WH03], an approach very similar to the algorithm presented in Section 4.9 is suggested. It avoids the fill-rate bottleneck by using the shadow mapping algorithm instead of the shadow volume algorithm to generate the hard shadows, and introduces a so-called penumbra map, which stores the shadow intensity of the corresponding shadow map pixel instead of using a texture on the planar receiver. Three passes are necessary:

- A standard shadow map is created from the center of the spherical light source.
- Then, the penumbra map is generated. Cones and sheets are placed at the vertices and edges similar as in [Hai01], generating the penumbra regions.

Fig. 4.21: The shadow map (left) and the corresponding *penumbra map* (right) of a test scene [WH03].

The intensity of each of the pixels lying there can be evaluated using shader fragment programs and is stored in the penumbra map (see Figure 4.21):

$$I = 1 - \frac{Z_P - Z_F}{Z_P - Z_{v_i}} = \frac{Z_F - Z_{v_i}}{Z_P - Z_{v_i}}, \tag{4.7}$$

where Z_{v_i} is the current vertex depth from the light, Z_F the depth of the current cone or sheet fragment, and Z_P the corresponding shadow map depth.

- In the final pass, the information stored in both the shadow map and the penumbra map is used to illuminate the scene accordingly.

Despite the use of the shadow mapping algorithm and the hence resulting increased robustness, it is still necessary to detect the object silhouettes in order to generate the cones and sheets, making this algorithm dependent on the scene complexity. Moreover, the limitation that only the outer penumbra is generated still exists.

4.11 Smoothies

Another very similar technique to the algorithms presented in Sections 4.9 and 4.10 is suggested in [CD03]: After creating a shadow map, geometric primitives are attached to the silhouette edges of the objects, which are extracted in object space (see Figure 4.22). These primitives are called *smoothies* and are rendered from the light's view into a so-called *smoothie buffer*, storing a depth and an alpha value for each pixel, which can be seen in Figure 4.23. The alpha value is dependent on the ratio of distances between the light source, blocker, and receiver.

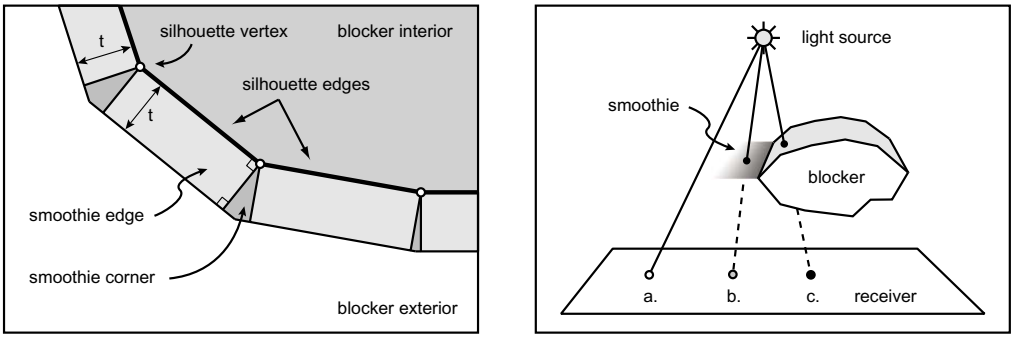

Fig. 4.22: Smoothies are created by extending the occluders' silhouettes by rectangles, which are connected at the corners (left). The smoothie then defines whether a pixel is lit, completely dark or in the penumbra (right) [CD03].

During rendering from the observer's view point, the generated information stored in the smoothie buffer as well as the shadow map is used to generate soft shadows:

- If the shadow map lookup returns that the processed point is in shadow, it stays dark.
- Else, if the sample's depth value is greater than the smoothie buffer depth's value, the point lies in the penumbra, and the corresponding alpha value is used to modify the illumination.
- Otherwise, the point is completely unoccluded.

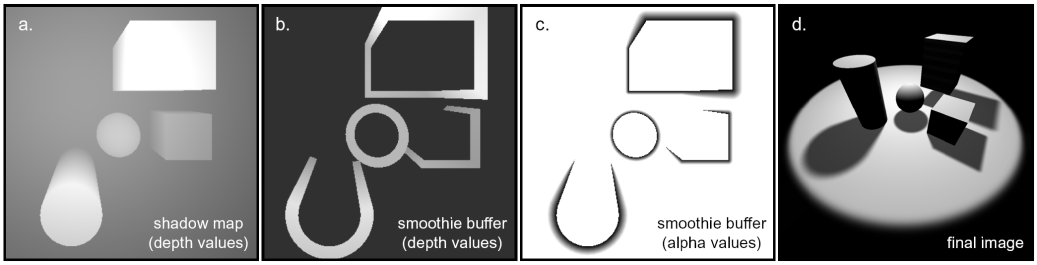

Fig. 4.23: From left to right: The standard shadow map, the depth values of the smoothie buffer, the alpha values of the smoothie buffer, and the final rendered image [CD03].

As in the two previously presented algorithms, only the outer penumbrae are calculated, and the object-based silhouette extraction precludes the algorithm to be used in arbitrarily complex scenes. Furthermore, connecting the smoothie edges is a big problem for large light sources, leading to noticeable artifacts.

4.12 Penumbra Wedges

An extension to the shadow volumes algorithm as explained in Section 2.3.2 was first presented in [AMA02] and later improved in [AAM03] and [ADMAM03]. The main idea is to calculate for each polygon which is part of the shadow volume a so-called *penumbra wedge* - a volume representing the penumbra (see Figure 4.24).

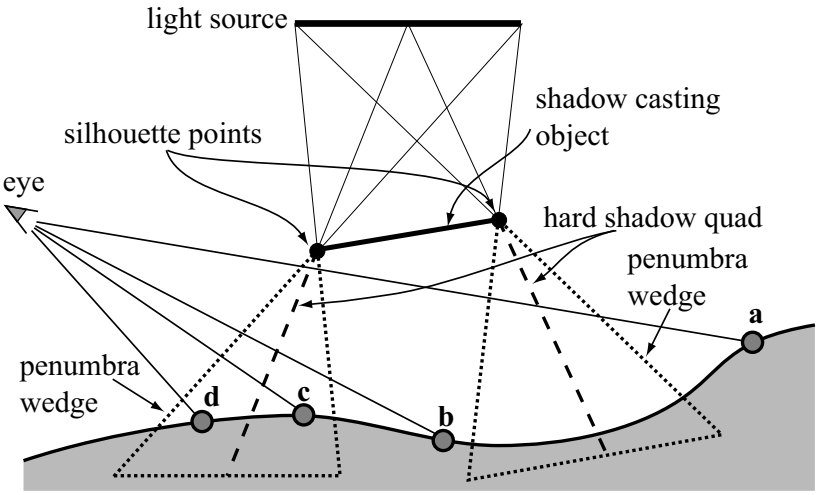

Fig. 4.24: For each hard shadow quad generated with the standard shadow volume algorithm from the center of the light source, a penumbra wedge volume is created [AAM03].

Instead of using the stencil buffer like in the hard shadow version, a high-precision floating point visibility buffer is used and initialized with a value of 1.0, which indicates that the viewer is outside any shadowed region. The algorithm then consists of the following steps:

- First, the scene is rendered in order to initialize the z-Buffer.
- In the second pass, the hard shadow volumes are then rendered exactly as in the regular shadow volumes algorithm, with the z-test enabled. Front-facing polygons decrease the value in the visibility buffer by 1.0, back-facing polygons increase it by 1.0. The visibility buffer now contains the hard shadows.
- Next, the soft shadow pass is executed: The front faces of all penumbra wedges are rendered, and for each pixel covered by these triangles, its intensity is calculated in a fragment program by computing the percentage of occlusion of the area light source. For this, the corresponding silhouette

edge and its hard shadow quad are projected onto the light source, where the coverage ratio is estimated (see Figure 4.25). According to this intensity, the visibility buffer is updated: If the point lies outside the hard shadow borders, the intensity value is subtracted, if it lies inside, the intensity is added. The visibility buffer now contains the amount of light source visibility for all pixels.

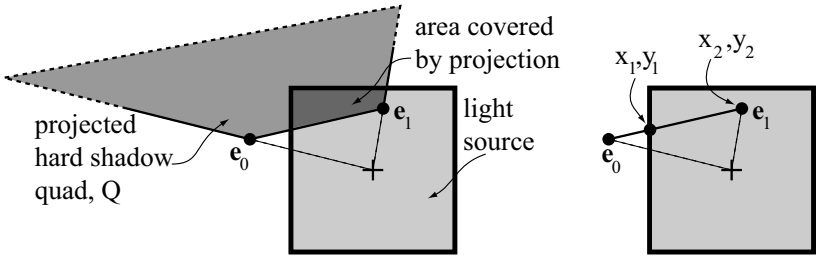

Fig. 4.25: Right: The hard shadow quad is projected onto the light source (as seen from pixel P, which lies in the penumbra wedge). Right: The silhouette edge e_0e_1 is clipped and used to estimate the percentage of occlusion [AAM03].

- In the final pass, the information stored in the visibility buffer is used to illuminate the scene.

The major drawback of this algorithm is, like in the standard hard shadow version, its fill-rate bottleneck. For scenes with a suitable polygon count, high frame rates and visually very convincing soft shadows (which are even physically correct in simple cases) can be achieved, though. Problems arise whenever the light source is very close to the receiver, so that the actual silhouette is very different from that generated from a single sample (see Section 3.2.1), or whenever several occluders generate a combined shadow (see Section 3.1.4). The authors suggest splitting up the light source in such cases. In [FBP06], the problem with the combined shadows of several occluders is treated with a special penumbra blending strategy.

