Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE

Boris Zápotocký

Měkké stíny na GPU

Katedra výuky software a informatiky

Vedoucí diplomové práce: RNDr. Josef Pelikán

Studijní program: Informatika, Softwarové systémy

Contents

Název práce: Měkké stíny na GPU
Autor: Boris Zápotocký
Katedra (ústav): Katedra výuky software a informatiky
Vedoucí diplomové práce: RNDr. Josef Pelikán
E-mail vedoucího: Josef.Pelikan@mff.cuni.cz
Abstrakt: Obrovský vývoj v oblasti grafického hardware a programovatelnost grafických karet umožňují vyšší výkonnost a flexibilitu programování. Mnoho algoritmů se stalo interaktivními. Velmi důležitou roli mezi grafickými algoritmy hrají metody na počítání vržených stínů. Stíny jsou velmi důležité pro lidské vnímání, měkké stíny jsou navíc fyzikálně přesnější a taky výpočetně složitější. Vznikly tři hlavní skupiny algoritmů: projektivní stíny, stínové mapy a stínová tělesa. Každá z nich má své výhody a nedostatky. Několik pokročilejších metod je popsáno podrobněji. Měření kvality a výkonnosti, společně s přiloženým demonstračním programem, umožňují bližší seznámení s popsanými metodami.
Klíčová slova: počítačová grafika, realtime, stíny, měkké stíny, GPU

Title: Soft shadows on GPU
Author: Boris Zápotocký
Department: Department of Software and Computer Science Education
Supervisor: RNDr. Josef Pelikán
Supervisor's e-mail address: Josef.Pelikan@mff.cuni.cz
Abstract: Huge graphics hardware development and GPU programmability enables higher performance and more flexibility in programming. Many algorithms are now realtime. Very significant role among CG algorithms is held by shadow computation methods. Shadows are essential for human perception, soft shadows are physically correct and computationally consuming. Three major types of algorithms were proposed: projective shadows, shadow maps and shadow volumes. Each of them has advantages and disadvantages. Some of the advanced soft shadowing methods are covered in more detail. Measurements of both quality and performance along with the accompanying demonstrative software enables on the fly familiarization with the mentioned methods.
Keywords: computer graphics, realtime, shadows, soft shadows, GPU

Chapter 1

Introduction

1.1 Real world vs. computer graphics

The vision of computer graphics scientists and artists is to be able to fully reproduce images of real world on graphics chips. We are still very far from this ultimate goal. Yet some artists are already capable of creating almost indistinguishable pictures and sceneries. Still it is important to add, that they are being rendered within minutes or even hours. Realtime algorithms do not allow this level of realism so far.

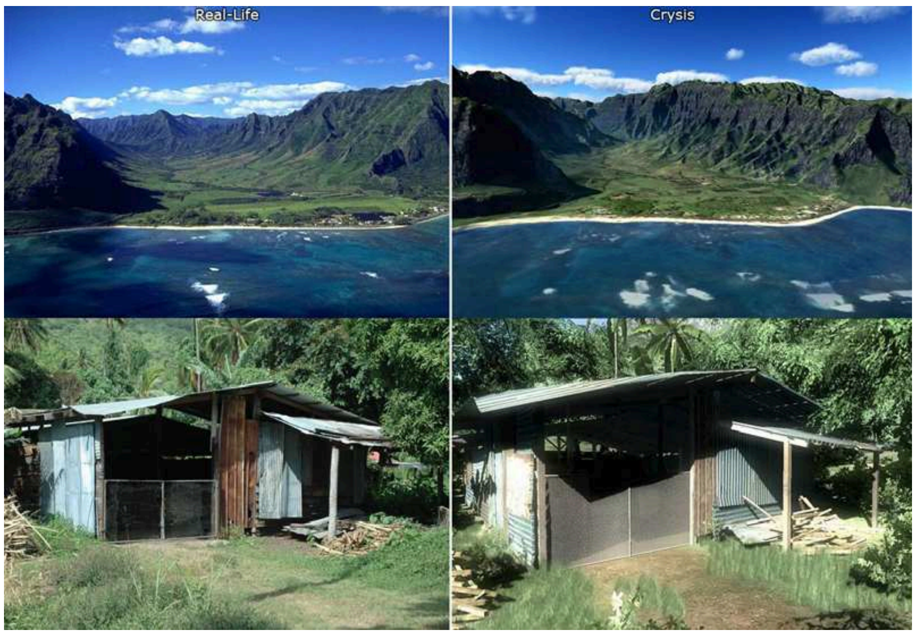


Figure 1.1: Comparison of real and cg scene *(courtesy of Crytek)*

Creating a real world physical realism in the realtime application (the most common example would be a computer game) is overwhelmingly complex. Lighting, shading, culling, rasterization... All of them are parts of a graphics pipeline. Yet only a part of CPU/GPU time is free for these computations, program needs to take care of physics (collision detection), AI (which is still a very weak part of today's games), processing of user inputs, sound, net communication... This makes it even more important to create visually perfect results in the shortest time possible.

Thus in realtime rendering it is always a struggle between visual complexity and performance. Thanks to the fast development of GPU segment in the last decade, we are able to produce better results in shorter time. Yet it is still very far from the quality of raytraced[1] images.

1.2 Computer graphics renaissance

In the last few years we have witnessed a great increase in the speed of graphics hardware development. It is the result of competition between industry leading companies, nVIDIA and ATI. Innovations and number of new features have overcome even the evolution of CPUs. Fixed function pipeline has been replaced by programmable *shaders*, greatly expanding the flexibility of realtime computer graphics programming.

With GPU programmability many time-consuming computations can be done directly on GPU, saving precious CPU time for other important evaluations. A programmer has a greater degree of freedom in what effects he wants to achieve. There are multiple high level shader programming languages today: HLSL (for Microsoft's Direct3D), GLSL (OpenGL) or cross platform Cg (nVIDIA). For more experienced developers assembler shaders are also possible (better optimization, than in the code created by high level language compilers, can be achieved).

Code 1.1: Vertex shader example

```
fpDataMap shadowMapVertex(
   uniform float4x4 ModelViewProj,
   uniform float4x4 TexTransform,
   uniform float4 LightPos,
   uniform float vis_dist,
   vpData IN)
{
   fpDataMap OUT;

   //get texture coords + position
   OUT.TexCoord0 = mul(TexTransform, IN.Position);
```

[1]raytracing is a popular method of non-realtime rendering

```
   OUT.Hposition = mul(ModelViewProj, IN.Position);

   return OUT;
}
```

Shaders can be divided[2] into vertex shaders (code run for every vertex of geometry) and fragment shaders (code run for every fragment after rasterization). The major drawback of first versions of shaders (mainly fragment profiles like `vp10`, `fp10`, etc.) was an absence of loops, if statements and severe limitations of code length. This is almost removed now, yet still not every aspect of C++ is possible (for example pointers). Debugging is yet another problem. It is not possible directly, thus increasing the time needed to successfully develop an effect.

Code 1.2: Fragment shader example

```
float4 shadowMapFragment(
   fpDataMap IN,
   uniform sampler2D ShadowMap,
   uniform float4 shadow) : COLOR
{
   //get depth value
   float depth = tex2Dproj(ShadowMap, IN.TexCoord0).r;

   return shadow*depth;
}
```

For more comfort, while working with shaders, the CgFx file format was developed. It encapsulates shaders, texture settings, etc. in one file. CgFx also enables creation of *techniques*, each defining some effect (with multiple passes). Different techniques for different generations of graphics HW are possible (the right code is chosen in runtime). Thus we can store effect shaders for PS3, X360, low-end GPUs and high-end GPUs in a single file.

Code 1.3: Technique example

```
//shadow map texture
sampler2D shadowMap = sampler_state {
   minFilter = Linear;
   magFilter = Linear;
   WrapS = ClampToBorder;
   WrapT = ClampToBorder;
   CompareMode = CompareRToTexture;
```

[2]new generation of nVIDIA cards brings unified shader architecture

```
   CompareFunc = GEqual;
   BorderColor = float4(1.0,1.0,1.0,1.0);
};

technique hardShadowMap < int type = 9; > {
   pass mapShadow {
      PolygonOffsetFillEnable = true;
      PolygonOffset = float2(-2.0,-10.0);

      DepthMask = false;
      DepthTestEnable = true;

      //cull back faces
      CullFaceEnable = true;
      CullFace = Back;

      BlendEnable = true;
      BlendFunc = int2(SrcAlpha,OneMinusSrcAlpha);

      VertexProgram = compile vp40 shadowMapVertex(
                mvp, shadowM,LightPosition,visibleDistance);
      FragmentProgram = compile fp40 shadowMapFragment(
                shadowMap,hardShadow);
   }
}
```

1.3 Shadows

The basic concept (no indirect lighting or multiple, area light sources are taken into account) of a *shadow* is simple. There are two possible situations. Either a point is in shadow or not. This model is considering *light source*, that is illuminating the scene, to be a *point light*(infinitely small emmiter of light, the closest example from the real world is the sun). Emmited light is cast in all directions uniformly. Surfaces hit by the light are lit and are considered to be *occluders* for other objects lying further away from the light source in that particular direction. These objects are in shadow and are called *receivers*.

Shadows are one of the most important factors in computer graphics (along with a color information). They allow us to perceive information about surface complexity:

- position and size of an occluder
- geometry of an occluder

- geometry of a receiver

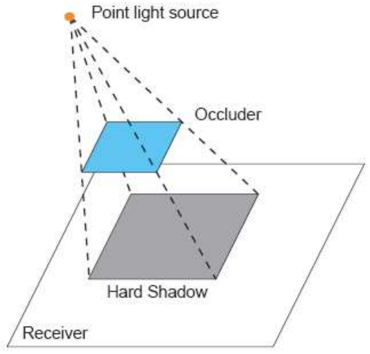


Figure 1.2: Shadow explanation

Shadows are also important for a human observer because our real world is full of them. Thus the human vision system is greatly used to their presence and absence of shadows has an essential influence on realism.

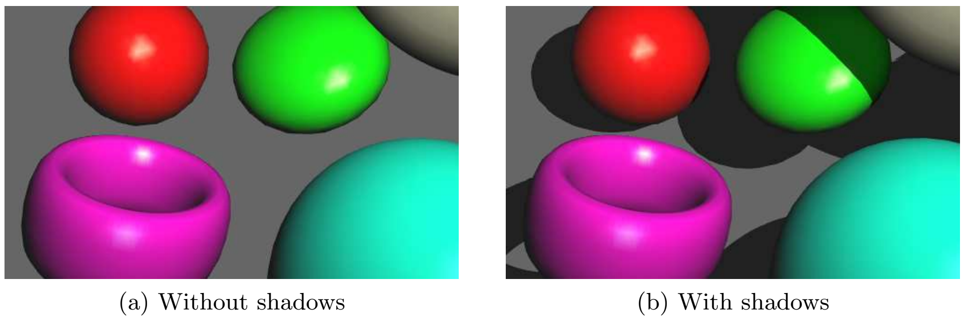

(a) Without shadows (b) With shadows

Figure 1.3: Importance of shadows

To go even further, we can distinguish two types of shadows:

- **self-shadows** occur when the shadow of an occluder is projected on itself (the occluder and receiver are the same)
- **cast shadows** occur when the shadow is projected on other object.

This distinction is very important because some algorithms (as shown further) cannot display self-shadows.

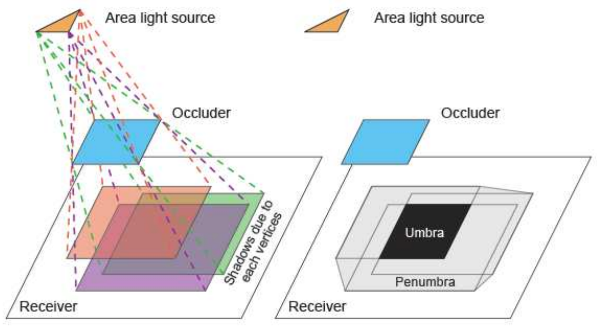
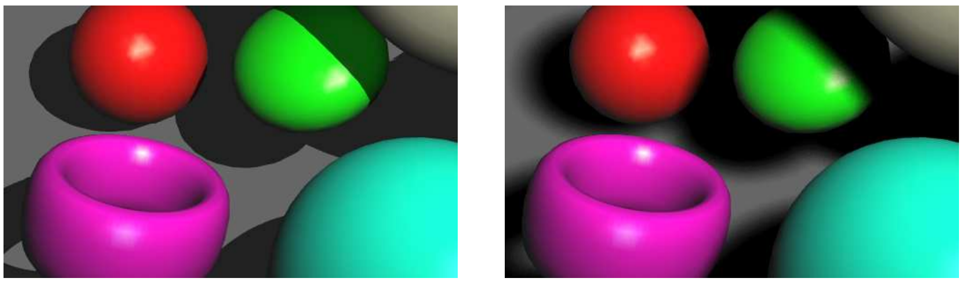


Figure 1.4: Soft shadow explanation

1.4 Soft shadows

Real world is full of shadows. Yet they are qualitatively much different from what we are used to see in realtime applications. They are much softer. It is caused mainly by physics in the world around us. Real light emitters are not point lights. Their dimensions, area, as well as the distance from shading object and distance from shaded surface are all factors resulting in more complex shadows. They have two parts:

- **umbra:** regions of full shadow (no part of the light is visible)
- **penumbra:** regions of partial shadow (not whole light is hidden)



(a) Hard shadow (b) Soft shadow

Figure 1.5: Hard vs. soft shadow

The main goal of shadow rendering development over the past few years is to devise an algorithm capable of computing realistically looking soft shadows with interactive frame rates. Almost every widely used algorithm for *hard shadows* is fast enough (on modern hardware). But for photo realistic rendering their quality is just insufficient.

To produce *soft shadows* (visually appealing) we need enormous number of light samples (up to 1024, some kind of the stochastic sampling is mainly used) and blending of shadows rendered with each of these samples. This degrades the performance linearly with their quantity.