4.13 Summary

In this Chapter, many state-of-the-art real-time soft shadow algorithms have been presented and explained. It is obvious that none of them can be seen as the "perfect" solution for any arbitrary scene configuration - it depends heavily on the application or game itself, which approach is to prefer, since all of them have their drawbacks. Still, the result images show us that both the image quality as well as

the realism increase dramatically whenever soft shadows are used instead of hard shadows. It will be interesting to watch the further progress in this research area, especially since multi-core GPUs are becoming more and more popular, from which highly parallelizable algorithms (like many soft shadow algorithms here) could gain a lot.

Where there is much light, the shadow is deep.

(*Johann Wolfgang von Goethe)*

Chapter 5

Real-Time Soft Shadows using Temporal Coherence

As shown in Chapter 4, a vast amount of real-time soft shadow algorithms have been developed during the past few years: While most of them use a fast single shadow map sample approach combined with clever filtering techniques or additional geometry to generate so-called "fake soft shadows", some slower algorithms (like in Section 4.1) compute the penumbra by taking multiple samples of the area light source and combining them to a physically accurate soft shadow.

In this Section, we present a new soft shadow algorithm, which is able to combine the speed of single sample approaches with the accuracy of multi sample approaches: The area light source is sampled over time, and only a single shadow map is calculated and evaluated per frame. By storing this shadowing information into a screen-space *shadow buffer* and employing temporal coherence [SJW07] together with spatial filtering, correct and very fast soft shadows can be generated. We recently submitted a corresponding paper [SSM09] to the *Graphics Interface 2009* conference.

5.1 Introduction

In Chapters 3 and 4 it was shown how important soft shadows are for a plausible perception of a computer-generated scene, and which algorithms are currently used in real-time applications in order to compute them: Most real-time soft shadowing methods approximate an area light by a point light source, which is located at the center of the area light source. By using clever filtering methods or with the help of additional geometry which is plugged onto the umbra, a perceptually plausible fake penumbra is estimated. Other single sample approaches use back-projection techniques; treating the shadow map texels as micro-patches and projecting them back onto the area light source, where they are used to estimate the amount of occlusion (see Section 4.8). Depending on the scene, such single sample approaches can generate visually pleasing (and, in the case of back-projection

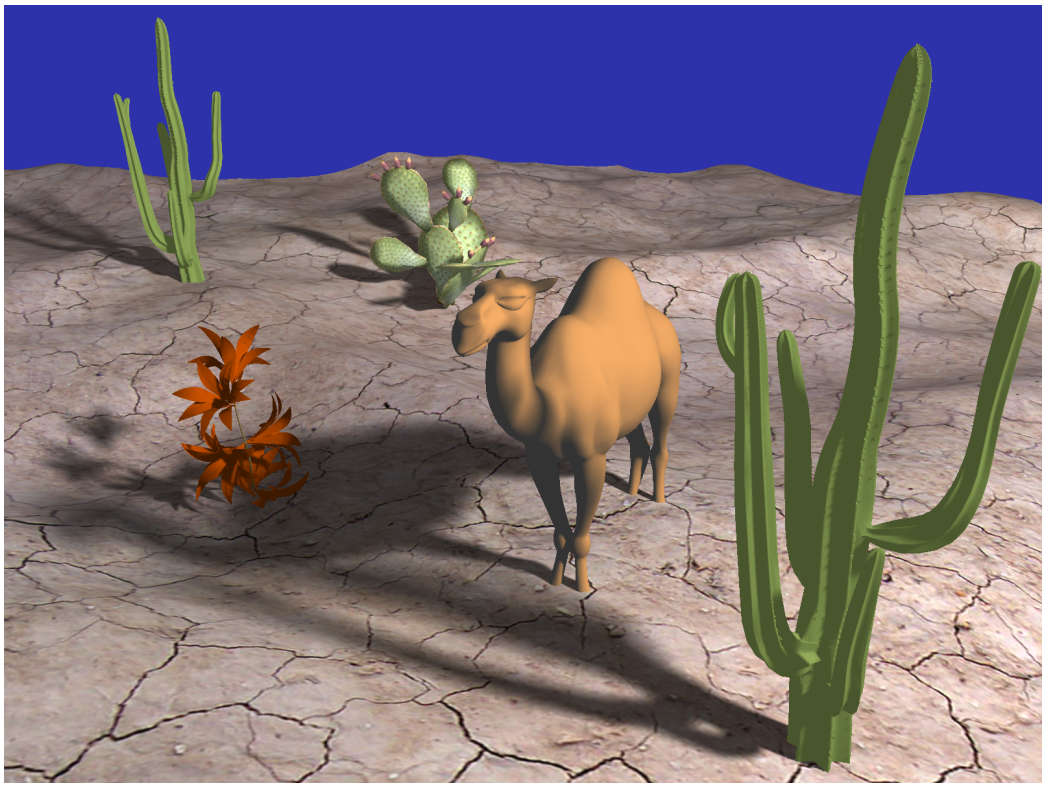

Fig. 5.1: This image shows a scene rendered with our method with overlapping occluders with 70k triangles at 344 FPS.

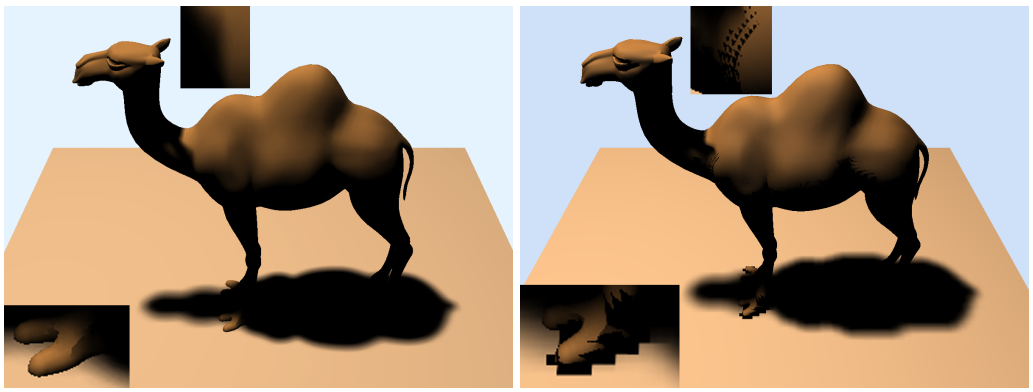

Fig. 5.2: *Left side*: Our Method (634FPS). *Right Side*: Bitmask Soft Shadows (156 FPS) [SS07]. Even very good single sample soft shadow methods show some artifacts, like biasing problems and contact shadow undersampling. Both can be avoided by using multiple samples as in our approach.

techniques, even physically plausible) soft shadows, especially since the human eye is not very sensible to their correctness. Still, no single sample algorithm can be used to calculate *physically accurate* soft shadows, as the area-from-point visibility problem is always replaced by a much simpler point-to-point visibility problem, and some artifacts caused by overlapping occluders or large penumbras are very likely to occur.

In order to generate accurate soft shadows, light source area sampling has to be used (see Section 4.1): The idea is to calculate many hard shadows by sampling the area of the light source and combining the results. Due to the vast amount of samples which are necessary to generate visually plausible shadows, a real-time application is problematic - but under the assumption of a light source with uniform color and dense enough sampling of the light source, the result of these approaches are correct soft shadows.

The new approach described here can be seen as a combination of light source area sampling and single sample filtering: Each frame, a single shadow map from a randomly selected position on the area light is created and used to update a screen space buffer called the *shadow buffer*. This buffer contains soft shadowing information accumulated over the previous frames for each screen space pixel. Whenever a buffer pixel needs to be updated, the newly gathered shadow map is used to evaluate the current shadow information, which is then added to the shadow buffer. The stored data is used to estimate the shadowing amount of this pixel. As long as the soft shadow information is judged to be insufficient, additional spatial filtering is applied to the buffer. For this, the soft shadow information from the neighboring pixels of the shadow buffer is gathered, and in combination with a penumbra size estimation, the shadows converge very fast (3-30 frames), smooth and graceful to the correct solution and can finally be rendered directly from the shadow buffer.

By combining temporal coherence as introduced by [SJW07] (through the use of the shadow buffer) with spatial filtering (penumbra estimation and pixel neighborhood), this soft shadow mapping algorithm produces *accurate real-time* soft shadows which are as fast as an optimized variant of PCSS [Fer05] (see Section 5.4).

5.2 The Algorithm

In order to calculate the amount of illumination of a fragment in a scene, the fraction of the area light source that is visible from this fragment position has to be estimated. In the presented algorithm, the reverse value, called occlusion percentage or soft shadow result $\psi(x, y)$ for a fragment (x, y), is used: $\psi(x, y)$ is 0 for a fragment that is illuminated by the whole area of the light source, and 1 for

a fragment that is not illuminated by the light source at all.