Second group of techniques is not meant to create physically correct shadows but rather visually nice and believable ones. They are called *fake soft shadow algorithms* and are using only one light sample combined with sophisticated ways of sampling the precomputed shadow texture. This way we get much faster methods without loosing a high level of realism.

As a consequence, realtime computer graphics (and also the shadow computation) requires unpopular steps improving performance at the expense of quality of the resulting picture. A good survey of realtime soft shadow methods can be found in [HJF03].

1.5 Future

Figure 1.6: Future of realtime rendering?

What will the future bring us? According to today's trends, huge pace of graphics hardware development is not going to decrease in the next few years. This means that many algorithms, that are too expensive and time consuming to be used in realtime application today, will become usable.

Of course not only the hardware is evolving. Some new shadow rendering techniques can also be expected. As the applications for common consumers are becoming more and more demanding, the usage of soft shadows is beginning to be an essential part of them. This brings an impulse (also financial) for a massive research in this field of expertise.

With all this in mind, also the mainly non-realtime technique of *raytracing* is undergoing an enormous evolution pointing towards an interactive rendering performance. Slussalek[3] et al. is developing a dedicated hardware for realtime raytracing. The OpenRT project[4] is very vital and who knows, maybe one day the GPU architecture as we know it today will be forgotten...

[3]Universität des Saarlandes, Germany

[4]The OpenRT Real-Time Ray-Tracing Project, http://www.openrt.de/

Chapter 2

Main shadow algorithms

2.1 Introduction

In this section we would introduce a few of the most frequently used algorithms for shadow computation. They have evolved through the years and are used in almost every contemporary game/application where some level of realism is desired. Three major categories have emerged:

- **Projective shadows**
- **Shadow maps**
- **Shadow volumes**
- *precomputed shadow textures (light maps, global illumination, etc.) → not the scope of this work*

Each algorithm introduces various levels of computational complexity and accordingly also differing visual quality. They can be used for soft shadows computation as well (brute force, fake soft shadows).

2.2 Projective shadows (planar)

2.2.1 Method

This method is still among the most used. It is fast and relatively visually appealing. Shadows are created by projection of the geometry into receiver's plane [Bli88]. This is really straightforward but brings also many drawbacks. Receiver has to be planar (this is the reason it is widely used in games for shadowing of floor tiles), each plane requires different transformation matrix (for example shadows on walls require running the algorithm for each wall separately, the same is true for multiple light sources, etc.). Other drawback is the absence of self-shadows (clear from the nature of the algorithm).

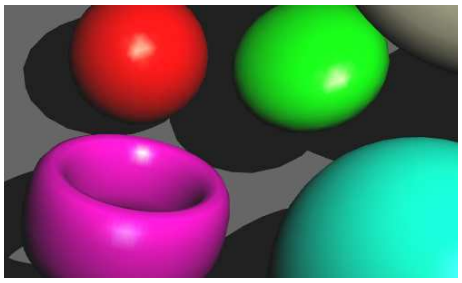


Figure 2.1: Projective shadow

2.2.2 Algorithm

Let us explain it a bit further. The crucial part is getting the shadow projection matrix (based on the light position and receiver's plane equation).

For this we need to know receiver's plane equation. Plane can be defined by its point p and normal vector $\vec{n}$. Then a point r lies in the plane `iff`

$$\vec{n} \cdot (p - r) = 0$$

This can be rewritten as

$$ax + by + cz + d = 0 \qquad (2.1)$$

where

$$\vec{n} = \begin{pmatrix} a \\ b \\ c \end{pmatrix}$$

and

$$\vec{n} \cdot p = -d \qquad (2.2)$$

With this knowledge, for a plane defined by three points p_1, p_2, p_3 we can compute desired coefficients a, b, c by applying a cross-product

$$\vec{n} = (p_2 - p_1) \times (p_3 - p_1)$$

and coefficient d by substituting point p_i into the equation (2.2).

The second step towards projective shadow is the projection matrix. All we need are the plane equation coefficients $\vec{p}$ (2.1) and light position homogeneous coordinates $\vec{l}$. With the help of the *dot product* of these vectors

$$dot = \vec{p} \cdot \vec{l} = p_0 * l_0 + p_1 * l_1 + p_2 * l_2 + p_3 * l_3$$

the resulting matrix can be computed as follows:

$$\begin{pmatrix} dot - l_0 * p_0 & -l_1 * p_0 & -l_2 * p_0 & -l_3 * p_0 \\ -l_0 * p_1 & dot - l_1 * p_1 & -l_2 * p_1 & -l_3 * p_1 \\ -l_0 * p_2 & -l_1 * p_2 & dot - l_2 * p_2 & -l_3 * p_2 \\ -l_0 * p_3 & -l_1 * p_3 & -l_2 * p_3 & dot - l_3 * p_3 \end{pmatrix} \tag{2.3}$$

After computing shadow projection matrix M_p, all that is left to do is to multiply current modelview transformation matrix M_{mv} with it and draw the occluder geometry. It will be projected into the receiver's plane.

$$P' = M_{mv} * M_p * P \tag{2.4}$$

Remark: *it is useful to avoid drawing the shadow outside receiver's geometry by setting accordingly the stencil buffer.*

2.2.3 Pseudo Code

Short recapitulation of the algorithm in a pseudo code follows:

Code 2.1: Projective shadows

```
//draw all shadow casters
drawScene();

//foreach shadow receiver plane
foreach(receiver) {
   //compute matrix transformation
   computeTransformMatrix(receiver,matrix);

   //update default modelview matrix
   multMatrix(matrix);

   //draw all shadow casters
   //with shadow color and blend
   drawScene(shadow);
}
```

2.2.4 Usage

Method is widely used in complex, realtime environments with a lot of outdoor spaces in daylight (so the projection of shadows is necessary only against floor plane and the only light source is the sun). Lack of self shadowing can be disguised with good textures. Common observer is satisfied with the presence of shadow phenomena, physical correctness is not that important.

2.2.5 Examples

This approach can be seen in the game Half-life 2 by Valve Software. Each character casts shadow on the ground, no self shadowing is present. Shadows are computed only against the sun position (even inside buildings). This can be seen on the ingame screenshot.

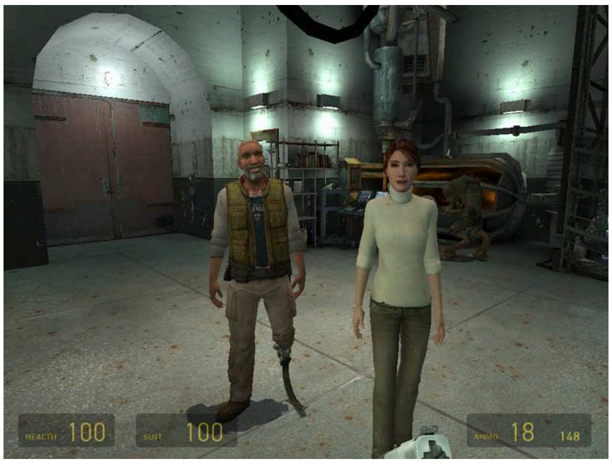


Figure 2.2: Half-Life 2 screenshot *(courtesy of Valve)*

2.2.6 Conclusion

	Pros		Cons
+	easy computation	-	no self shadows
+	visual quality	-	planar receiver only

Table 2.1: Projective shadows technique review

2.3 Shadow maps

2.3.1 Method

Method produces shadows using depth value textures [L.78]. Scene is rendered from a light point of view into the depth texture. When rendering resulting image, distance of current pixel to the light source is compared with the depth stored in the texture. If it is larger, pixel is in shadow (some occluder geometry is closer to the light source). This approach is very popular because of an easy implementation.

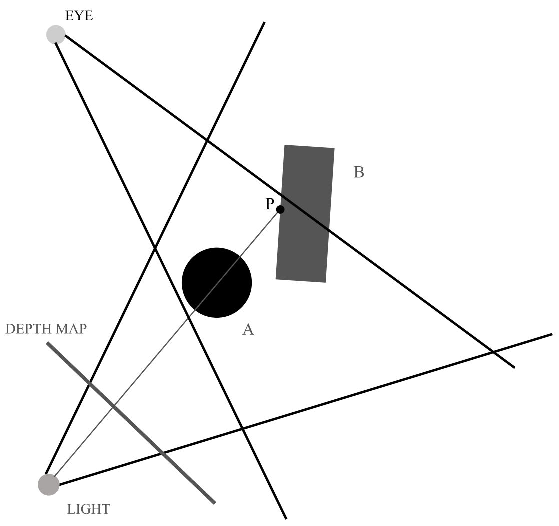


Figure 2.3: Shadow map illustration

Problems are with an appropriate resolution of the shadow map (resolution too low creates big "jaggies" (aliasing) at the edges of drawn shadow, this is even amplified by perspective projection). Bigger maps are, on the other hand, very memory consuming. More ways how to deal with these problems have been proposed: perspective shadow maps [MS02], PCF (percentage closer filtering) [RWR87]... Each is good in some contexts and problematic otherwise.

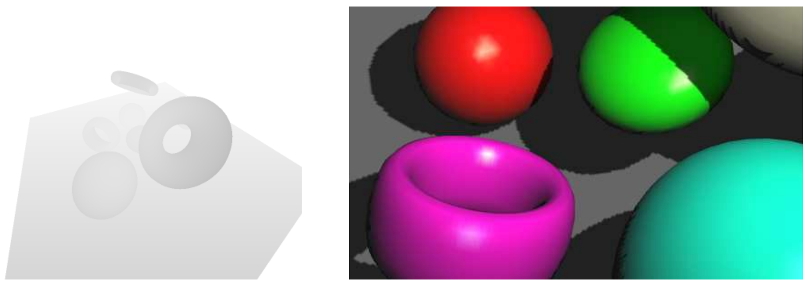

(a) Visualisation (b) Resulting shadows

Figure 2.4: Shadow mapping

2.3.2 Algorithm

This method has two important (independent) stages:

1. **depth map acquisition** – needs recomputation only for dynamic scenes
2. **depth comparison** – can be automated (in OpenGL using texture mode `GL_COMPARE_R_TO_TEXTURE` with an additional HW interpolation) or done in a shader

Acquisition of depth map can be done in advance (to save precious computation time). When the only moving object in a scene is camera, we do not need to update shadow maps.

During this part of the algorithm, the scene is rendered from a light point of view (has to be done for each light source separately). Only depth buffer information is important. It is stored in a texture (each pixel of the texture will have value from 0 to 1 according to the previous setting of the view frustrum).

It is important to save matrix for transformation of the world coordinates into the light space coordinates along with depth texture generation. We can use modelview matrix M and projection matrix P from the light point of view settings. Multiplication by these matrices will convert point coordinates into the light space coordinates (values from -1 to 1). Yet we need texture space coordinates (they are usually from 0 to 1). Thus one more matrix multiplication is needed. We need to scale values by 0.5 and translate them by +0.5. Desired bias matrix is therefore

$$ST = \begin{pmatrix} 0.5 & 0 & 0 & 0.5 \\ 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.5}$$

Complete transformation matrix is then a result of matrix multiplication:

$$T = ST * P * M \tag{2.6}$$

The second part is all about depth comparisons. After drawing the scene in normal colors we need one extra pass to draw a shadow over shaded parts of the scene.

Remark: *It can be naturally done also vice versa, by drawing the whole scene in a shadow color and then adding not shaded parts of screen in normal colors.*

For each pixel its depth value d_{obj} is computed (z coordinate after transformation by T matrix (2.5))). This is compared with a stored value from the texture d_{map}. There are two possibilities:

1. $d_{obj} < d_{map} \Rightarrow$ pixel is **not in shadow**
2. $d_{obj} >= d_{map} \Rightarrow$ pixel is **in shadow**

Remark: *There are several problems connected with the use of shadow maps. Because of numerical errors during transformations and limited precision of the depth buffer* ***self shadow alias*** *artifacts arise. They can be limited by computing the shadow maps only from back-facing geometry triangles and by ε translation of the geometry closer to the light source (by enabling* `GL_POLYGON_OFFSET_FILL` *in OpenGL). Another important thing is the previously mentioned* ***alias on the shadow edges****, caused by low depth texture resolution and the need to magnify texture after projection. Partial solution is to increase resolution of shadow map or to use some kind of PCF. It does not remove alias completely, but makes it less apparent.*

2.3.3 Pseudo Code

Short recapitulation of the algorithm in a pseudo code follows:

Code 2.2: Shadow maps

```
//draw scene from the light point of view
//capture depth values into shadow map texture
//and save transformation matrix
map = generateShadowMap(lightPosition, &shadowMatrix);

//draw default scene, use pregenerated texture
foreach (pixel P in scene) {
   //transform P into light space
   P’ = transform(P,lightPosition);

   //compare P’s depth value with stored depth value
   if (P’.z > map(P’.x,P’.y))
      //pixel is shaded
      PixelColor(shadow);
   else
      //pixel is not shaded
      PixelColor(P->object);
}
```

Remark: *It is important to generate shadow maps with sufficient resolution. This is usually higher than current screen resolution (up to 4096x4096 on modern GPUs). Thus it is impossible to get such texture with classic OpenGL color buffers and various extensions have to be used (for example* `EXT_framebuffer_object`*).*

2.3.4 Usage

Method is very popular in games, because it is easy to compute (or precompute if the scene is static) and apply. A crucial part is to choose the resolution for the shadow maps wisely. They have to be updated only for dynamic parts of the scene, static parts can use precomputed shadow maps (then their application is reduced to a simple texture lookup).

2.3.5 Examples

Technique is widely used in Neverwinter Nights 2, etc.

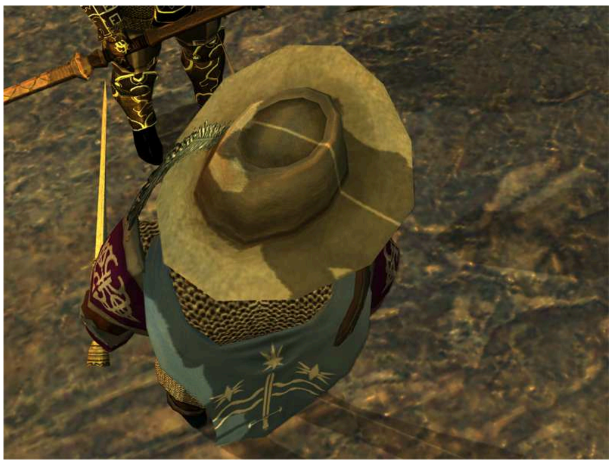


Figure 2.5: Neverwinter Nights 2 screenshot *(courtesy of Obsidian Entertainment)*

2.3.6 Conclusion

	Pros		**Cons**
+	easy computation	-	visual quality
+	self shadowing	-	memory requirements

Table 2.2: Shadow map technique review

2.4 Shadow volumes

2.4.1 Method

This method computes shadows by creating a shadow geometry [C.77]. This is done by extending the silhouette edges (edges which separate front-facing faces from back-facing faces) along the light-to-vertex direction away from the light. Each object is thus transformed into a shadow volume. The shadow volume delimits the part of space where the light source is not visible (it is shaded by the given object). Algorithm is robust with a pleasing quality of results. Yet without any optimization it can be really slow.

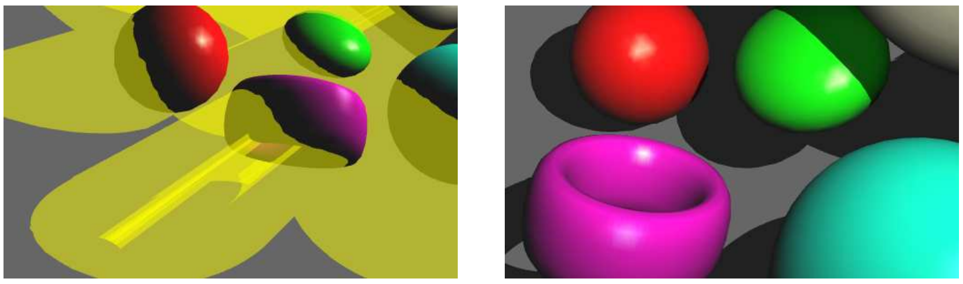

(a) Visualization (b) Resulting shadows

Figure 2.6: Shadow volumes

Optimizations are mainly trying to do shadow volume computations only against silhouette edges. Their finding and maintenance (in case of a dynamic scene) is rather complex and slow (done on CPU). Some methods were proposed for finding candidates for silhouette edges [Len02].

Usual implementation works with manipulation of the stencil buffer. Actual computation of whether the pixel is in shadow or not can be done by three different approaches: **depth-pass, depth-fail** and **exclusive-or** (they differ in number of passes they require and also in the way of altering stencil buffer values).

2.4.2 Algorithm

The process of this method is composed of three parts. First, we draw normal scene. Then the shadow volumes are created (with the chosen method). An important optimization is to precompute silhouette edges candidates on CPU and use only them for shadow volumes, otherwise the performance is low. The last thing that needs to be done is to draw the scene geometry again (this time in a shadow color), now only over the parts set by stencil buffer.

Over the time many extensions have been proposed for HW developers, which can speed shadow volume algorithms up a bit (for example `GL_NV_depth_clamp`, `GL_EXT_stencil_two_side`[1], `GL_EXT_stencil_wrap`, etc.). They, however, require modern graphics cards.

[1]enables two different stencil operations according to the face orientation, this saves one pass

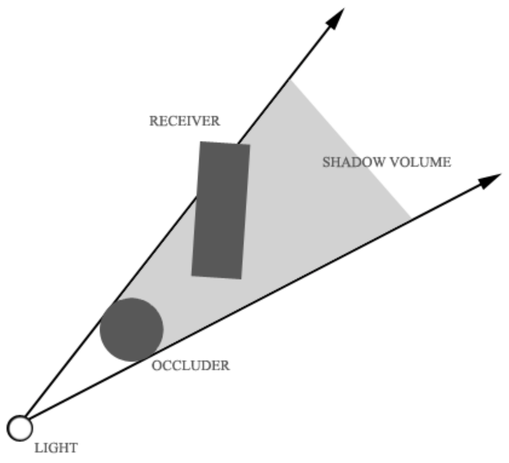


Figure 2.7: Shadow volume illustration

2.4.3 Depth-pass

This is the first proposed method to use stencil buffer during the shadow volume generation [Hei91]. It requires two passes, where front-facing and back-facing faces of the shadow geometry are counted in the stencil buffer. It is based on a fact that object's surface point, which is in the shadow will have more front-facing faces than back-facing faces between itself and the eye. Therefore it can be written like this:

1. use back-face culling
2. set the stencil operation to increment on depth pass
3. render shadow geometry
4. use front-face culling
5. set the stencil operation to decrement on depth pass
6. render shadow geometry

After this render shadow only where stencil buffer is $\neq 0$.

Remark: *The scene settings where eye is situated inside the shadow volume are problematic. Then we get reverted and incorrect results.*

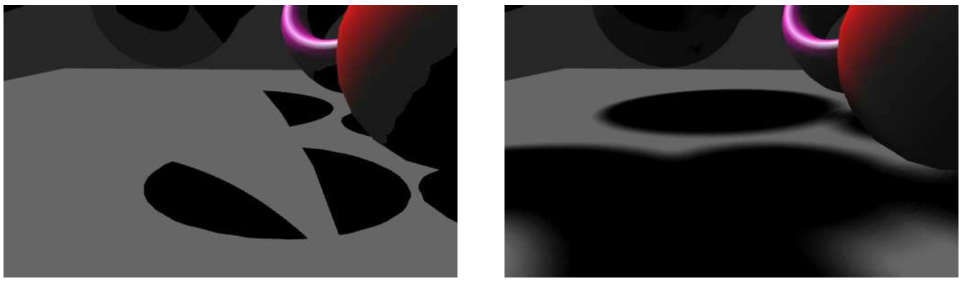

(a) Problem (b) Correct, soft version

Figure 2.8: Depth pass method

2.4.4 Depth-fail

Around year 2000, several people discovered[2] a way how to avoid problems with the previous method [BB99, Car00]. All that is necessary is to reverse the depth. Instead of counting volumes in front of the object, the same can be done for the volumes behind the object. This solves the problem of eye being in shadow, yet it needs back caps of shadow volumes (otherwise the shadow will be missing where shadow volume points to infinity). Therefore the depth-fail version of algorithm can be written like this:

1. use front-face culling
2. set the stencil operation to increment on depth fail
3. render shadow geometry
4. use back-face culling
5. set the stencil operation to decrement on depth fail
6. render shadow geometry

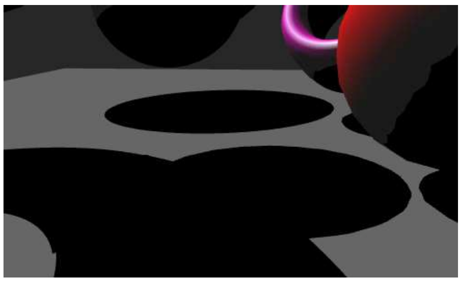
After this render shadow only where stencil buffer is $\neq 0$.

Remark: *This method is slower than depth-pass and therefore should be used only when necessary.*

2.4.5 Exclusive-or

This method requires only one pass, yet it is correct only for special situations and does not deal properly with overlapping shadow volumes. It can be written like this:

[2]one of them is John Carmack of ID software, method is also known as Carmack's Reverse


(a) Correct

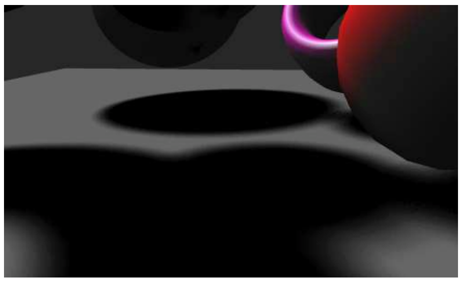
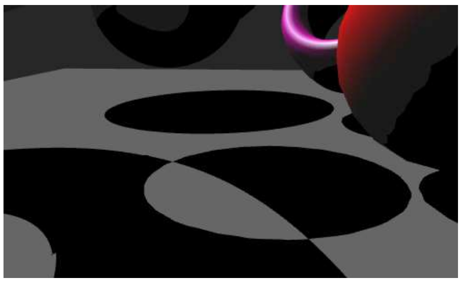

(b) Correct, soft version

Figure 2.9: Depth fail method

1. disable culling
2. set the stencil operation to XOR on depth pass
3. render shadow geometry

After this render shadow only where stencil buffer is $\neq 0$.


(a) Problem

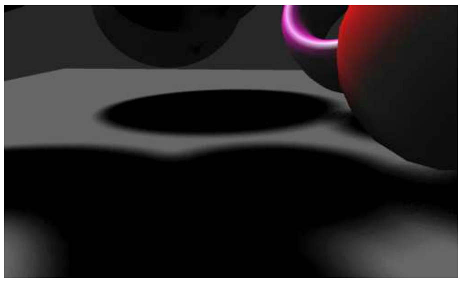

(b) Correct, soft version

Figure 2.10: Exclusive-or method

2.4.6 Pseudo Code

Short recapitulation of the algorithm in a pseudo code (using the most common depth fail settings) follows:

Code 2.3: Shadow volumes

```
//1.shadow computation phase
//using extended "shadow" geometry construct a mask
//in the buffer, with non zero values where shadow is
//prepare geometry, set stencil
```

```
setCulling(front);
setStencil(incr,depth_fail);

//render geometry
drawScene();

//prepare geometry, set stencil
setCulling(back);
setStencil(decr,depth_fail);

//render geometry
drawScene();

//2.rendering phase
//draw the default scene (completely lit)
drawScene();

//draw the scene with shadow color
//according to the prepared mask
drawScene(mask, shadowColor);
```

2.4.7 Usage

This is a popular choice for scenes with complex lighting conditions (multiple, dynamic, moving light sources...), where artifacts from the use of shadow maps would be unbearable. Usage of shadow volumes brings great visual experience with self shadowing and visually stunning shadows.

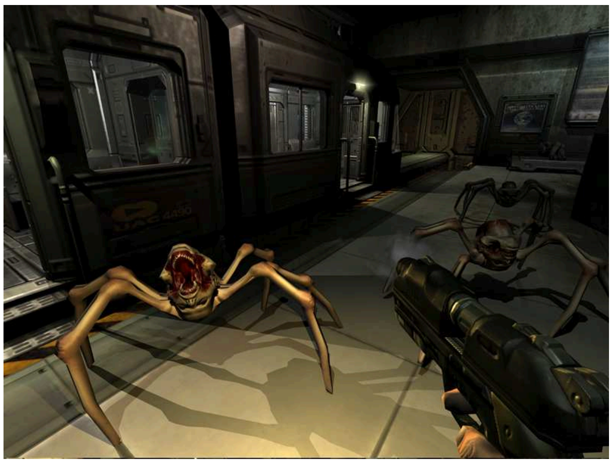


Figure 2.11: Doom3 screenshot *(courtesy of id Software)*

2.4.8 Examples

Approach appeared in Doom3 by ID software. Improvements to the algorithm were introduced during the game development (depth-fail version of algorithm).

2.4.9 Conclusion

	Pros		**Cons**
+	visual quality	-	slow, complex computation
+	self shadowing		

Table 2.3: Shadow volume technique review

Chapter 3

Soft shadow algorithms

3.1 Introduction

As mentioned before, shadows are a crucial part of understanding scene complexity, mutual positions, closeness of the objects, etc. Generation of hard shadows is well mapped and easily implementable on today's GPUs. Yet they are unnatural and not suitable for realistic image rendering.

Future unprecedentedly requires soft shadows. Not even they feel realistic (even when computed by fake methods), they are also addictive and if you have seen a realtime application with soft shadows, every hard shadow would seem like a step back into the prehistory...

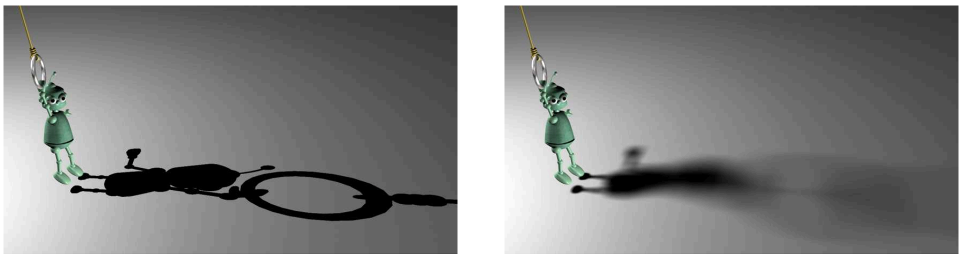

(a) Hard shadow (b) Soft shadow

Figure 3.1: Visual difference between hard and soft shadow

To achieve realistic, physically correct shadow an area light has to be densely sampled (number of samples depends on the dimensions of light and its distance from occluders, etc.). Then the shadow is computed as a superset of shadows generated for each light sample. This is very slow as it requires many rendering passes with the scene geometry.

Thus in interactive applications mainly fake shadow methods are used. Shadows are not entirely physically correct, yet for human observer that is not such a problem as this fact is not as disturbing as hard shadow edges (for which there is no real world equivalent and our eyes are very sensitive to this kind of the edges).

Main disadvantage of all methods is the computational complexity to achieve good results. For some of the more advanced methods (shadow volume penumbra wedges [TAM02a]), the main bottleneck remains the necessity to do a lot of work on CPU (in every cycle for a dynamic scene) as some more complex precomputation steps (as silhouette edges lookup) are not achievable[1] on GPUs with their streaming architecture.