5.2.1 Estimating Soft Shadows from n Samples

The soft shadow information is calculated by approximating the area light source by n different point light sources, which are randomly distributed over the area light. In order to obtain a high frame rate and to make the calculation feasible for rasterization hardware, sampling and shadow mapping (see Section 2.3.1) are used. A shadow map is evaluated for each point light source, providing information on whether a screen space fragment is illuminated by it or not.

$$\tau_i(x,y) = \begin{cases} 0 & \text{lit from point light } i \\ 1 & \text{in shadow of point light } i \end{cases} \tag{5.1}$$

$\tau_i(x,y)$ is the result of the hard shadow test for the ith shadow map for the screen space fragment at position (x,y). Assuming that the point light placement on the area light source is sufficiently random, the soft shadowing result ψ (i.e., the fractional light source area occluded from the fragment) can be estimated by the proportion $\hat{\psi}_n$ of shadowed samples

$$\hat{\psi}_n(x,y) = \frac{1}{n}\sum_{i=1}^{n}\tau_i(x,y). \tag{5.2}$$

The number of occluded samples $n\hat{\psi}_n$ underlies the Binomial distribution with variance $n\psi_n(1-\psi_n)$. It is therefore possible to give an unbiased estimator for the variance of the proportion $\hat{\psi}_n$ as

$$\hat{var}(\hat{\psi}_n(x,y)) \quad = \quad \frac{\hat{\psi}_n(x,y)(1-\hat{\psi}_n(x,y))}{n-1} \tag{5.3}$$

This estimation makes it possible to estimate the quality of the soft shadow solution after taking n samples. The standard error derived from this estimate is later used to switch to other methods when the statistical approach has an expected error that is deemed too high. Table 5.2.1 shows these formulas applied to some real-world τ_i and increasing sample sizes. Please note that although the standard error decreases when the sample size is increased, the estimator for the standard error is not guaranteed to do so.

To create smooth penumbrae for the 256 attenuation levels available with $8bit$ color channels, at least 256 shadow maps are required, so $n \geq 256$. However, just having a smooth penumbra is not enough to completely solve the area visibility problem associated with light sources. Arbitrarily complex visibility events can make it necessary to take even more samples, which would make it a hard task to achieve real-time performance. A solution for this problem is to employ Temporal Coherence, which is explained in Section (5.2.2).

n	1	2	3	4	5	6	7
τ_n	1	0	1	1	1	0	1
$\hat{\psi}_n$	1.00	0.50	0.67	0.75	0.80	0.67	0.71
$\hat{var}(\hat{\psi}_n)$	0	0.25	0.11	0.06	0.04	0.04	0.03
$\hat{s}$	0	0.50	0.33	0.25	0.20	0.21	0.18

Tab. 5.1: Evaluation of the presented formulas for one fragment. Increasing the sample size generally reduces variance and standard error, $\hat{s} = \sqrt{\hat{var}}$.

5.2.2 Temporal Coherence

Since the calculation of hundreds of shadow maps per frame is very costly and makes real-time frame rates nearly impossible, only a single shadow map is created and evaluated per frame. Using the temporal coherence of the current frame with the previous ones makes it possible to solve the formulas from Section 5.2.1 iteratively.

The temporal coherence method as described in [SJW07] improves the resolution of standard hard shadow maps. It is based on the assumption that most screen space fragments stay the same from frame to frame. By reprojecting (to account for camera movement) fragments from the new frame into the old and comparing their respective depths, it is possible to detect whether a fragment has also been present in the previous frame. If the compared depth difference is smaller than a predefined ϵ, the two fragments are considered equal and therefore fragment data from the previous frame is reused.

$$|depth_{current}(x, y) - depth_{previous}(x, y)| < \epsilon \tag{5.4}$$

Otherwise, the fragment was not present in the last frame and is therefore considered to be disoccluded, and no previous data for this fragment is available.

In the new soft shadow approach, this technique can be re-used to solve a very similar problem: it is necessary to compute $\hat{\psi}_n(x, y)$ iteratively for each frame from the information saved from previous frames together with the information gathered for the current frame. This is done by keeping $\rho_n(x, y) := \sum_{i=1}^{n} \tau_i(x, y)$ from the previous frame. $\rho_n(x, y)$ stores all the shadow map tests already performed for the n previously calculated shadow maps. The sample size n is equal to the number of shadow map test which have already been performed for this fragment.

If this fragment was occluded in the last frame, $n = 0$ and $\rho_n(x, y) = 0$, because no previous information is available. Therefore n can be different for each screen space fragment, which is denoted by writing $\eta(x, y)$ instead of n. $\hat{\psi}_{current}(x, y)$ for the fragment at screen space position (x, y) for the current frame

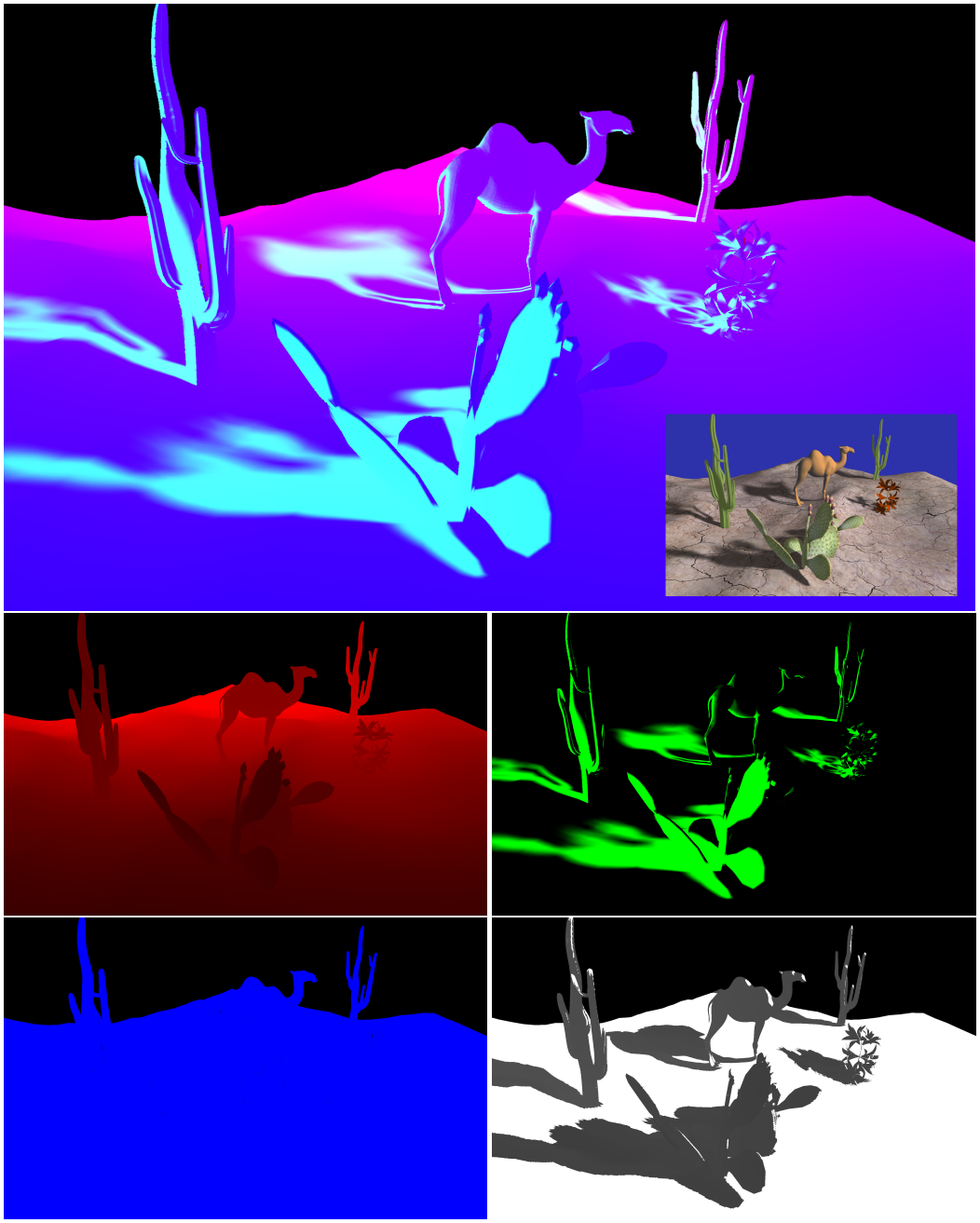

Fig. 5.3: *Top*: A visualization of the shadow buffer belonging to the scene displayed in the lower right corner. *Bottom*: The different channels the shadow buffer consists of. In this example, the red channel is used to store the depth, the green one to count the *successful* shadow tests, and the blue one to store the number of *all* performed shadow tests. The α-channel is used to store the penumbra estimation value as described in Section 5.2.3. *Note*: The buffer values have been adapted to fit into RGB color space.

can now be calculated as

$$\hat{\psi}_{current}(x,y) = \frac{\tau_{current}(x,y) + \rho_n(x,y)}{\eta(x,y)+1}. \tag{5.5}$$

Note that it is only necessary to access $\eta(x,y)$ (the sample size of this fragment from the previous frame), $\rho_n(x,y)$ (sum of the shadow map tests up to the previous frame) and the current shadow map. This is done by storing the updated sample size $\eta(x,y)+1$ and $\tau_{current}(x,y)+\rho_n(x,y)$ for every fragment into a screen sized off-screen buffer called the *shadow buffer*, which can then be accessed and used in the next frame. Using this technique, the above formula can be evaluated very quickly in a fragment shader and real-time frame rates can easily be achieved. Note that the depth of each fragment has to be stored as well in order to be able to evaluate the test in Equation 5.4. See Figure 5.3 for a visualization example of the shadow buffer.

Problems occur whenever a fragment is newly disoccluded due to camera movement: Especially for the first few shadow tests, the soft shadow estimation error is usually quite large. This can be seen in detail by looking at the example in Table 5.2.1: At $n = 2$, the standard error is 0.5, which means the real ψ is probably inside of 0.5 ± 0.5 - definitely a very bad estimate for $\hat{\psi}_2$.

Another closely connected problem that worsens the situation even more is caused by discontinuity in time. The difference between $\hat{\psi}_n$ and $\hat{\psi}_{n+1}$ can be quite large for the first few samples. For instance, $\left|\hat{\psi}_1 - \hat{\psi}_2\right| = 0.5$, meaning that the soft shadow results are 128 attenuation values apart, and causing very noticeable flickering artifacts due to the large alteration of the soft shadow value from frame to frame. In Section 5.2.3, it is therefore suggested to use spatial filtering to avoid this.

5.2.3 Spatial Filtering

So by employing Temporal Coherence and random sampling of the area light source, it is possible to simulate physically correct soft shadows with real time performance. Still, some drawbacks remain:

- The fragment needs to be visible for many frames in order to obtain a good estimation of $\psi(x,y)$ with $\hat{\psi}_n(x,y)$.
- Whenever a fragment is disoccluded, $\hat{\psi}_n(x,y)$ has a large standard error and may change drastically from one frame to the next. This causes visually noticeable flickering of these fragments.

To account for these problems, spatial filtering similar to the techniques used in many single sample soft shadow approaches is used if $\sqrt{\hat{var}(\hat{\psi}_n(x,y))}$ is larger

than a predefined threshold. In such a situation, the error introduced by a simple single sample soft shadow approach is probably smaller than the method with Temporal Coherence. In order to avoid sudden changes in the shadow caused by hard switching between these orthogonal methods (the one samples in time, the other in space), the two approaches are softly blended using the standard error as blending weight.

In the following Sections a fast single sample approach is introduced, which uses a simple penumbra estimation (Section 5.2.3) and the soft shadowing information stored in the neighboring fragments of the shadow buffer (Section 5.2.3). The two approaches are then finally blended together (Section 5.2.4). Please note that other single sample soft shadow approaches could also be modified to work together with the basic Temporal Coherence approach.

Penumbra width estimation

The penumbra width estimation is based on the formula also used in the PCSS algorithm [Fer05], in which the assumption is made that blocker, receiver and light source lie in parallel planes. This makes the calculation of the $penumbra_{width}$ fast and simple:

$$penumbra_{width} = \frac{receiver_{depth} - \overline{blocker_{depth}}}{\overline{blocker_{depth}}} light_{size}, \tag{5.6}$$

where $receiver_{depth}$ is the depth of the current fragment and $light_{size}$ is the size of the light source. The calculation of the average blocker depth $\overline{blocker_{depth}}$ is one of the costliest steps in the PCSS algorithm, as it is necessary to search the neighboring k texels in the shadow map and average these k depths for each fragment each frame. By exploiting temporal coherence again, this costly search can be avoided for most of the time: When a fragment is disoccluded, the blocker search is performed as described above, and the average blocker depth is stored in the shadow buffer and can be accessed in the next frame. There it is used to reconstruct the sum of the depths sampled so far:

$$\sum_{i=1}^{n-1} depth_i = \overline{blocker_{depth}}(n-1+k), \tag{5.7}$$

where $\overline{blocker_{depth}}$ is the stored average blocker depth from the last frame, k is the number of neighborhood samples searched in the first step and n is the current sample size of this fragment. The new blocker depth $blocker_{new_{depth}}$ from the shadow map of the current frame is added, and the new average $\overline{blocker_{depth}}_{new}$ is calculated (and again stored in the shadow buffer):

$$\overline{blocker_{depth}}_{new} = \frac{blocker_{new_{depth}} + \sum_{i=1}^{n-1} depth_i}{n+k}. \tag{5.8}$$

Neighborhood sampling

In the PCSS algorithm [Fer05], the estimated penumbra width value (as described in Section 5.2.3) modifies the kernel size of the PCF filter which is used during shadow map sampling. A larger penumbra estimation value leads to a larger filter size, which results in a wider and softer penumbra. However, this approach does not work together with Temporal Coherence: Since the shadow map is created from a different sampling point on the area light source each frame, using a PCF filter would lead to inconsistent or "jumping" shadows.

Instead, it is assumed that neighboring fragments in the screen space shadow buffer will probably have a similar $\hat{\psi}_n(x, y)$, as long as the difference between their depths is smaller than a predefined ϵ. This is of course not true for hard shadow borders, but they have a tiny penumbra width estimation and therefore their neighborhood can be disregarded anyway. Thus, a filter in screen space is applied by sampling the shadow values in the shadow buffer using a Poisson disk centered at the current fragment with a fixed sample size (usually 16 samples). The penumbra width estimation is used to modify the sample spacing, making the penumbrae softer the farther the blocker is away from the shadowed fragment. The looked up screen space values are already in post perspective space, so perspective foreshortening as well as the camera aspect (which changes the Poisson disk into an ellipse) have to be taken into account in the calculation of the Poisson disk size by noting

$$scale_{x_{proj}} = penumbra_{width_{eye}} \frac{near}{frag_{depth_{eye}}} aspect \tag{5.9}$$

$$scale_{y_{proj}} = penumbra_{width_{eye}} \frac{near}{frag_{depth_{eye}}} \tag{5.10}$$

where $scale_{(x|y)_{proj}}$ are the x and y scale factors to apply to the Poisson disk in projective space, $penumbra_{width_{eye}}$ is the penumbra width estimation in eye space and $frag_{depth_{eye}}$ is the fragment's depth in eye space. $near$ is the near plane distance and $aspect$ is the aspect of camera height to width, which depends on the camera model and could also be related to the field of view of the camera.

The final Poisson deltas which are added to the current fragment coordinates are then given by

$$delta_{i_x} = P_{i_x} scale_{x_{proj}} \tag{5.11}$$

$$delta_{i_y} = P_{i_y} scale_{y_{proj}}, \tag{5.12}$$

where $P_{i_{(x|y)}}$ is the ith Poisson sample's x or y coordinate. The average of the soft shadow values $\hat{\psi}_{current}(x + delta_{i_x}, y + delta_{i_y})$ is computed from all samples that pass the discontinuity test (i.e. their depth values are nearly the same) and taken

as the soft shadow value $\overline{\psi}(x,y)$ for the current fragment. $\overline{\psi}(x,y)$ is not stored in the shadow buffer and reused in the next frame, since every neighbor will then have an updated $\hat{\psi}_{current}$.

5.2.4 Blending

Apart from the Temporal Coherence method that samples the light source and generates correct soft shadows if the sample size is high enough, a single sample approach that generates perceptually plausible shadows is now available. Depending on the standard error of the variance estimator in Equation 5.3, the one or the other approach is to be preferred. Note that this standard error can be wrong with high probability if the sample size is below a very low (10-16 samples) limit, so solely the single sample approach is used for $n < 16$.

Two error bounds are introduced: A maximal error bound s_{max}, at which the sampling approach is started to be used, and a minimal error bound $s_{min} < s_{max}$, at which the single sample approach is stopped to be used. In between these error bounds, the two approaches are blended by using

$$w = \frac{s_{max} - s(x,y)}{s_{max} - s_{min}} \tag{5.13}$$

$$frag_{shadow}(x,y) = \overline{\psi}(x,y)(1-w) + \hat{\psi}(x,y)w \tag{5.14}$$

where $s(x,y)$ is the current standard error of $\hat{\psi}(x,y)$, and $frag_{shadow}(x,y)$ is the soft shadow result for the current fragment to be used in the shading of the fragment. A $s(x,y) = 1/16$ for $8Bit$ fragment color channels would for instance mean that the result of $\hat{\psi}(x,y)$ is expected to be within ± 16 attenuation levels of the correct result. If $s_{max} = 1/16$, solely the single sample approach would be used with this error. If on the other hand $s_{max} = 1/8$ and $s_{min} = 1/32$, the two approaches would be blended using $1/3$ of $\overline{\psi}(x,y)$ and $2/3$ of $\hat{\psi}(x,y)$.

Although s_{min} and s_{max} can be chosen freely and can be constant, one could also use light and material parameters to guide the selection. The calculated soft shadow value $frag_{shadow}(x,y)$ is a darkening factor that is applied $(1 - frag_{shadow}(x,y))$ to the diffuse color of the light source. So other factors like the diffuse color, the ambient light, or the texture and material color used for shading the fragment limit the attenuation range that can be influenced by $frag_{shadow}(x,y)$. In such circumstances, higher s_{min} and s_{max} can therefore be chosen for this fragment.

5.2.5 Moving Objects

Dynamic scene objects do not work very well together with Temporal Coherence techniques: As a result of the object movement, disocclusions happen very fre-

Fig. 5.4: Some noise artifacts can appear at the shadow borders of moving objects.

quently and increase the standard error in these areas. It is therefore necessary to tag them in the shadow map as well as in the scene in order to be able to treat them separately.

Therefore, in the depth pass, the shadow map depth values caused by moving objects are stored with a negative sign. This is possible since the shadow map depth values are always positive, and so a moving object can easily be identified in the second pass by simply checking the sign (this can be during the shadow map test, when the absolute depth value needs to be retrieved for every screen-space fragment anyway).