The most GPU-friendly algorithms (with all the desired features, like self-shadowing and dynamic penumbra size) are based on shadow mapping technique. GPUs capabilities of handling texture lookups is getting faster and more complex with each new generation of graphics chips.

Over the years many methods were proposed, some of them usable for interactive applications and some not. The most interesting ones are listed below and they will be explained further.

- **PCF** – Percentage Closer Filtering, basic method for softening of hard and aliased edges of shadow maps, average quality
- **Blurred PCF** – blurring the PCF shadow map with Gaussian filtering, very fast, very good quality, yet some visible artifacts are present
- **Jittered PCF** – stratified supersampling of shadow map, very good quality, slower
- **PCSS** – Percentage Closer Soft Shadowing, very good quality, slower

Of course also their combination and modifications are possible. These algorithms can be used globally. For specific scenes (static, light source very far, small number of shadow receivers, etc.) also other specific purpose methods exist.

3.2 Percentage closer filtering (PCF)

This soft shadow mapping method was proposed by Reeves et al. in 1987 [RWR87]. It is based on filtering of shadow transparency by calculating how many percent of neighborhood texels are closer to the light than the point being illuminated.

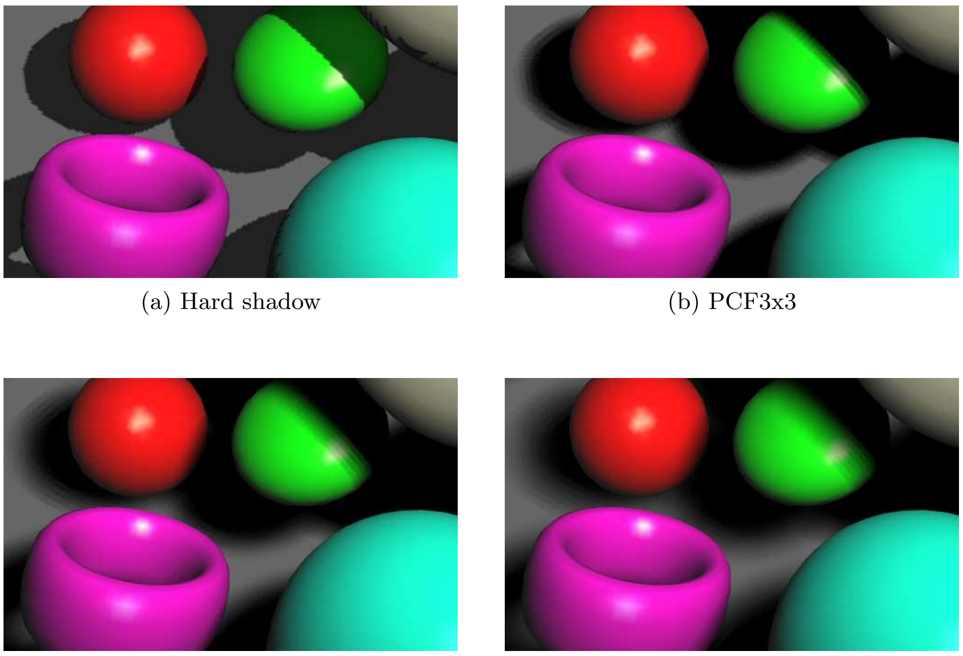
Algorithm is based on texture lookups, on modern GPUs there is also a HW support for PCF2x2 prefiltering (with bilinear interpolation). With this support enabled even hard shadow mapping is blurred a bit at the edges.

[1]with the new generation of GPUs based on G80 chip from nVIDIA with the capability of unified shaders it is a little bit less painful. Even the raytracer with acceleration structures (kD-trees) have been successfully implemented on this HW.

$$\begin{array}{|c|c|c|}\hline 74 & 79 & 56 \\ \hline 70 & 64 & 39 \\ \hline 69 & 53 & 21 \\ \hline \end{array} \overset{<60?}{\Longrightarrow} \begin{array}{|c|c|c|}\hline 0 & 0 & 1 \\ \hline 0 & 0 & 1 \\ \hline 0 & 1 & 1 \\ \hline \end{array} \overset{\sum/9}{\Longrightarrow} \quad 44\%$$

Figure 3.2: PCF3x3 filtering

Another way of automating this process is to set the texture comparison mode (in OpenGL) to `GL_COMPARE_R_TO_TEXTURE`. Not the exact depth value is returned after a texture lookup but rather a `boolean` (depending on comparison function settings) result of texture value comparison with fragment depth.


(a) Hard shadow

(b) PCF3x3

(c) PCF5x5

(d) PCF7x7

Figure 3.3: PCF with different sizes

Resulting softness of the drawn shadow is affected by the size of sampled neighborhood. For a good result we usually need at least 7x7 neighborhood averaging, yet with growing size the algorithm also gets slower as this filtering takes place in each fragment processed. Fragment shaders are very sensitive to the number of queried operations.

Remark: *This method is a fake shadow map method, as the size of the penumbra depends only on the size of neighborhood and not on the physical properties of the scene.*

3.2.1 Code example

Fragment shader involved can be written as follows:

Code 3.1: PCF method

```
float4 fakeShadowMapFragment(fpDataMap2 IN,
   uniform sampler2D ShadowMap,
   uniform float offset,
   uniform float4 shadow) : COLOR
{
   float shadowHardness = 0.0;

   //PCF 7x7
   for (int i=-3;i<=3;i++)
      for (int j=-3;j<=3;j++)
         shadowHardness += tex2Dproj(ShadowMap,
                           IN.TexCoord0 + offset*float2(i, j));

   //get average value
   shadowHardness = shadowHardness / 49;

   //output color
   return shadow*shadowHardness;
}
```

3.2.2 Conclusion

	Pros		**Cons**
+	soft shadows	-	visual quality
+	HW support	-	constant penumbra size

Table 3.1: PCF technique review

3.3 Blurred PCF

This method is an extension to the widely used PCF. It was published in the year 2005 [Sha05]. Its big advantage is fast computation and thus good performance even in complex scenes.

It is based on shadow map generation. Instead of actually using it to draw shadow in the scene, shadow map is drawn into the texture with the help of PCF.

This texture is then blurred with Gaussian filter, which is separable. Therefore it can be used for blurring in horizontal and afterwards vertical direction doing only $2N$ texture lookups (which is the main reason of speedup) instead of N^2. Blurred result is then projected back into the screen space and blended with the basic scene.

Here are the steps involved:

1. generate shadow map
2. render shadowed parts of the scene into the texture
3. blur this texture in horizontal direction and store
4. blur resulting texture in vertical direction and store
5. project the blurred texture into the scene

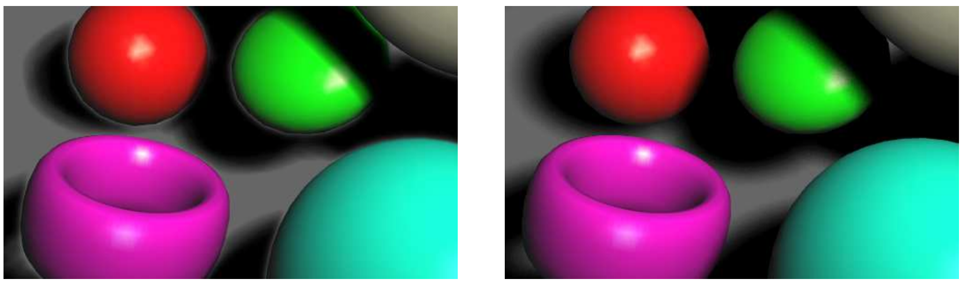

(a) PCF blur (b) Soft shadow

Figure 3.4: PCF blur halo problem

Main problem of this approach is the lost depth information. Blurring the resulting shadow texture causes spreading of not shadowed parts into original umbra. Thus objects touching (in screen space) some occluder (area without shadow in original shadow texture) have clearly visible, not shadowed halo along the occluder's edge.

Remark: *Still it works perfectly for scenes without overlaps (in screen space) between occluders and receivers, etc.*

3.3.1 Code example

Code consists of the basic fragment shader for PCF (simple 3x3 version) and a fragment shader for blurring stored texture with Gaussian:

Code 3.2: PCF blur method

```
float4 shadowMapBlur(
   float2 texCoord : TEXCOORD0,
   uniform sampler2D blurTexture,
   uniform float gaussWeight[],
   uniform float2 gaussOffset[]) : COLOR
{
   //accumulated color
   float4 vAccum = float4( 0.0f, 0.0f, 0.0f, 0.0f );

   //sample the taps (gaussOffset holds the texel offsets
   //and gaussWeight holds the texel weights)
   for(int i = 0; i < gaussOffset.length; i++ )
   {
      vAccum += tex2D( blurTexture, texCoord + gaussOffset[i] )
                * gaussWeight[i];
   }

   return vAccum;
}
```

3.3.2 Conclusion

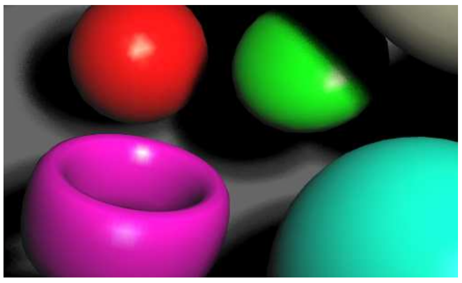
	Pros		**Cons**
+	soft shadows	-	halo effect artifacts
+	very fast	-	constant penumbra size
+	visual quality		

Table 3.2: Blurred PCF technique review

3.4 Jittered PCF

The main problem of PCF shadow maps is a high number of texture lookups needed to achieve good looking results. To increase this number means only to move the banding artifacts into higher frequencies. Yet human visual system is very sensitive to the strong edges (high frequency information).

This method was proposed with the aim on removing the banding artifacts [Ura05]. It is very well known that the human visual system is much less sensitive to a noise than the edges, therefore it is possible to do this PCF filtering not with regular grid samples but instead using jittered grid samples (where each sample is inside its grid cell pushed by a random vector).


(a) 32 samples

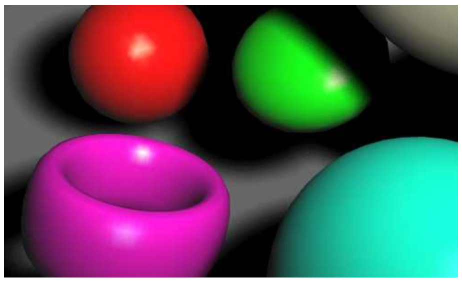

(b) 64 sample

Figure 3.5: PCF jitter comparison

Furthermore, jittered grid is warped to form a disk. This brings less distortion artifacts, when compared to a square grid. While generating the square grid is pretty straightforward, warping it into disk (with area preserving square-disk transformation) is trickier. It can be done by formulas:

$$x = \sqrt{v} \times \cos 2\pi u \tag{3.1}$$

$$y = \sqrt{v} \times \sin 2\pi u \tag{3.2}$$

where $u, v \in [0..1]$ represent jittered sample location within square domain and x, y are their counterparts in the disk domain.

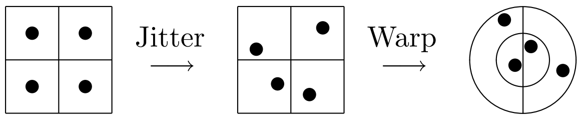


Figure 3.6: Regular grid jittering and warping

We can precompute jittered samples and store them in a texture. Authors recommend 3D texture (x, y dimensions are small and represent a small block where each texel has different set of samples stored in the z dimension). This way we avoid situation when two neighboring fragments will sample shadow texture in the same positions. So the key point is that the sequence of shadow map sample locations is each time different, thus the approximation error is different at different fragments. This effectively replaces banding with high frequency noise. Error is still there but it is much less visible.

Last thing worth of notice is the branching in fragment shader. Here we can use the fact, that in umbra regions it is useless to use all samples (author uses 64 samples per pixel) because all of them will result in full shadow. Thus we can first test pixel only on lower number of samples. Only if the result is not clear (not 0 or 1), which means a high probability of penumbra pixel, we do the remaining texture lookups.

3.4.1 Code example

Fragment shader involved can be written as follows:

Code 3.3: PCF jitter method

```
float4 jitterPCF(fpDataMap IN,
   uniform sampler2D ShadowMap,
   uniform sampler3D jitterTex,
   uniform float2 offset) : COLOR
{
   float shadowHardness = 0.0;

   float2 offsetScale = offset;

   //coordinates for lookup jitter texture
   float3 coord = float3(IN.TexCoord0.x,IN.TexCoord0.y,0);

   //cheap test samples (8)
   for( int i=0; i<4;i++) {
      //sample lookup texture
      float4 off = tex3D(jitterTex, coord)*offsetScale.xyxy;

      coord.z += jitterInvSamples;

      //is in shadow? 2 offsets are stored in 1 value from jitterTex
      shadowHardness += tex2Dproj(shadowMap,IN.TexCoord0 + off.xy);
      shadowHardness += tex2Dproj(shadowMap,IN.TexCoord0 + off.zw);
   }

   //do we need to continue sampling (is penumbra hit highly probable)?
   if ((shadowHardness - 8) * shadowHardness != 0) {
      for (int i=0; i < jitterSamplesDiv2 - 4; i++) {
         //sample lookup texture
         float4 off = tex3D(jitterTex, coord)*offsetScale.xyxy;

         coord.z += jitterInvSamples;

         //is in shadow? 2 offsets are stored in 1 value from jitterTex
         shadowHardness += tex2Dproj(shadowMap,IN.TexCoord0 + off.xy);
         shadowHardness += tex2Dproj(shadowMap,IN.TexCoord0 + off.zw);
      }

      //average value
      shadowHardness *= 0.5 * jitterInvSamples;
```

```
    }

    //return computed shadow
    return float4(0,0,0,shadowHardness);
}
```

3.4.2 Conclusion

	Pros		Cons
+	soft shadows	-	only average speed
+	visual quality	-	constant penumbra size

Table 3.3: Jittered PCF technique review

3.5 Percentage closer soft shadows (PCSS)

Each of the previously mentioned methods has one drawback in common: inability to create penumbra regions with varying sizes depending on distances between the light source, occluder and receiver.