Artifacts caused by moving objects are most likely to occur whenever an object starts or stops being an occluder for the current fragment, since the illumination of this fragment has to change drastically (e.g. a completely unoccluded fragment, on which several thousand sampling tests have been performed, could be completely occluded within three frames due to a fast moving blocker object - and the shadow value should be adapted just as fast.). Without special treatment, newly occluded areas don't darken fast enough, and newly unoccluded areas suffer from shadows that are "dragged behind" the caster.

Therefore, shadows which have been cast by moving objects in previous frames have to be identifiable. This is done by storing the information in the shadow buffer, making it easily accessible in the next frame. Again, a negative sign can be used on any of the signless values of the buffer (e.g. the eye space depth used for the depth comparison) for tagging.

For these detected fragments, the dynamic shadow needs to be removed by gradually decrementing the sample size $\eta(x, y)$ stored at this fragment and changing the $\rho_n(x, y)$ accordingly with

$$r(x, y) = \frac{\rho_n(x, y)}{\eta(x, y)} \tag{5.15}$$

$$\eta'(x, y) = \eta(x, y) - \Delta \tag{5.16}$$

$$\rho'_n(x, y) = \eta'(x, y) r(x, y), \tag{5.17}$$

where Δ is the decrementing factor and $\eta'(x, y)$ and $\rho'_n(x, y)$ are the new values for $\eta(x, y)$ and $\rho_n(x, y)$. This is done repeatedly each frame until $\eta'(x, y) = 0$. This way, shadows that are dragged behind the object can be removed at the expense of some noise artifacts (see Figure 5.4).

5.3 Implementation

Our new approach was developed and tested using our own DirectX 10 framework (under Windows Vista), in which several state-of-the-art shadow algorithms were implemented as well, allowing us to compare our algorithm directly to other methods in terms of speed and shadow quality (see Figure 5.5 for an application screenshot). We performed tests on several different hardware configurations, but the test images and benchmarks were all taken on a system with an Intel Core 2 Duo E6600 CPU and a NVIDIA 280GTX GPU, using shadow maps that were generated with standard uniform shadow mapping with a resolution of 1024^2.

For the shadow map, a 32bit floating point texture is used to store the values computed in the first pass. We store the depth linearly, as was proposed in the PCSS paper [Fer05]. Selecting the sample position on the area light is done using a Halton sequence, but other quasi-random sequences showed similar behavior.

In the second pass, the multiple render target functionality is used to render into the shadow buffer and into an 8bit RGB buffer at the same time. The shadow buffer consists of a 4-channel 32bit floating point texture and is double buffered, since read and write operations on the same render target are not possible on current graphics hardware. This results in an additional memory requirement of $840 \times 630 \times 4 \times 4 \approx 8MB$. It contains $\rho_n(x, y)$, $\eta(x, y)$, the linear depth of the fragment and the $\overline{blocker_{depth}}$.

Fig. 5.5: A screenshot of our test application used to develop and compare our new algorithm (Windows Vista, DirectX 10).

Finally, the RGB buffer is rendered into the frame buffer in a third render pass. By choosing the resolution of the shadow buffer about 5% larger than the frame buffer, the shadow quality on frame buffer border texels can be improved, since they can then use more and better converged shadow data from their neighboring texels during the spatial filtering step. This means, for example, that if a frame buffer size of 800x600 is chosen, the shadow buffer size would be 840x640.

We found that the test depicted in Equation 5.4 used in [SJW07] for determining if fragment data is available from previous frames is numerically unstable, when used in conjunction with linear depth values. We therefore use

$$\left| 1 - \frac{depth_{current_{xy}}}{depth_{previous_{xy}}} \right| < \epsilon \tag{5.18}$$

instead in the reprojection step and in the discontinuity test when sampling the neighborhood of a fragment, resulting in an error threshold that is relative to the distance.

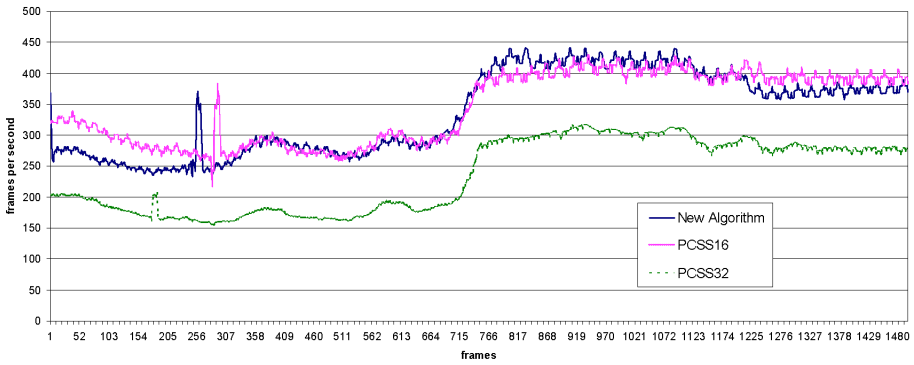

Fig. 5.6: A sample walkthrough in one of our test scenes for our new method, PCSS using 16/16 samples for blocker/PCF lookup and PCSS with 32/32 samples.

Note: In Appendix A, the shader source code (DirectX 10 HLSL, shader model 4.0) used in our framework can be found.

5.4 Results

Our goal was to create a physically accurate soft shadow algorithm which runs at a similar frame-rate as single-sample approaches like for example PCSS. We compared our algorithm to a fast PCSS version using only 16 texture lookups for the blocker search and 16 texture lookups for the PCF kernel. Figure 5.6 shows the performance in frames per second for a typical walkthrough of our test scene, using a frame buffer size of 800x600 pixels.

In cases of many disocclusions, our algorithm is slower, as it has to perform the blocker search there (which PCSS has to perform each frame): Due to the higher complexity (more ifs) than the PCSS shader, the performance of our method can be worse compared to PCSS in such circumstances. Still, the used PCSS16 *always* performs 16+16 lookups, while our shader usually only performs one shadow map lookup and one shadow buffer lookup every frame. 16 shadow map lookups only occur in the case of a disocclusion, and 16 shadow buffer lookups are necessary for the neighborhood filter for every fragment where the spatial filtering approach is active.

Moreover, the shadow quality of our approach is significantly better (see Figure 5.7): Apart from being physically accurate, our method does not suffer from typical single-sample artifacts (like for example artifacts caused by overlapping occluders and band artifacts caused by big penumbras).

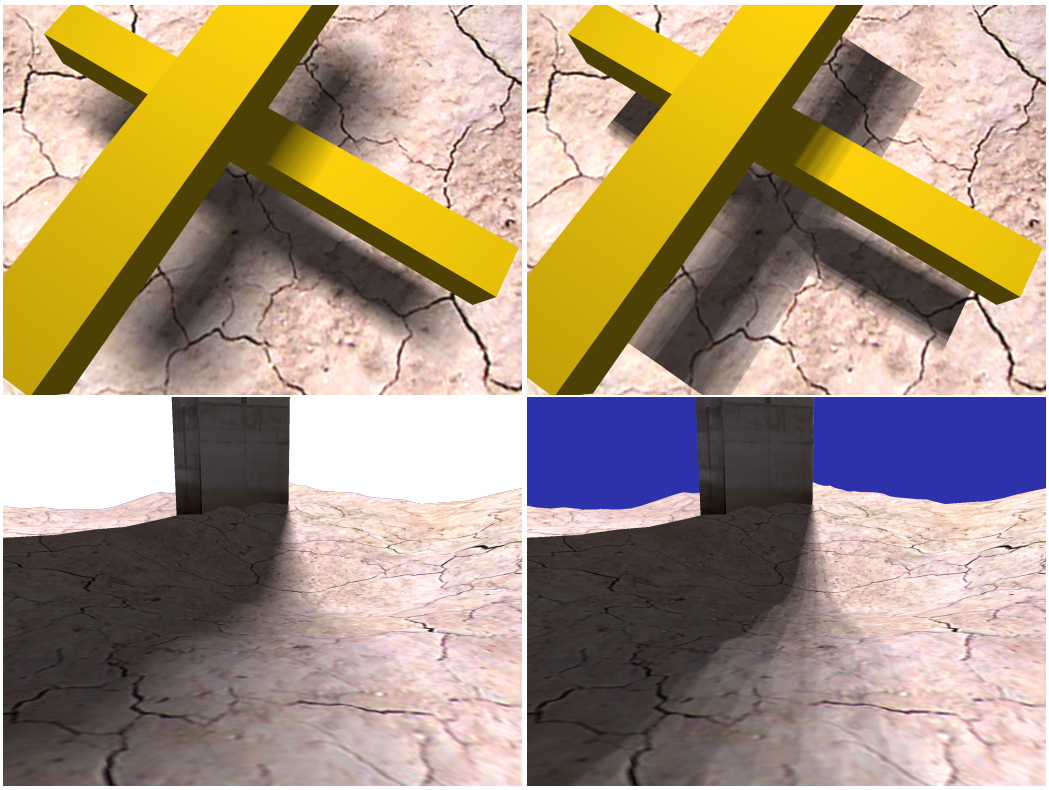

Fig. 5.7: *Left side*: Our Method. *Right Side*: PCSS 16/16 [Fer05]. Overlapping occluders (upper row) and bands in big penumbras (lower row) are known problem cases for single sample approaches.

5.4.1 Limitations

Due to the use of Temporal Coherence methods, our approach has difficulties with moving objects and moving light sources, as disocclusions are very likely to happen. Even though moving objects can be handled, their shadows suffer from artifacts. Moving light sources cannot be handled yet. A solution to these problems might be an approach where older light source samples are weighted less, and where shadow fragments are assigned an "age factor".

5.5 Summary

In this chapter, a fast, image-based real-time soft shadow approach, which generates physically accurate soft shadows and uses Temporal Coherence, was presented. Spatial filtering is applied whenever temporal reprojection is insufficient, allowing perceptually plausible results also on recently disoccluded fragments.

We tested the method in our framework, and the results show that it is as fast as the fastest single sample approaches, but at the same time provides better image quality.

I don't need a friend who changes when I change and who nods when I nod; my shadow does that much better.

(*Plutarch)*

Chapter 6

Summary

This thesis gave a detailed overview of the field of real-time soft shadow algorithms. Chapter 2 introduced the reader to the basic concepts and methods: Apart from an explanation of the terms *shadow*, *real-time* and *rendering*, it was shown why the use of shadows in computer-generated images is important at all. The most widely used shadow algorithms, namely *shadow mapping* and *shadow volumes*, were described as well.

Soft shadows, which were introduced in Chapter 3, are far more complex to calculate than hard shadows. They are cast by area light sources, making the calculation much more complicated, as the point-to-point visibility problem is expanded to a three-dimensional visibility problem. Still, the increased effort is definitely worthwhile: Since point light sources do only exist in theory, we are used to area light sources and the resulting soft shadow boundaries, and perceive them as much more "realistic".

Since the generation of physically exact soft shadows is a very complicated and computationally expensive task, none of the algorithms presented in Chapter 4 is able to generate correct soft shadows in real-time for an arbitrary scene. It is therefore necessary to make trade-offs between speed and accuracy, and use fake approaches which use a simplified, but faster way to calculate visually plausible results. Of course, these simplified visibility calculations do not provide exact solutions and do sometimes generate artifacts, but in many cases (especially in dynamic scenes), they are absolutely sufficient. Which approach is to prefer, depends heavily on the requirements of the application or the game itself.

Finally, we presented our own algorithm in Chapter 5: By exploiting *Temporal Coherence*, it is possible to generate physically accurate soft shadows in real-time. The main idea is to sample the area light source randomly, but only once per frame, making the method run as fast as single sample fake soft shadow approaches. The shadow results are stored in a *screen space shadow buffer*, from where they can be re-used in the next frame. In cases when there is only insufficient shadowing information available for a fragment, additional spatial filtering is applied, allowing perceptually plausible results also on recently disoccluded

fragments (see Figure 6.1 for a screenshot).

Fig. 6.1: Screenshot of a test scene rendered with our soft shadow algorithm in our shadow mapping test framework (DirectX 10).

Appendix A

Shader source code

In this Appendix, the shader code used in our test framework (DirectX 10 HLSL, shader model 4.0) is shown.

A.1 Depth pass shader code

```
// Variables
matrix View;
matrix Projection;
matrix World;
bool bMovingObject;

// Structures
struct VS_INPUT
{
        float3 Pos      : POSITION;             //position
};

struct PS_INPUT
{
        float4 Pos      : SV_POSITION;          //position
        float Tex       : TEXCOORD;             //texture coordinates
};

// Vertex Shader
PS_INPUT VS( VS_INPUT input )
{
        PS_INPUT output = (PS_INPUT)0;
        output.Pos = mul( float4(input.Pos,1), World );
        output.Pos = mul( output.Pos, View );
        output.Pos = mul( output.Pos, Projection );
        output.Tex = output.Pos.z;

        return output;
}

// Pixel Shader
float PS( PS_INPUT input) : SV_Target
{
        return input.Tex.x*(bMovingObject ? -1.0f:1.0f);
}
```