In the year 2005 a new method was published [Ran05], based on PCF but allowing this essential feature of generated soft shadow believability. Its basic assumption is the notion, that increasing PCF kernel creates softer shadows, therefore challenge is to vary filter size intelligently to achieve accurate degree of softness (based on relative location of objects in the scene).

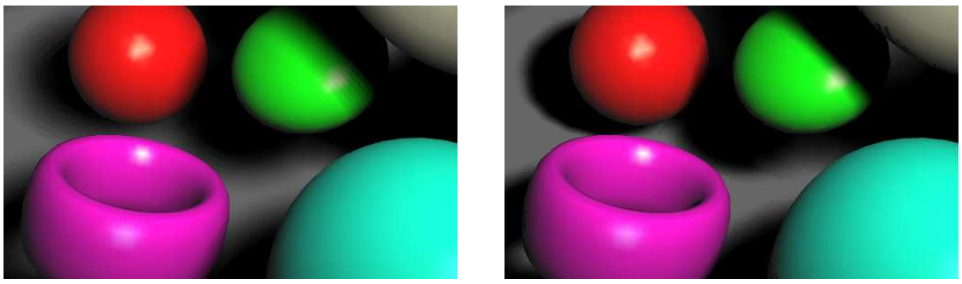

(a) PCF (b) PCSS

Figure 3.7: PCSS visual dominance over PCF

This is done in three steps:

1. **Blocker search** – Shadow map is searched in small neighborhood. Size depends on the light size and the distance between receiver and the light source. Retrieved depths (only that are closer to the light source than to the receiver) are averaged to get the distance light-occluder. Neighborhood search and averaging is essential to get the correct values also for outer penumbra, where no depth value is stored in shadow map.

2. **Penumbra estimation** – Penumbra width is estimated based on the light size, blocker and receiver distances from the light (with the assumption, that the blocker, receiver and light source are parallel):

$$w_{penumbra} = (d_{receiver} - d_{blocker}) \cdot w_{light}/d_{blocker} \tag{3.3}$$

3. **Filtering** – Typical PCF filtering with kernel size proportional to the penumbra size estimation.

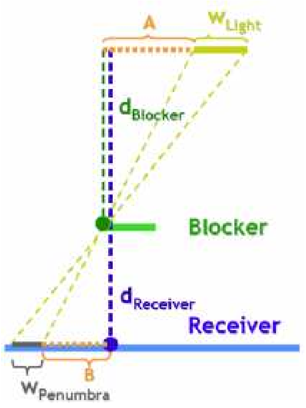


Figure 3.8: Penumbra size estimation

Remark: *Resulting shadows are visually appealing and dynamically react to the changes in the scene geometry distances. Yet this method still doesn't take into account other than perfectly circular area light source and is relatively slow to achieve good results.*

3.5.1 Code example

Fragment shader involved can be written as follows (inspired by an implementation in [Mik07]):

Code 3.4: PCSS method

```
float4 fragmentShadowMapPCSS(
   fpDataMap IN,
   uniform sampler2D shadowMap,
   uniform sampler2D shadowMap2,
   uniform float4 shadow,
   uniform float2 offset): COLOR
{
   //compute radius of neighborhood for occluder scanning
   float receiver = IN.TexCoord0.z/IN.TexCoord0.w;;
   float2 radius = offset*(flatSize/receiver);

   float occluder = 0.;
   float occluderCount = 0.;

   //scan the neighbourhood for occluders
   for (int i=-1; i<=1; i++)
      for (int j=-1; j<=1; j++) {
         //get the depth value from shadow map
         float depthVal = tex2Dproj(shadowMap2,IN.TexCoord0
                           + float2(i*radius.x, j*radius.y)).r;

         float isBlocker = step(depthVal, receiver);

         //update occluder count and stored depth
         occluder += depthVal*isBlocker;
         occluderCount += isBlocker;
   }

   //compute average of occluding depths
   occluder /= occluderCount;

   //estimation of penumbra size
   float penumbra = flatSize*(receiver-occluder)/occluder;

   radius = 2*offset*penumbra;

   float shadowHardness = 0.;

   //perform ordinary PCF with diameter of penumbra size
   for (int i=-3; i<=3; i++)
      for (int j=-3; j<=3; j++)
         shadowHardness += tex2Dproj(shadowMap,IN.TexCoord0
                            + float2(i*radius.x, j*radius.y));

   shadowHardness /= 49.;

   //return fragment color
   return float4(0,0,0,shadowHardness);
}
```

3.5.2 Conclusion

	Pros		**Cons**
+	soft shadows	-	only average speed
+	visual quality		
+	varying penumbra size		

Table 3.4: PCSS technique review

Chapter 4

Implementation

4.1 Introduction

The purpose of this thesis was to implement various methods of soft shadow computation. Especially techniques involving a large amount of work done on modern GPUs were desired. The goal was to compare (quality, performance on various types of scene settings, etc.) the results of implemented techniques and discuss their advantages and disadvantages.

Therefore I have decided to implement all of the chosen methods into a universal framework to allow comparison of their achievements interactively, with freedom to switch between the techniques. To do this the *OpenGL API*, *.cgfx* file format (ideal for having many different effect techniques (based on *shaders*) stored in one file) and *Cg runtime* were chosen. As a result of this, the framework is cross-platform. Price paid for this presentational level of the implementation of so many principally different methods is slightly lower performance (as too many bindings and variables have to be computed, not all of them important for current method, etc.). Higher emphasis was put on a universal platform for all the methods. Yet, it serves comparison purposes quite well.

4.2 Inspiration

I have chosen this topic for the thesis, as it was promising a lot of gained knowledge from the shadowing methods field of computer graphics. I am particularly interested in 3D computer graphics and so this sounded challenging, allowing me to study a lot of papers and to learn the most up-to-date programming techniques using GPU programmability.

As this is a highly dynamic field of CG, there are still huge opportunities for scientific work and also for future job positions. Computer graphics effects are used more and more extensively in movie industry, computer games development, CAD systems. Knowledge of various algorithms can only be an advantage in further professional life.

A huge challenge was to learn how to read papers and extract only the useful information from them. Over the time many algorithms for shadow computation were proposed. Many claims for interactive, realtime frame rates have been told. Yet almost every time there is some hidden disadvantage (interactive rates only on cluster of processors, works only for a very special scene case, only static scene, etc.). I was looking for algorithms as universal as possible, to be usable over the most scene settings and still have good performance ($\geq$20 fps). It is true that the increasing brute power of graphics chips makes all the methods faster over time. Still it is good to look for the compromise between quality and performance. Another problem connected with some published papers is the lack of the real implementation. All we see is an idea that looks good on the paper but encounters many problems connected with today's APIs (or the implementation requires specially designed, dedicated HW to run fast enough).

4.3 Problems encountered

Graphics application development is quite painful. It is mainly due to the inability of direct access into the buffers. Furthermore, everything depends on a complicated state automaton, where every switch, change can easily ruin code functionality elsewhere (you can always set everything in every function, but state changes are a bit time consuming (at least in OpenGL) to be used freely and too frequently). Therefore debugging is sometimes really complicated.

Shader development makes it even harder and brings quite a lot of additional drawbacks. The major one is the almost absolute absence of debugging possibilities. You do not have direct access to variables and parameters of the shaders. The only way how to evaluate partial results is to return them as a color information and manually try to understand the error, problem occurring.

A huge diversity of underlying graphics hardware is also important to mention. This means that for a commercial application many different shaders need to be written, as some profiles (with their improvements, as for example branching in fragment shader) are supported only on newer cards. And of course every manufacturer has its own profiles tailored precisely for slight differences in HW implementation.

In addition to this, my choice of Cg as a shader language was not the best one. It is still not fully mature and documentation is in some cases really poor or missing completely (or at least containing errors). Also the compiler error messages are a bit confusing. In some cases (technique state settings) they are missing completely.

During the writing of the code another unpleasant problem occurred. Performance of written shaders dropped drastically (in some cases up to 8 times) upon switching from Cg Toolkit 1.41 to Cg Toolkit 1.5! After some debugging the problem was identified as connected with many calls of `cgSetPassState()` and `cgResetPassState()` methods. Precompiled dynamic libraries included in this new version of the package slowed the execution of this code, probably due to some bug. Lately a new Cg Toolkit was released (1.5 February 2007 release) and solved these issues almost completely. It is a question, whether the development of Cg language will continue. If yes, I assume that all of the problems mentioned can be eradicated...

4.4 Improvements

During the actual coding and study of papers some minor flaws occurred to me. Therefore I have tried to implement some additional functionality to improve the performance of methods and visual quality of the results.

4.4.1 Variable area light

Almost every method for fake soft shadows is satisfied with just some blurring of the shadow edges. Yet it is absolutely not conforming to physics. Some more sophisticated methods (as PCSS method) take into account the area of light. This is much more precise, giving softness actually depending on the light geometry. Still, assumption is of a circular light. But what about more complicated shapes (some rectangular shape for example)? Here the ratio of side lengths has also to be taken into account (shadow is softer on the edges in the direction of longer light edge).

As shadow maps are built from the light position, offsets used for texture lookups cannot be purely

$$off_y = ratio * off_x \tag{4.1}$$

to cover successfully the desired variable softness according to the light elongation. We need function that will modify these offsets in such a way, that they will oscillate between values 1 and *ratio*. This can be achieved by simply using the `sine` and `cosine` functions. All we need is an angle between the light position (projected into the floor plane) and a vector

$$\vec{v} = (0,1,1) - (0,0,1)$$

pointing into the direction of axis y. The angle is then computed by applying *dot product* law.

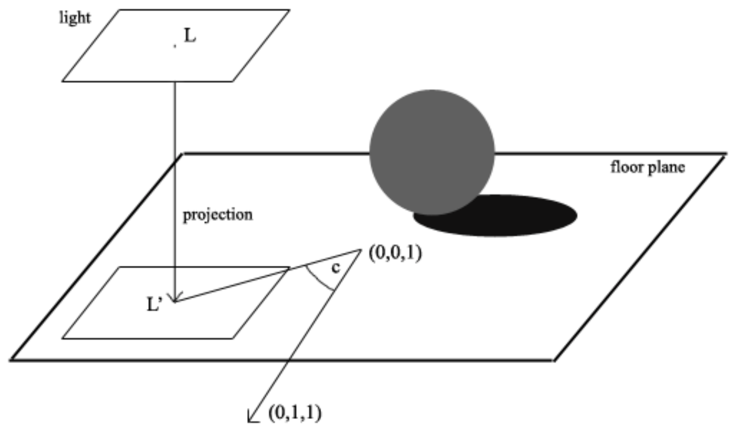


Figure 4.1: Offset angle computation

This way we get an approximation of elongation dependent variable softness. Resulting offset variables are finally computed as follows:

$$off_y = 1 + \sin(angle) * (ratio - 1) \tag{4.2}$$

$$off_x = 1 + \cos(angle) * (ratio - 1) \tag{4.3}$$

4.4.2 Midpoint shadow maps

The basic shadow maps (created by a pure copy of depth values from the depth buffer into a texture) suffer from a strong self shadowing alias (it can be attenuated by using `GL_POLYGON_OFFSET_FILL` but the settings are very empirical and scene settings dependent). Therefore another method for shadow map creation was proposed [Woo92].

It is based on the notion that the self shadowing artifacts occur because of the numerical errors cohering with depth values comparison (shadow maps from front facing polygons have artifacts on lightened surfaces and vice versa for back faces shadow maps). To avoid the closeness of the compared values, all we need to do is an averaging of two depths from the front and back face shadow maps.

Most methods are working with the classic shadow maps. Yet as the usage of midpoint shadow maps is equally fast, better results are achieved with them. Methods in the framework can use midpoint shadow maps as well as the classic ones. The only problem is a longer time needed to construct midpoint shadow map. Therefore they are more suitable for static scenes settings.

Remark: *The main flaw of midpoint shadow maps is an error when using objects with holes (not fully closed surfaces). Then the average depth value is wrong of course and artifacts can appear on resulting shadow. Yet it is not an issue for well defined objects.*

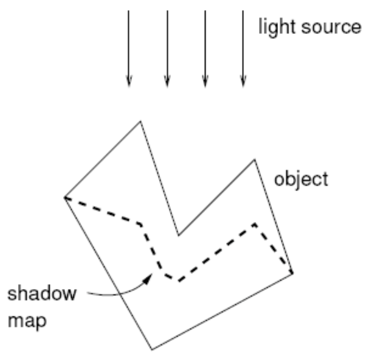
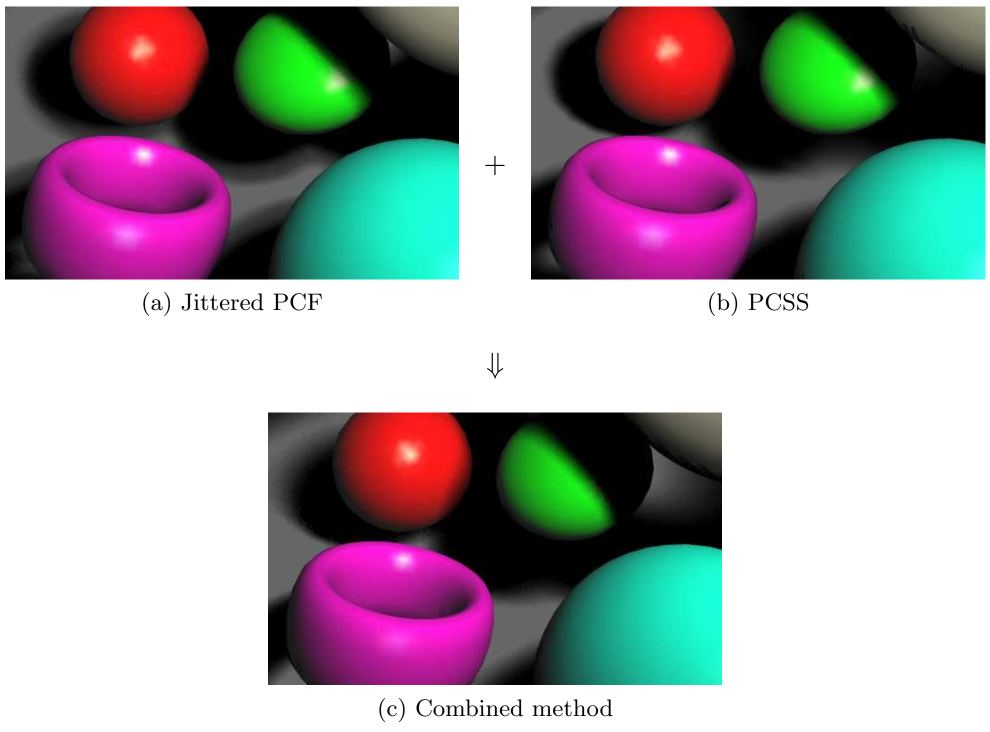


Figure 4.2: Midpoint shadow map explanation

4.4.3 Combined shadow map method

As I was experimenting with different soft shadow techniques, it occurred to me as useful to combine good characteristics of some methods and create their hybrid, giving better resulting quality with the comparable performance.