A.2 Shadowing pass shader code

```
// Variables
Texture2D g_txDiffuse;          // Object texture
Texture2D g_txSM;               // Shadow Map
Texture2D g_txHistoryBuffer;    // History Buffer Texture

matrix World;                   // World Matrix
matrix View;                    // View Matrix
```

```
8  matrix Projection;              // Projection Matrix
9  matrix LastView;                // Last View Matrix
10 matrix LastProjection;          // Last Projection Matrix
11 matrix LightView;               // Light View Matrix
12 matrix LightProjection;         // Light Projection Matrix
13
14 float2 vLightDimensions;        // Area Light x/y Dimensions
15 float2 vSMtexelSize;            // Shadow Map Texel Size (1/SMsize)
16 float2 vAspectRatio;            // Screen aspect ratio (y=1, x=resolution_y/resolution_x)
17
18 //Definitions
19 const float EPSILON_SM_BIAS = 0.03f;    //Shadow Mapping Bias
20 const float EPSILON_DEPTH = 0.001f;     //Depth Comparison Epsilon (for reprojection)
21 const int BLOCKER_SEARCH_LOOKUP = 16;   //Number of lookups for blocker search
22 const int PCF_NUM_SAMPLES = 16;         //Number of samples for spatial filtering
23
24 //Poisson disk:
25 cbuffer POISSON_DISKS
26 {
27         float2 poissonDisk[16] = {
28                 float2( -0.94201624, -0.39906216 ),
29                 float2( 0.94558609, -0.76890725 ),
30                 float2( -0.094184101, -0.92938870 ),
31                 float2( 0.34495938, 0.29387760 ),
32                 float2( -0.91588581, 0.45771432 ),
33                 float2( -0.81544232, -0.87912464 ),
34                 float2( -0.38277543, 0.27676845 ),
35                 float2( 0.97484398, 0.75648379 ),
36                 float2( 0.44323325, -0.97511554 ),
37                 float2( 0.53742981, -0.47373420 ),
38                 float2( -0.26496911, -0.41893023 ),
39                 float2( 0.79197514, 0.19090188 ),
40                 float2( -0.24188840, 0.99706507 ),
41                 float2( -0.81409955, 0.91437590 ),
42                 float2( 0.19984126, 0.78641367 ),
43                 float2( 0.14383161, -0.14100790 )
44         };
45 };
46
47 // Structures
48 struct VS_INPUT
49 {
50         float3 Pos      : POSITION;         //position
51         float3 Norm     : NORMAL;           //normals
52         float2 Tex      : TEXCOORD0;        //texture coordinate
53 };
54
55 struct PS_INPUT
56 {
57         float4 Pos      : SV_POSITION;      //position
58         float3 Norm     : NORMAL;           //normals
59         float4 wPos     : TEXCOORD0;        //world space position
60         float4 LightPos : TEXCOORD1;        //fragment position in light space
61         float4 BufferPos: TEXCOORD2;        //fragment position in shadow buffer
62         float2 Tex      : TEXCOORD3;        //texture coordinates
63         float Depth     : TEXCOORD4;        //fragment depth
64 };
65
66 struct PS_OUTPUT_2RT
67 {
68         float4 Col0     : COLOR0;           //render target 1 (RGB)
69         float4 Col1     : COLOR1;           //render target 2 (shadow buffer)
70 };
71
72 // Texture Samplers
73 SamplerState samLinear
74 {
75         Filter = MIN_MAG_MIP_LINEAR;
76         AddressU = Wrap;
77         AddressV = Wrap;
78 };
79
80 SamplerState samPoint
81 {
82         Filter = MIN_MAG_MIP_POINT;
83         AddressU = Clamp;
84         AddressV = Clamp;
85 };
86
87 // Vertex Shader
88 PS_INPUT VS( VS_INPUT input )
89 {
90         PS_INPUT output = (PS_INPUT)0;
```

```
91
92          //transform vertices to world space
93          output.wPos = mul( float4(input.Pos,1), World );
94
95          //transform to light projection space
96          output.LightPos = mul( output.wPos, LightView );
97          output.LightPos = mul( output.LightPos, LightProjection );
98
99          //transform to shadow buffer space
100         output.BufferPos = mul( output.wPos, LastView );
101         output.BufferPos = mul( output.BufferPos, LastProjection );
102
103         //transform to camera projection space
104         output.Pos = mul( output.wPos, View );
105         output.Pos = mul( output.Pos, Projection );
106
107         //transform normals to world space
108         output.Norm = mul( input.Norm, World );
109
110         //propagate tex coords
111         output.Tex = input.Tex;
112
113         //save Depth
114         output.Depth = output.Pos.z;
115
116         return output;
117 }
118
119 //Helper function checking validity of texture coordinates
120 bool outsideOld(float2 _Tex) {
121         return any(bool2(_Tex.x < 0.0f, _Tex.y < 0.0f))
122                 || any(bool2(_Tex.x > 1.0f, _Tex.y > 1.0f));
123 }
124
125 //Helper function performing the shadow test
126 float shadowTest(const float depthSM_, const float fragmentDepth_) {
127         return depthSM_+EPSILON_SM_BIAS < fragmentDepth_ ? 0.0 : 1.0;
128 }
129
130 //Helper function calculating the SM coordinates
131 float2 texSpace(const float4 input_) {
132         float2 output = 0.5 * input_.xy / input_.w + float2( 0.5, 0.5 );
133         output.y = 1.0f - output.y;
134         return output;
135 }
136
137 //Helper function doing the blocker search
138 float findBlockerSM(const float2 pos_TS_, const float fragmentDepth_) {
139         float sum = 0;
140         float cnt = 0;
141
142         for(int i = 0; i < BLOCKER_SEARCH_LOOKUP; ++i) {
143                 const float2 offset = poissonDisk[i] * vSMtexelSize*7.0f;
144                 const float2 pos = pos_TS_ + offset;
145
146                 const float Depth = abs(g_txSM.SampleLevel( samPoint, pos, 0).x);
147                 if(0.5 > shadowTest(Depth,fragmentDepth_)) {
148                         sum += Depth;
149                         cnt++;
150                 }
151         }
152         return (cnt > 0) ? sum / cnt : -1000000.0f;
153 }
154
155 //Helper function doing the screen space spatial filtering
156 float neighborhoodFilter(const float2 uv, const float2 filterRadiusUV,
157                         const float currentDepth_,
158                         const float currentCnt_) {
159         float sum = 0;
160         float cnt = 0;
161
162         for(int i = 0; i < PCF_NUM_SAMPLES; ++i ) {
163                 const float2 offset = poissonDisk[i] * filterRadiusUV;
164
165                 if(!outsideOld(uv + offset)) {
166                         const float3 data = g_txHistoryBuffer.SampleLevel( samPoint, uv + offset, 0).xyz;
167                         const float count = data.z;
168
169                         const float depth = abs(data.x);
170                         if(abs(1-currentDepth_/depth) < 15*EPSILON_DEPTH) {
171                                 sum += data.y/count;
172                                 ++cnt;
173                         }
```

```
174                     }
175             }
176
177             if(cnt>0) {
178                     return sum / cnt;
179             } else {
180                     return -1;
181             }
182 }
183
184 PS_OUTPUT_2RT PSSoftShadows( PS_INPUT input) : SV_Target
185 {
186             PS_OUTPUT_2RT output = (PS_OUTPUT_2RT)0;
187
188             bool bCurrentShadowFromMovingObject = false;     //Blocker does not move
189             bool bPreviousShadowFromMovingObject = false;    //Blocker DID not move
190             bool inDisoccludedRegion = true;                 //Fragment has just been disoccluded
191
192             float ShadowAmount = 0.0f;                       //Shadow amount
193             float cnt = 1;                                   //Number of shadow tests
194
195             //Average Blocker Depth, needed for penumbra estimation.
196             // a negative value indicates that no blocker has been found
197             // note: this has to be extremely low, since a linear lookup takes place in the
198             // histoy buffer, and the value gets interpolated! it can become positive, leading
199             // to a wrong penumbra estimation. a very low value can extremely reduce this effect!
200             float avgBlockerDepth = -1000000.0f;
201
202             //shadow sampling coordinates
203             const float2 smCoord = texSpace(input.LightPos);
204             //Linear depth of current fragment in light space
205             const float fragmentDepth = input.LightPos.z;
206
207             if(!outsideOld(smCoord)) {
208                     //sample depth in shadow map
209                     const float Depth = g_txSM.SampleLevel( samPoint, smCoord, 0).x;
210                     //get absolute value
211                     const float absDepth = abs(Depth);
212
213                     //make shadow test
214                     ShadowAmount = 1.0-shadowTest(absDepth,fragmentDepth);
215
216                     //if the current fragment is in shadow
217                     if(ShadowAmount == 1.0) {
218                             //get new depth as estimate for avgBlockerDepth
219                             avgBlockerDepth = absDepth;
220                             //check if the blocker is moving
221                             bCurrentShadowFromMovingObject = Depth < 0.0;
222                     }
223             }
224
225             //history buffer sampling coordinates:
226             const float2 HisBuffTexC = texSpace(input.BufferPos);
227
228             //check if the pixel is inside the history buffer:
229             if(!outsideOld(HisBuffTexC)) {
230                     //inside of old data -> we can check for depth delta
231
232                     //get the history buffer info:
233                     float4 oldData = g_txHistoryBuffer.Sample( samLinear, HisBuffTexC);
234                     //Moving Object Shadow identification
235                     if(oldData.x < 0) bPreviousShadowFromMovingObject = true;
236
237                     const float oldDepth = abs(oldData.x);
238                     const float oldAvgBlockerDepth = oldData.w;
239                     float oldSum = oldData.y;
240                     float oldCount = oldData.z;
241
242                     //check if the pixel was in shadow of a moving object and is outside of it now:
243                     if(!bCurrentShadowFromMovingObject && bPreviousShadowFromMovingObject) {
244                             //drastically reduce the shadow
245                             const float ratio = oldSum/oldCount;
246                             const float delta = 1.0;
247                             if(oldCount > 6.0f) {
248                                     oldCount = delta;
249                             } else {
250                                     oldCount = oldCount-delta;
251                             }
252
253                             oldSum = ratio * oldCount;
254
255                             if(oldCount <= delta) {
256                                     oldSum = 0;
```

```
257                                 oldCount = 0;
258                                 bPreviousShadowFromMovingObject = false;
259                         }
260                 }
261
262                 //check if depths are alike, otherwise we have a very different sample
263                 if(abs(1-input.BufferPos.z/oldDepth) < EPSILON_DEPTH && !bCurrentShadowFromMovingObject)
264                 {
265                         //calculate the new shadow amount:
266                         ShadowAmount = oldSum + ShadowAmount;
267                         cnt = oldCount + 1;
268                         inDisoccludedRegion = false;
269
270                         //Average Blocker Depth
271                         if(oldAvgBlockerDepth >= 0.0f) {
272                                 if(avgBlockerDepth >= 0.0f) {
273                                         float sum = oldAvgBlockerDepth*(cnt-1+BLOCKER_SEARCH_LOOKUP);
274                                         sum += avgBlockerDepth;
275                                         avgBlockerDepth = sum/(cnt+BLOCKER_SEARCH_LOOKUP);
276                                 } else {
277                                         avgBlockerDepth = oldAvgBlockerDepth;
278                                 }
279                         }
280                 }
281         }
282
283         //for newly dissoccluded fragments estimate avg blocker depth
284         if(cnt < 1.5) avgBlockerDepth = findBlockerSM(smCoord,fragmentDepth);
285
286         //calculate the soft shadow value generated with Temporal Reprojection:
287         float softShadow = ShadowAmount / cnt;
288
289         //if little samples or large error just do aproximation
290         const float errorMin = 1.0/50.0;
291         const float errorMax = 1.0/5.0;
292         const float error = cnt <= 1.5 ? 1.0 : sqrt(softShadow*(1-softShadow)/(cnt-1));
293         if((cnt <= BLOCKER_SEARCH_LOOKUP || error >= errorMin) && (avgBlockerDepth > 0)) {
294                 //penumbra estimation like in PCSS, but with the
295                 //average occluder depth from the history buffer:
296                 const float penumbraEstimation = ((fragmentDepth - avgBlockerDepth) / avgBlockerDepth) *
297                                 vLightDimensions[0];
298
299                 //do spatial filtering in the shadow buffer (screen space) using the penumbra estimation:
300                 const float multiplicator = (0.1f/input.Depth);
301                 const float blurredShadow = neighborhoodFilter(HisBuffTexC,
302                                 vAspectRatio * multiplicator * penumbraEstimation, input.Depth,cnt);
303
304                 if(blurredShadow > 0.0f) {
305                         if(inDisoccludedRegion) {
306                                 ShadowAmount = (blurredShadow + softShadow) * 8;
307                                 cnt = 16;
308                                 softShadow = ShadowAmount / cnt;
309                         } else if (error > errorMax || cnt <= BLOCKER_SEARCH_LOOKUP) {
310                                 //very big error -> only use aproximation
311                                 softShadow = blurredShadow;
312                         } else {
313                                 //blend
314                                 const float weight = (errorMax-error)/(errorMax-errorMin);
315                                 softShadow = blurredShadow * (1-weight) + softShadow * weight;
316                         }
317                 }
318         }
319
320         //store a moving object in the shadow buffer by giving a value a negative sign later
321         const float encodeMovingObject = (bCurrentShadowFromMovingObject
322                                 || bPreviousShadowFromMovingObject ? -1.0f : 1.0f);
323
324         //store depth (with encoded moving object), nr of positive shadow tests,
325         //nr of all shadow tests, and average blocker depth in shadow buffer
326         output.Col1 = float4(encodeMovingObject*input.Depth, ShadowAmount, cnt, avgBlockerDepth);
327
328         // vLight is the unit vector from the light to this pixel
329         const float3 vLight = normalize( float3( vLightPos.xyz - input.wPos.xyz ) );
330         //per pixel lighting - use whatever function you want...
331         const float4 DiffuseColor = DiffusePerPixelLighting(vLight, input.Norm);
332         //set the output color depending on lighting and shadowing:
333         output.Col0.rgb = DiffuseColor.rgb * (1-softShadow) + vLightColor * vMaterialAmbient;
334         //add texture information (if available):
335         if(bUseTexture) output.Col0 *= g_txDiffuse.Sample( samLinear, input.Tex );
336
337         return output;
338 }
```

List of Figures

List of Tables

Bibliography

[AAM03] Ulf Assarsson and Tomas Akenine-Möller. A geometry-based soft shadow volume algorithm using graphics hardware. *ACM Trans. Graph.*, 22(3):511–520, 2003.

[ADM+08] Thomas Annen, Zhao Dong, Tom Mertens, Philippe Bekaert, Hans-Peter Seidel, and Jan Kautz. Real-time, all-frequency shadows in dynamic scenes. *ACM Trans. Graph.*, 27(3):1–8, 2008.