(a) Jittered PCF

(b) PCSS

(c) Combined method

Figure 4.3: Combined fake shadow map method

Two methods were ideal for a combination. Jittered shadow mapping with its great visual quality (swapping banding artifacts with noise, which is less irritating for a human perception) and PCSS with its physical dominance (variable penumbra size depending on the light-occluder-receiver geometry). In addition some experimenting with elongated rectangular light source improved resulting impression even more.

Whole method is therefore comprised from these steps:

1. **Blocker search** – shadow map is searched in a small neighborhood, retrieved depths are averaged to get the distance light-occluder
2. **Penumbra estimation** – penumbra width is estimated based on the light size, blocker and receiver distances from the light:

$$w_{penumbra} = (d_{receiver} - d_{blocker}) \cdot w_{light}/d_{blocker}$$

3. **Jittered filtering** (*use the computed penumbra size with combination of precomputed offsets to sample shadow map*)
   a) **Cheap sampling** – test only a small number of samples, if all are lying in the umbra or penumbra, no further testing is needed
   b) **Additional sampling** – if previous sampling is ambiguous, add other samples up to the full number

As can be visible by performance measurements, this method is as fast as PCSS with slightly better visual quality. Fragment shader involved can be written as follows:

Code 4.1: Combined shadow map method

```
float4 fragmentShadowMapCOMBINED(
            fpDataMap IN,
            uniform sampler2D shadowMap,
            uniform sampler2D shadowMap2,
            uniform sampler3D jitterTex,
            uniform float4 shadow,
            uniform float2 offset): COLOR
{
   //compute radius of neighbourhood for occluder scaning
   float receiver = IN.TexCoord0.z/IN.TexCoord0.w;;
   float2 radius = 4*offset*(flatSize/receiver);

   float occluder = 0.;
   float occluderCount = 0.;

   // PHASE I: compute radius (depends on scene geometry, distances)
   //coordinates for lookup jitter texture (prime to give better results)
   float3 coord = float3(IN.TexCoord0.x*17,IN.TexCoord0.y*13,0);

   float depthVal, isBlocker;

   //translate into last 16 samples
   coord.z = (jitterSamplesDiv2 - 8)*jitterInvSamples;

   //scan the circular neighbourhood for occluders
   for( int i=0; i<8;i++) {
      //sample lookup texture
      float4 off = tex3D(jitterTex, coord)*radius.xyxy;

      coord.z += jitterInvSamples;

      //2 offsets are stored in 1 value from jitterTex
      //A)
      depthVal = texPCF(shadowMap2,IN.TexCoord0,off.xy,true);

      isBlocker = step(depthVal, receiver);

      //update occluder count and stored depth
      occluder += depthVal*isBlocker;
      occluderCount += isBlocker;

      //B)
      depthVal = texPCF(shadowMap2,IN.TexCoord0,off.zw,true);

      isBlocker = step(depthVal, receiver);

      //update occluder count and stored depth
      occluder += depthVal*isBlocker;
      occluderCount += isBlocker;
   }

   //compute average of occluding depths
   occluder /= occluderCount;

   //estimation of penumbra size
   float penumbra = flatSize*(receiver-occluder)/occluder;

   radius = 5*offset*penumbra;

   //reverse changes to coordinates for lookup jitter texture
   coord.z = 0;

   float shadowHardness = 0.;
```

```
   // PHASE II : compute shadow color (jitter lookup)
   //cheap test samples (8)
   for( int i=0; i<4;i++) {
      //sample lookup texture
      float4 off = tex3D(jitterTex, coord)*radius.xyxy;

      coord.z += jitterInvSamples;

      //is in shadow? 2 offsets are stored in 1 value from jitterTex
      shadowHardness += texPCF(shadowMap,IN.TexCoord0,off.xy);
      shadowHardness += texPCF(shadowMap,IN.TexCoord0,off.zw);
   }

   //do we need to continue sampling (is penumbra hit highly probable)?
   if ((shadowHardness - 8) * shadowHardness != 0) {
      for (int i=0; i < jitterSamplesDiv2 - 4; i++) {
         //sample lookup texture
         float4 off = tex3D(jitterTex, coord)*radius.xyxy;

         coord.z += jitterInvSamples;

         //is in shadow? 2 offsets are stored in 1 value from jitterTex
         shadowHardness += texPCF(shadowMap,IN.TexCoord0,off.xy);
         shadowHardness += texPCF(shadowMap,IN.TexCoord0,off.zw);
      }

      //average value
      shadowHardness *= 0.5 * jitterInvSamples;
   }

   //return fragment color
   return float4(0,0,0,shadowHardness);
}
```

4.5 Further work

There is still much to do in the field of shadow volume algorithms. With the new generation of graphics chips and incoming *unified shaders* the programmability has been taken into a higher level again. Now it is possible to implement and use many acceleration structures on graphics chips directly, thus freeing the CPU from this workload (which is quite significant when using shadow volumes because of silhouette edges lookup, maintenance of shadow volume structures, etc.). This means that many of the algorithms used today can be transferred fully into GPU, therefore increasing their performance by a large amount (relieving both the CPU and the bus workflow). That will allow usage of shadow volumes for soft shadow computations in realtime applications. Maybe the most used contemporary methods (based on shadow maps) will be completely forgotten in the future...

With this in mind the shadow framework accompanying this thesis can be expanded to cover more volume based methods. New type of shaders is a thing too tempting to let go without testing its capabilities. Still, so far the development for this new generation of graphics cards is hindered by driver incompatibilities, low optimization and other system-driver-hardware based issues. Yet, in my opinion, it needs only some more time to become a mainstream in the development of shader based effects.

Chapter 5

Performance comparison

5.1 Introduction

To be able to compare quality and performance of all the implemented methods two different scenes were prepared. One with moderate number of triangles and also a huge scene (to see the performance level under extreme conditions).

Shadow methods work upon the full level of detail objects, whereas in contemporary applications shadows are computed mainly using the coarse models with low number of polygons (typical game of these days has to display up to 500,000 triangles per frame, yet to compute the shadows only a fraction of them is used, thus explaining the relevance of further measurements).

Many different scene attributes are important (static, dynamic light, moving objects, number of light samples, complexity of models, etc.). Different techniques are sensitive to different settings.

5.2 Basic scene

Following tables show some comparison of the results measured on a medium sized scene. It is clear, which method's performance is highly dependent on the scene size, on the resolution, eventually on the number of light position samples.

Performance is stated in frames per second as well as in milliseconds needed for the scene to be drawn. Differences in the scene settings and their short explanation follow:

- resolution – 700x700 or 900x900
- faces (triangles) – total number of 17408
- samples – number of light samples, 64 or 100

- dynamic light – static or dynamic light (increases computation complexity)

Brute force soft shadow methods are very sensitive to every increase in the screen resolution, number of light samples and also number of faces in the scene. In addition, 64 samples is the least number giving still only modest looking results. Quality is increasing with higher numbers of samples but the performance drops quickly.

Fake shadow methods are on the other hand sensitive mainly to the resolution (shadow map methods require more fragment shader calls) or a geometry complexity (shadow volume, projective shadows). Quality is much better (even though not fully physically correct) and performance remains better.

Remarks:

- ***shadow volume methods*** *– no acceleration structures*
- ***depth pass, depth fail, XOR*** *– brute force soft shadow*
- ***projective soft*** *– brute force soft shadow*
- ***projective fake*** *– PCF7x7 on a saved projective shadow texture*
- ***shadow map /b/*** *– basic shadow maps, self shadow artifacts, HW presampling*
- ***shadow map /m/*** *– midpoint shadow maps, lower dynamic performance, no HW presampling*
- ***map soft*** *– brute force soft shadow*
- ***map pcf*** *– PCF7x7*
- ***map blur*** *– PCF3x3, gaussian blur (15 point filter) 2x*
- ***map jitter*** *– PCF 64 disk samples, stratified jitter of offsets*
- ***map pcss*** *– 5x5 penumbra evaluation, PCF7x7 with penumbra radius*
- ***map combined*** *– 4x4 penumbra, PCF 32 stratified disk with penumbra radius*

Methods based on shadow maps use two different approaches. Basic shadow map (based on depth buffer information only) is a fast way, how to create, maintain and use shadow mapping. However, it brings some artifacts (self shadow, mainly because of z-fighting, inaccuracy of numerical precision). They can be dealt with by polygon offsets but their correct setting is empirical and very dependent on the current scene. Yet this is a way how to do soft shadowing in dynamic scenes.

Method		Dyn. part (ms)	700x700				900x900			
			static		*dynamic*		*static*		*dynamic*	
			fps	**ms**	**fps**	**ms**	**fps**	**ms**	**fps**	**ms**
Basic scene	–	0	*1121.21*	0.89	*1121.21*	0.89	*730.16*	1.37	*730.16*	1.37
Shadow volume	***visualization***	0	*610.00*	1.64	*610.00*	1.64	*437.50*	2.29	*437.50*	2.29
	hard	0	*387.76*	2.58	*387.76*	2.58	*276.02*	3.62	*276.02*	3.62
	depth pass	0	*16.21*	61.69	*16.21*	61.69	*11.33*	88.26	*11.33*	88.26
	depth fail	0	*10.00*	100.00	*10.00*	100.00	*6.80*	147.06	*6.80*	147.06
	XOR	0	*10.28*	97.28	*10.28*	97.28	*7.01*	142.65	*7.01*	142.65
Projective shadow	***hard***	0	*687.86*	1.45	*687.86*	1.45	*453.87*	2.20	*453.87*	2.20
	soft	0	*56.54*	17.69	*56.54*	17.69	*38.37*	26.06	*38.37*	26.06
	fake	0	*77.88*	12.84	*77.88*	12.84	*50.86*	19.66	*50.86*	19.66
Shadow map	***visualization /b/***	2	*759.04*	1.32	*301.44*	3.32	*567.21*	1.76	*265.74*	3.76
	visualization /m/	13	*862.07*	1.16	*70.62*	14.16	*625.00*	1.60	*68.49*	14.60
	hard /b/	2	*540.07*	1.85	*259.63*	3.85	*378.79*	2.64	*215.52*	4.64
	hard /m/	13	*580.84*	1.72	*67.93*	14.72	*422.36*	2.37	*65.07*	15.37
	soft /b/	116	*27.61*	36.22	*6.57*	152.22	*18.06*	55.37	*5.84*	171.37
	soft /m/	171	*30.00*	33.33	*4.89*	204.33	*20.18*	49.55	*4.53*	220.55
	fake (pcf) /b/	2	*28.34*	35.29	*26.82*	37.29	*19.01*	52.60	*18.31*	54.60
	fake (pcf) /m/	13	*34.87*	28.68	*23.99*	41.68	*23.71*	42.18	*18.12*	55.18
	fake (blur) /b/	2	*85.13*	11.75	*72.74*	13.75	*81.01*	12.34	*69.71*	14.34
	fake (blur) /m/	13	*88.77*	11.27	*41.21*	24.27	*82.78*	12.08	*39.87*	25.08
	fake (jitter) /b/	2	*26.75*	37.38	*25.39*	39.38	*18.01*	55.52	*17.38*	57.52
	fake (jitter) /m/	13	*29.61*	33.77	*21.38*	46.77	*19.98*	50.05	*15.86*	63.05
	fake (pcss) /b/	4	*18.89*	52.94	*17.56*	56.94	*12.67*	78.93	*12.06*	82.93
	fake (pcss) /m/	22	*26.45*	37.81	*16.72*	59.81	*17.79*	56.21	*12.79*	78.21
	fake (combined) /b/	4	*31.00*	32.26	*27.58*	36.26	*21.47*	46.58	*19.77*	50.58
	fake (combined) /m/	22	*25.83*	38.71	*16.47*	60.71	*17.73*	56.40	*12.75*	78.40

Table 5.1: Performance comparison (17,408 triangles, 64 light samples)

Method		Dyn. part (ms)	700x700				900x900			
			static		***dynamic***		***static***		***dynamic***	
			fps	**ms**	**fps**	**ms**	**fps**	**ms**	**fps**	**ms**
Basic scene	–	0	*1113.07*	0.90	*1113.07*	0.90	*725.81*	1.38	*725.81*	1.38
Shadow volume	***visualization***	0	*616.82*	1.62	*616.82*	1.62	*426.38*	2.35	*426.38*	2.35
	hard	0	*397.52*	2.52	*397.52*	2.52	*275.86*	3.63	*275.86*	3.63
	depth pass	0	*10.37*	96.43	*10.37*	96.43	*7.17*	139.47	*7.17*	139.47
	depth fail	0	*6.19*	161.55	*6.19*	161.55	*4.27*	234.19	*4.27*	234.19
	XOR	0	*6.59*	151.75	*6.59*	151.75	*4.50*	222.22	*4.50*	222.22
Projective shadow	***hard***	0	*651.38*	1.54	*651.38*	1.54	*446.81*	2.24	*446.81*	2.24
	soft	0	*37.33*	26.79	*37.33*	26.79	*25.16*	39.75	*25.16*	39.75
	fake	0	*72.57*	13.78	*72.57*	13.78	*50.89*	19.65	*50.89*	19.65
Shadow map	***visualization /b/***	2	*801.28*	1.25	*307.88*	3.25	*567.77*	1.76	*265.87*	3.76
	visualization /m/	13	*823.94*	1.21	*70.35*	14.21	*599.01*	1.67	*68.17*	14.67
	hard /b/	2	*542.62*	1.84	*260.22*	3.84	*377.22*	2.65	*215.01*	4.65
	hard /m/	13	*612.35*	1.63	*68.34*	14.63	*430.00*	2.33	*65.25*	15.33
	soft /b/	182	*17.67*	56.59	*4.19*	238.59	*11.56*	86.51	*3.72*	268.51
	soft /m/	266	*19.32*	51.76	*3.15*	317.76	*13.06*	76.57	*2.92*	342.57
	fake (pcf) /b/	2	*29.17*	34.28	*27.56*	36.28	*19.40*	51.55	*18.68*	53.55
	fake (pcf) /m/	13	*35.71*	28.00	*24.39*	41.00	*23.93*	41.79	*18.25*	54.79
	fake (blur) /b/	2	*84.67*	11.81	*72.41*	13.81	*80.78*	12.38	*69.54*	14.38
	fake (blur) /m/	13	*88.17*	11.34	*41.08*	24.34	*82.63*	12.10	*39.84*	25.10
	fake (jitter) /b/	2	*27.11*	36.89	*25.72*	38.89	*18.39*	54.38	*17.74*	56.38
	fake (jitter) /m/	13	*29.48*	33.92	*21.31*	46.92	*19.85*	50.38	*15.78*	63.38
	fake (pcss) /b/	4	*19.41*	51.52	*18.01*	55.52	*12.47*	80.19	*11.88*	84.19
	fake (pcss) /m/	22	*27.32*	36.60	*17.06*	58.60	*17.44*	57.34	*12.60*	79.34
	fake (combined) /b/	4	*31.66*	31.59	*28.10*	35.59	*21.62*	46.25	*19.90*	50.25
	fake (combined) /m/	22	*25.63*	39.02	*16.39*	61.02	*17.34*	57.67	*12.55*	79.67

Table 5.2: Performance comparison (17.408 triangles. 100 light samples)

On the other hand we can use midpoint shadow maps. Their creation is slower, yet the resulting shadow lacks (almost completely) annoying shadow errors. The principle is a combination of two shadow maps (back facing and front facing), thus getting intermediate depth map.

5.3 Huge scene

In this case the performance was measured against a big scene ($\geq$100,000 triangles). Again, the results show comparison between methods in the static and dynamic environment. As we can see, some methods are not suitable for a real-time, interactive usage (games, CAD applications preview, etc.).

Remark: *For this test the scene was composed from the well known Stanford bunny models*[1]*. 7 downsampled models (16,301 faces each) were placed into the scene with different rotation, scale and translation.*

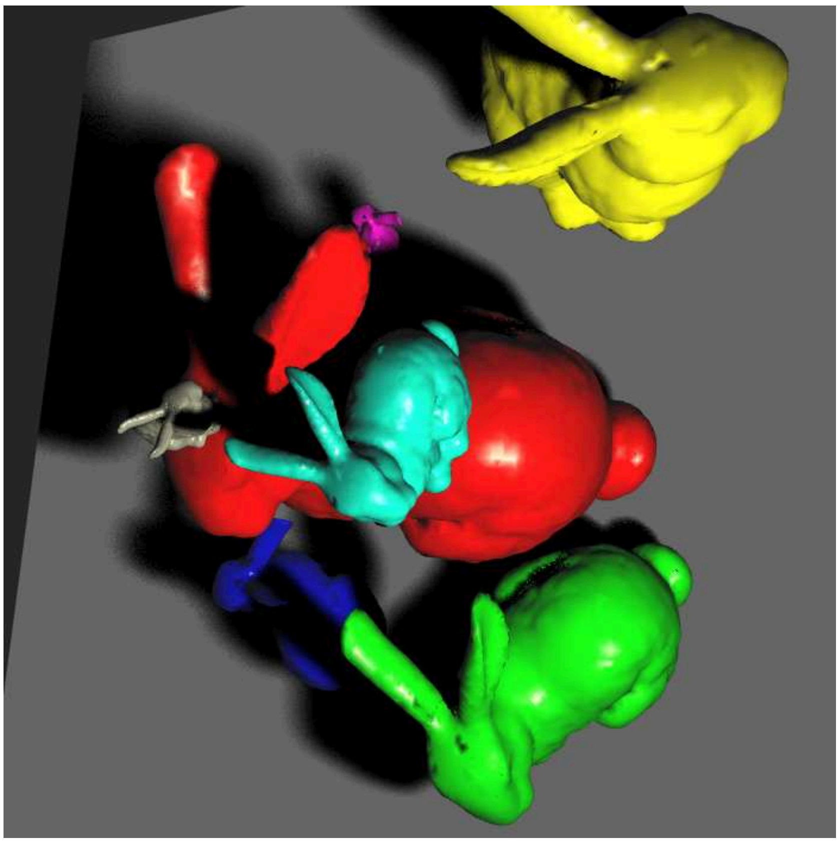


Figure 5.1: Huge scene

[1]freely downloadable from The Stanford 3D Scanning Repository, http://graphics.stanford.edu/data/3Dscanrep/

Method		Dyn. part (ms)	700x700				900x900			
			static		*dynamic*		*static*		*dynamic*	
			fps	ms	fps	ms	fps	ms	fps	ms
Basic scene	–	0	*255.40*	3.92	*255.40*	3.92	*220.88*	4.53	*220.88*	4.53
Shadow volume	***visualization***	0	*133.41*	7.50	*133.41*	7.50	*115.00*	8.70	*115.00*	8.70
	hard	0	*72.16*	13.86	*72.16*	13.86	*62.20*	16.08	*62.20*	16.08
	depth pass	0	*1.95*	512.82	*1.95*	512.82	*1.76*	568.18	*1.76*	568.18
	depth fail	0	*1.55*	645.16	*1.55*	645.16	*1.31*	763.36	*1.31*	763.36
	XOR	0	*1.79*	558.66	*1.79*	558.66	*1.46*	684.93	*1.46*	684.93
Projective shadow	***hard***	0	*151.08*	6.62	*151.08*	6.62	*130.60*	7.66	*130.60*	7.66
	soft	0	*6.47*	154.56	*6.47*	154.56	*6.10*	163.93	*6.10*	163.93
	fake	0	*54.16*	18.46	*54.16*	18.46	*39.43*	25.36	*39.43*	25.36
Shadow map	***visualization /b/***	6	*229.89*	4.35	*96.62*	10.35	*204.61*	4.89	*91.85*	10.89
	visualization /m/	19	*238.01*	4.20	*43.10*	23.20	*208.19*	4.80	*42.01*	23.80
	hard /b/	6	*150.99*	6.62	*79.22*	12.62	*130.20*	7.68	*73.10*	13.68
	hard /m/	19	*149.94*	6.67	*38.96*	25.67	*132.57*	7.54	*37.67*	26.54
	soft /b/	228	*6.17*	162.07	*2.56*	390.07	*5.44*	183.82	*2.43*	411.82
	soft /m/	428	*6.35*	157.48	*1.71*	585.48	*5.78*	173.01	*1.66*	601.01
	fake (pcf) /b/	6	*14.81*	67.52	*13.60*	73.52	*11.10*	90.09	*10.41*	96.09
	fake (pcf) /m/	19	*20.40*	49.02	*14.70*	68.02	*15.43*	64.81	*11.93*	83.81
	fake (blur) /b/	6	*51.88*	19.28	*39.56*	25.28	*52.03*	19.22	*39.65*	25.22
	fake (blur) /m/	19	*57.49*	17.39	*27.48*	36.39	*54.44*	18.37	*26.76*	37.37
	fake (jitter) /b/	6	*13.34*	74.96	*12.35*	80.96	*9.80*	102.04	*9.26*	108.04
	fake (jitter) /m/	19	*15.36*	65.10	*11.89*	84.10	*11.35*	88.11	*9.34*	107.11
	fake (pcss) /b/	9	*11.63*	85.98	*10.53*	94.98	*8.42*	118.76	*7.83*	127.76
	fake (pcss) /m/	28	*16.18*	61.80	*11.14*	89.80	*11.81*	84.67	*8.88*	112.67
	fake (combined) /b/	9	*16.70*	59.88	*14.52*	68.88	*12.32*	81.17	*11.09*	90.17
	fake (combined) /m/	28	*14.33*	69.78	*10.23*	97.78	*10.66*	93.81	*8.21*	121.81

Table 5.3: Performance comparison (114,107 triangles, 64 light samples)

5.4 Quality differences

Implemented methods differ not only in performance (frame rates), resulting images have also various visual quality, physical correctness. Following table recapitulates achievements of different methods. Resulting images are being compared with a paragon image (physically correct shadow achieved by sampling the light with $\geq$1,000 samples). Degree (percentage) of correctness is evaluated with a pixel per pixel comparison of the images.

Remarks:

1. *physically correct soft shadow generated by (oversampled)* ***depth fail shadow volume*** *method was used as a paragon*
2. ***performance*** *is taken from the 700x700, static scene with 64 light samples*
3. ***quality*** *takes into account performance, physical correctness and presence of visual artifacts*

Type	**Method**	**Performance (fps)**	**Correctness (%)**	**Quality (●)**
Basic scene	*No shadow*	1121.21	57.193	○○○○○○○○○○
Shadow volume	*Hard shadow*	387.76	88.113	○○○○○○○●●●
	Depth pass	16.21	99.939	○○○○○●●●●●
	Depth fail	10.00	99.942	○○○○●●●●●●
	XOR	10.28	94.078	○○○○○○○●●●
Projective shadow	*Hard shadow*	687.86	88.348	○○○○○○○○○●
	Soft shadow	56.54	95.759	○○○○○○○●●●
	Fake shadow	77.88	92.288	○○○○○○○●●●
Shadow map	*Hard shadow /b/*	540.07	89.251	○○○○○○○○●●
	Hard shadow /m/	580.84	88.770	○○○○○○○○○●
	Soft shadow /b/	27.61	99.549	○○○○●●●●●●
	Soft shadow /m/	30.00	99.416	○○○●●●●●●●
	Fake (pcf) /b/	28.34	90.699	○○○○○○●●●●
	Fake (pcf) /m/	34.87	90.710	○○○○○●●●●●
	Fake (blur) /b/	85.13	90.699	○○○○○●●●●●
	Fake (blur) /m/	88.77	90.710	○○○○●●●●●●
	Fake (jitter) /b/	26.75	93.925	○○○●●●●●●●
	Fake (jitter) /m/	29.61	94.008	○○●●●●●●●●
	Fake (PCSS) /b/	18.89	94.994	○○●●●●●●●●
	Fake (PCSS) /m/	26.45	95.756	○●●●●●●●●●
	Fake (combined) /b/	31.00	94.560	○●●●●●●●●●
	Fake (combined) /m/	25.83	95.325	●●●●●●●●●●

Table 5.4: Quality comparison of the implemented methods

Chapter 6

Conclusion

We have been witnessing a huge hardware development boom over the last few years. Along with increasing brute power of graphics chips, their programmability was introduced as well (thus replacing fixed rendering pipeline). Higher performance and more flexibility in programming enabled many algorithms to breach the borders of realtime applications. It naturally brought a better visual quality into this important segment of the market.

Among the frequently used computer graphics algorithms a very significant role is held by shadow computation methods. They are essential for human perception (their lack is immediately noticed by every user, thus flawing the overall impression). Furthermore, soft versions of shadows (the only physically correct ones) are also very computationally expensive.

Over the years three major types of shadow computation algorithms were proposed and are still under further development (many papers are suggesting improvements, comprising performance boosts or quality enhancement). These are projective shadows, shadow maps and shadow volumes. Each of them is useful in some situations, none is the absolute winner. They differ in resulting quality of a shadow, performance and robustness (presence of artifacts, commonness of the scene used, etc.). Variances can be found even in the dependence on CPU implementation (thus disabling pure GPU execution). Advantages and disadvantages are mentioned over the course of the thesis.

However, the main goal was to study, implement and compare multiple methods of soft shadow computation under various conditions. Their mainly GPU implementation (with the use of shaders and modern graphics hardware) and high efficiency were desirable. To make the comparison easier, a universal framework was created. The included demonstrative software enables on-the-fly familiarization with the mentioned methods, interactive comparison of the efficiency and realism. All of this can be performed under multiple scene settings (static, dynamic; medium sized or huge scene; etc.).

Furthermore, the thesis covers topics from explanation of basic algorithms, through advanced soft shadowing methods, to measurements of both quality and performance of a couple of algorithms under varying conditions. According to the resulting data, the most universal methods appear to be from the class of shadow mapping techniques (PCSS, jittered PCF, their combination), presenting good overall performance and high degree of physical correctness. Moreover, they are well suited for contemporary hardware, enabling (almost) pure GPU implementation. This cannot be stated about algorithms based on shadow volumes, where huge precomputation (and CPU bound) steps are needed.