[ADMAM03] Ulf Assarsson, Michael Dougherty, Michael Mounier, and Tomas Akenine-Möller. An optimized soft shadow volume algorithm with real-time performance. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 33–40. Eurographics Association, 2003.

[AHL+06] Lionel Atty, Nicolas Holzschuch, Marc Lapierre, Jean-Marc Hasenfratz, Chuck Hansen, and François Sillion. Soft shadow maps: Efficient sampling of light source visibility. *Computer Graphics Forum*, 25(4), dec 2006.

[AHT04] Jukka Arvo, Mika Hirvikorpi, and Joonas Tyystjärvi. Approximate soft shadows using image-space flood-fill algorithm. *Computer Graphics Forum*, TODO(TODO):TODO, 2004.

[AMA02] Tomas Akenine-Möller and Ulf Assarsson. Approximate soft shadows on arbitrary surfaces using penumbra wedges. In *Proceedings of the 13th Eurographics Workshop on Rendering*, pages 309–218. Eurographics, 2002.

[AMB+07] Thomas Annen, Tom Mertens, Philippe Bekaert, Hans-Peter Seidel, and Jan Kautz. Convolution shadow maps. In Jan Kautz and Sumanta Pattanaik, editors, *Rendering Techniques 2007: Eurographics Symposium on Rendering*, volume 18 of *Eurographics / ACM SIGGRAPH Symposium Proceedings*, pages 51–60, Grenoble, France, June 2007. Eurographics.

[AMHH08] Tomas Akenine-Möller, Eric Haines, and Natty Hoffman. *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2008.

[ARHM00] Maneesh Agrawala, Ravi Ramamoorthi, Alan Heirich, and Laurent Moll. Efficient image-based methods for rendering soft shadows. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 375–384. ACM Press/Addison-Wesley Publishing Co., 2000.

[ASK06] B. Aszdi and L. Szirmay-Kalos. Real-time soft shadows with shadow accumulation. In *Eurographics 2006 Short Presentations*, pages 53–56, 2006.

[Bli88] James F. Blinn. Me and my (fake) shadow. *IEEE Computer Graphics and Applications*, 8(1):82–86, Jan. 1988.

[BS02] Stefan Brabec and Hans-Peter Seidel. Single Sample Soft Shadows Using Depth Maps. In *Proceedings of Graphics Interface*, pages 219–228, May 2002.

[CD03] Eric Chan and Frédo Durand. Rendering fake soft shadows with smoothies. In *Proceedings of the Eurographics Symposium on Rendering*, pages 208–218. Eurographics Association, 2003.

[Cro77] Franklin C. Crow. Shadow algorithms for computer graphics. In James George, editor, *Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, volume 11, pages 242–248. ACM Press, July 1977.

[Cro84] Franklin C. Crow. Summed-area tables for texture mapping. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 207–212, New York, NY, USA, 1984. ACM.

[dB04] Willem H. de Boer. Smooth penumbra transitions with shadow maps. In *Submitted to Journal of Graphics Tools*, pages 185–195. AK Peters, 2004.

[DF94] George Drettakis and Eugene Fiume. A fast shadow algorithm for area light sources using backprojection. In *SIGGRAPH '94 Proc.*, pages 223–230, 1994. http://safran.imag.fr/Membres/George.Drettakis/pub.html.

[DL06] William Donnelly and Andrew Lauritzen. Variance shadow maps. In *In SI3D 06: Proceedings of the 2006 symposium on Interactive 3D graphics and games, ACM*, pages 161–165. Press, 2006.

[ED06] Elmar Eisemann and Xavier Décoret. Plausible image based soft shadows using occlusion textures. In Rodrigo Lima Oliveira Neto, Manuel Menezes deCarceroni, editor, *Proceedings of the Brazilian Symposium on Computer Graphics and Image Processing, 19 (SIBGRAPI)*, Conference Series. IEEE, IEEE Computer Society, 2006.

[FBP06] Vincent Forest, Loc Barthe, and Mathias Paulin. Realistic Soft Shadows by Penumbra-Wedges Blending. In *Graphics Hardware, Vienne, Autriche, 03/09/2006-04/09/2006*, pages 39–48, http://www.eg.org/, 2006. Eurographics.

[Fer05] Randima Fernando. Percentage-closer soft shadows. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Sketches*, page 35, New York, NY, USA, 2005. ACM.

[GBP06] Gael Guennebaud, Loc Barthe, and Mathias Paulin. Real-time soft shadow mapping by backprojection. In *Eurographics Symposium on Rendering (EGSR), Nicosia, Cyprus, 26/06/2006-28/06/2006*, pages 227–234, http://www.eg.org/, 2006. Eurographics.

[GBP07] Gael Guennebaud, Loc Barthe, and Mathias Paulin. High-Quality Adaptive Soft Shadow Mapping. *Computer Graphics Forum, Eurographics 2007 proceedings*, 26(3):525–534, septembre 2007.

[Hai01] Eric Haines. Soft planar shadows using plateaus. *J. Graph. Tools*, 6(1):19–27, 2001.

[HBS00] Wolfgang Heidrich, Stefan Brabec, and Hans-Peter Seidel. Soft shadow maps for linear lights. In *Proceedings of the Eurographics Workshop on Rendering Techniques 2000*, pages 269–280. Springer-Verlag, 2000.

[Her97] Michael Herf. Efficient generation of soft shadow textures. Technical Report CMU-CS-97-138, CS Dept., Carnegie Mellon U., May 1997.

[HH97] Paul S. Heckbert and Michael Herf. Simulating soft shadows with graphics hardware. Technical Report CMU-CS-97-104,

CS Dept., Carnegie Mellon U., Jan. 1997. CMU-CS-97-104, http://www.cs.cmu.edu/ ph.

[HLHS03] Jean-Marc Hasenfratz, Marc Lapierre, Nicolas Holzschuch, and François Sillion. A survey of real-time soft shadows algorithms. In *Eurographics*. Eurographics, Eurographics, 2003. State-of-the-Art Report.

[HN85] J. C. Hourcade and A. Nicolas. Algorithms for antialiased cast shadows. *Computers and Graphics*, 9(3):259–265, 1985.

[KD03] Florian Kirsch and Juergen Doellner. Real-time soft shadows using a single light sample. In *Journal of WSCG (Winter School on Computer Graphics 2003)*, page 11(1), 2003.

[Lau07] Andrew Lauritzen. Summed-area variance shadow maps. In Hubert Nguyen, editor, *GPU Gems 3*, chapter 8, pages 157–182. Addison Wesley, 2007.

[PSS98] Steve Parker, Peter Shirley, and Brian Smits. Single sample soft shadows. Technical Report UUCS-98-019, Computer Science Department, University of Utah, october 1998.

[rad05] `http://de.wikibooks.org/wiki/Datei:Blender3D_Radiosity_FromTest.jpg`, 2005.

[ray07] `http://de.wikipedia.org/wiki/Datei:Raytracing.svg`, 2007.

[RSC87] William T. Reeves, David H. Salesin, and Robert L. Cook. Rendering antialiased shadows with depth maps. volume 21, pages 283–291, July 1987.

[SAPP05] Jean-François St-Amour, Eric Paquette, and Pierre Poulin. Soft shadows from extended light sources with penumbra deep shadow maps. In *Graphics Interface 2005*, pages 105–112, May 2005.

[Sch05] Daniel Scherzer. Shadow mapping of large environments. Master's thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, 8 2005.

[SD02] Marc Stamminger and George Drettakis. Perspective shadow maps. In *Proceedings of the 29th annual conference on Computer*

graphics and interactive techniques, pages 557–562. ACM Press, 2002.

[SJW07] Daniel Scherzer, Stefan Jeschke, and Michael Wimmer. Pixel-correct shadow maps with temporal reprojection and shadow test confidence. In Jan Kautz and Sumanta Pattanaik, editors, *Rendering Techniques 2007 (Proceedings Eurographics Symposium on Rendering)*, pages 45–50. Eurographics, Eurographics Association, June 2007.

[SS98] Cyril Soler and François X. Sillion. Fast calculation of soft shadow textures using convolution. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 321–332. ACM Press, 1998.

[SS07] Michael Schwarz and Marc Stamminger. Bitmask soft shadows. *Comput. Graph. Forum*, 26(3):515–524, 2007.

[SSM09] Daniel Scherzer, Michael Schwärzler, and Oliver Mattausch. Real-time soft shadows using temporal coherence, 2009. Submitted to GI 2009 for publication.

[WH03] Chris Wyman and Charles Hansen. Penumbra maps: approximate soft shadows in real-time. In *Proceedings of the 14th Eurographics workshop on Rendering*, pages 202–207. Eurographics Association, 2003.

[Wil78] Lance Williams. Casting curved shadows on curved surfaces. *Computer Graphics (SIGGRAPH '78 Proceedings)*, 12(3):270–274, Aug. 1978.

[WSP04] Michael Wimmer, Daniel Scherzer, and Werner Purgathofer. Light space perspective shadow maps. In *Proceedings of Eurographics Symposium on Rendering 2004*, 2004.

[ZYD02] Min Tang Zhengming Ying and Jinxiang Dong. Soft shadow maps for area light by area approximation. In *10th Pacific Conference on Computer Graphics and Applications*, pages 442–443. IEEE, 2002.