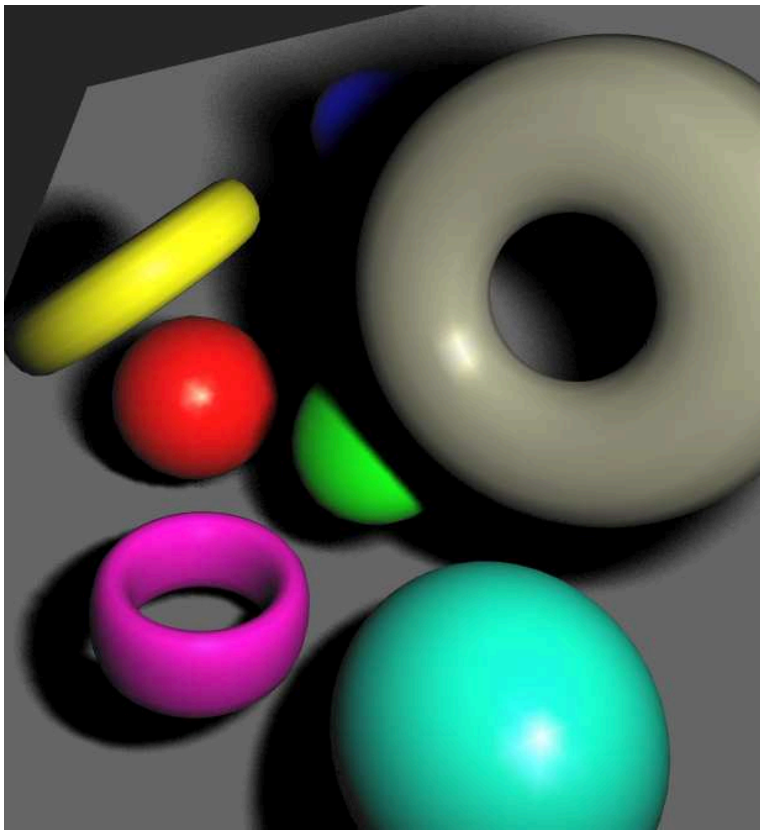


Figure 6.1: Soft shadows (combined method)

This work tries to cover a blind spot on the computer graphics map. There exist plenty of papers suggesting the best methods for soft shadow computation. They include some results, yet their comparison is problematic because of different HW configurations, various scene data, etc. The shadow framework is a single platform with multiple implemented methods, allowing to compare their performance and visual quality under uniform settings.

Along with the usage of well or less known algorithms, some improvements, minor updates and fixes were proposed during the development of this paper. This includes rectangular (elongated) area light, wide usage of midpoint shadow maps and combination of advantages of jittered and PCSS methods resulting in a higher physical correctness with a comparable performance level.

Method	**Performance (fps)**	**Quality (●)**
Shadow volume (depth fail)	10.00	○○○○●●●●●●
Shadow map	30.00	○○○●●●●●●●
Jittered PCF	29.61	○○●●●●●●●●
PCSS	26.45	○●●●●●●●●●
Combined method	25.83	●●●●●●●●●●

Table 6.1: Methods comparison excerpt (64 samples)

On the other hand, the most sophisticated soft shadow methods, using shadow volumes, are not covered yet. Their integration into the framework is not straightforward as they require many non trivial steps (silhouette edges lookup, etc.) done on the CPU. Still, with the latest development in the graphics hardware (new generation of chips, unified shaders) it is becoming possible to execute many of these computations on the side of the GPU. This could be the way to expand my work in the future.

~

All the buzz in this scientific field in the last years enabled quick intrusion of shadows into realtime applications. People are getting used to see realistic shadows in virtual environments. And now the time for something more has come, for the whole new level of realism using soft shadows. Algorithms for their computation are gaining strength and are almost ready to be used worldwide. Be prepared!

References

[BB99] Mike Songy Bill Bilodeau. Real time shadows. *Creative Labs Inc. Sponsored game developer conferences, Creativity 1999*, 1999.

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

[C.77] Crow F. C. Shadow algorithms for computer graphics. *SIGGRAPH 1977*, 1977.

[Car00] J. Carmack. e-mail to private list. *published online on http://developer.nvidia.com*, 2000.

[Ebe06] David H. Eberly. *3D Game Engine Design. A Practical Approach to Real-Time Computer Graphics (2nd edition).* Morgan Kaufmann, 2006.

[Fer03] Randima Fernando. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics.* Addison-Wesley Pub Co., 2003.

[Hei91] T. Heidmann. Real shadows, real time. *Iris Universe*, 18:28–31, 1991.

[HJF03] Holzschuch N. Hasenfratz J., Lapierre M. and Sillion F. A survey of real-time soft shadow algorithms. *ComputerGraphicsForum*, 22(4), 2003.

[L.78] Williams L. Casting curved shadows on curved surfaces. *Computer Graphics (SIGGRAPH 1978Proceedings)*, pages 270–274, 1978.

[Len02] Eric Lengyel. *Mathematics for 3D Game Programming and Computer Graphics.* Charles River Media, 2002.

[Mik07] Miroslav Mikšík. Generating false soft shadows in real-time. *study assignment*, 2007.

[MP04] Greg Humphreys Matt Pharr. *Physically Based Rendering: From Theory to Implementation.* Morgan Kaufmann, 2004.

[MS02] G. Drettakis M. Stamminger. Perspective shadow maps. *ACM Transactions on Graphics*, 21(3):557–562, 2002.

[Ran05] Fernando Randima. Percentage-closer soft shadows. *nVIDIA Corporation*, 2005.

[RSWJ04] Benjamin Lipchak Richard S. Wright Jr. *OpenGL Superbible (3rd edition).* Sams, 2004.

[RWR87] Salesin D.H. Reves W.T. and Cook R.T. Rendering antialiased shadows with depth maps. *Computer Graphics (SIGGRAPH 1987Proceedings)*, pages 283–291, 1987.

[Sha05] Anirudh.S Shastry. Soft-edged shadows. *published online on http://www.gamedev.net*, 2005.

[TAM02a] Ulf Assarsson Tomas Akenine-Mller. Approximate soft shadows on arbitrary surfaces using penumbra wedges. *ACM International Conference Proceeding Series*, 28, 2002.

[TAM02b] Eric Haines Tomas Akenine-Moller. *Real-Time Rendering (2nd edition).* A K Peters Ltd., 2002.

[Ura05] Yury Uralsky. Efficient soft-edged shadows using pixel shader branching. *GPU Gems 2*, pages 269–282, 2005.

[Woo92] A. Woo. The shadow depth map revisited. *Graphics Gems III*, pages 338–342, 1992.

Appendices

Appendix A

Implementation details

A.1 Used technologies

(a) *n*VIDIA (b) OpenGL (c) Cg

Figure A.1: Core technologies

For the core part of the framework OpenGL API with the connection of C++ (a great free IDE is Microsoft's Visual Studio C++ Express) was chosen. It allows the application to run cross-platform. This is an advantage over Microsoft's DirectX API (despite its slightly better performance on Windows).

In addition GLUT libraries were used. After some time inevitability of various OpenGL extensions (`EXT_framebuffer_object`) occurred and so the extension wrap up library GLEW needed to be added.

The part of the shaders was covered with Cg Toolkit from *n*VIDIA. It offers API for connection with both OpenGL and Direct3D. All of the major shader formats (Cg, GLSL, HLSL) are supported along with the most shader profiles. A great addition is the CgFX files support allowing to create files with techniques (multipass shader effects). Another nice feature is Cg Runtime, allowing the runtime compilation (a bit slower but can profit from the future Cg compiler versions).

Short overview along with the hardware involved (has a great influence on the performance results) follows:

- **Hardware**
  - CPU AMD Athlon64 3200+
  - GPU nVIDIA GeForce7600GT (ForceWare 93.71)
  - 1024MB RAM
- **Software**
  - Visual Studio Express C++
  - OpenGL 2.1
  - GLUT 3.7.6
  - GLEW 1.3.6
  - Cg Toolkit 1.5 (February 2007 revision)

A.2 Code overview

As was already mentioned before, code is C++ based using OpenGL API for fixed graphics pipeline computations and Cg language for shaders (programmable pipeline). Because of the use of advanced shader profiles (`fp40`, `vp40`) and multiple OpenGL extensions (for example `EXT_framebuffer_object`) a newer HW (GeForce6+) is required to run correctly.

Code is divided into the framework (C++) and effects (.cgFX) part. Framework brings a basic functionality, initializations, scene parsing (from external files) and drawing pipeline handling. GPU makes use of various techniques (consisting of multiple shaders) from the effects file in every drawing cycle.

Code structure:

- ***Framework***
  - *common.h* – I/O functions, .OBJ parser, snapshots
  - *constants.h* – global constants, bindings for Cg
  - *gpuShadows.cpp* – GLUT default functions
  - *projectiveShadow.h* – functions for projective shadows computation
  - *scene.h* – scene drawing, initialization, display lists creation
  - *shadowMap.h* – functions for shadow maps computation
  - *shadowVolume.h* – functions for shadow volumes computation
- ***Shaders***

  – *shadows.cgfx* – techniques, samplers, shaders used by framework

Let me further explain the workflow of this application. As it is based on GLUT, at the beginning some initialization is made (GLUT settings, Cg context creation and .cgfx file linkage, techniques validation, frame buffer objects, context menu, parsing of the scene, shadow maps precomputation, etc.) and afterwards a main loop is started (it takes care of all the computations and drawing into the buffers). And of course some memory freeing is scheduled upon exiting the application.

Application workflow in more details:

1. **Basic initialization:** GLUT window creation, callbacks registration, display lists buildup (parsing of external .obj files), textures generation, frame buffer objects initialization, context menu definition, lighting settings
2. **Cg initialization:** Cg context, effect creation, optimal profile options setting, techniques validation, setup of bindings (parameters, samplers)
3. **Static part precomputation:** constants used in fake shadows computation, shadow maps, projection matrices - *in case of a dynamic scene has to be reevaluated after each light or objects movement*
4. **Main drawing loop:**
   a) timer reset, buffers cleanup
   b) initial transformation settings
   c) appropriate drawing pipeline call (according to the technique currently used), takes care of basic scene drawing, shadow computation and also shadow displaying
   d) performance update, buffers swap
5. **Cleaning:** textures and frame buffer object deletion, Cg context, Cg effect destruction

A.3 Framework - common.h

This header file contains mainly functions for I/0 operations. Further description of the most important ones follows:

```
parseOBJ(std::vector<float> &vertices, ... char *fileName);
```

Scene data (shadow casting objects) are stored in the external files. File format is a basic OBJ (text format, can include information about vertices, normals, texture coordinates, faces, materials, objects...).

```
#we need only information about vertices, normals and faces
#other lines are ignored

#vertices (x,y,z)
v 4.134725 4.805875 -5.847136
v 4.616718 2.659916 -6.807846
v 2.881218 4.728204 -7.401609

...

#normals (x,y,z)
vn 0.962615 0.255867 -0.088595
vn 0.924131 0.343944 -0.166173
vn 0.936583 0.350414 0.000000

...

#faces (vertex/texture_coord/normal ...)
f 283//10 13//11 164//12
f 164//12 405//8 283//10
f 103//2 283//10 405//8

...
```

Figure A.2: OBJ file example

File is parsed line by line and `std::vector<float>` containers are built. Lines with face info contain identifications of vertices and normals (their sequential number), lines can be in any particular order. That is the reason why we need to parse the whole file and build arrays of vertices, normals and faces, before we can start building display lists (OpenGL).

Remark: Models were exported with the help of Blender[1] modeling software.

`takeSnapshot(int width, int height, char* fileName);`

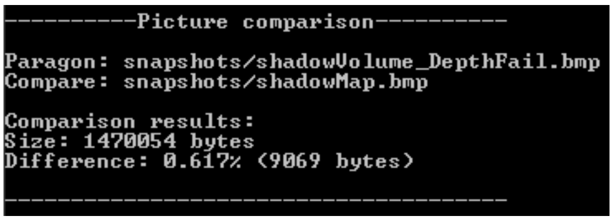
Function captures current framebuffer content into *.bmp* (bitmap) image (with correct header settings). `glReadPixels()` has to be called with `GL_BGR` as an internal format because bitmap stores color channels in swapped order. Snapshots are needed for quality comparison.

Remark: Function uses *Microsoft specific* type definitions and therefore is only applicable on computers running the Windows system.

`compareResults(char* paragon,char* compare);`

[1]freeware, http://www.blender.org

Function compares two given pictures (in *.bmp* format), paragon (is supposed to be the physically correct shadow) and image, quality of which we want to evaluate. Bitmaps are compared pixel by pixel and differences are accumulated (with a small difference allowed). Comparison result is printed out.



```
----------Picture comparison----------

Paragon: snapshots/shadowVolume_DepthFail.bmp
Compare: snapshots/shadowMap.bmp

Comparison results:
Size: 1470054 bytes
Difference: 0.617% (9069 bytes)

--------------------------------------
```

Figure A.3: Picture quality comparison

A.4 Framework - scene.h

This header file contains mainly functions covering basic drawing of objects and display lists preparation. Further description of the most important ones follows:

`drawWindowRect();`

Special auxiliary function, used in some fake shadow methods, when we need to combine textures (for example midpoint shadow maps generation). Draws textured rectangle over the whole window.

`buildSolid(int num, GLfloat color[4], bool huge = false);`

Builds display list for given solid with chosen color. Solid is parsed with `parseOBJ()` function. Afterwards the display list is built from resulting vertices, normals and faces arrays. It is called in the initialization phase, as it is relatively slow. Yet display lists enable precompilation of whole objects and therefore a faster display of the geometry later.

`drawFlatLight(GLfloat lightPosition[]);`

Draws area light representation (for presentational purposes, shows stochastic sampling of the light).

`basicScene(bool light = true);`

Draws scene (prepared display lists). This is done without the usage of shaders as it does not need any intermediate computations on the vertex, fragment levels.

`initLights();`

Initializes positions of area light samples. Stochastic jittered sampling is used to avoid banding and aliasing artifacts as much as possible. These samples are used for brute force soft shadow algorithms (still, really large number of samples is needed for the result without artifacts, ideally $\geq$1024, which makes these methods unusable in any realtime application).

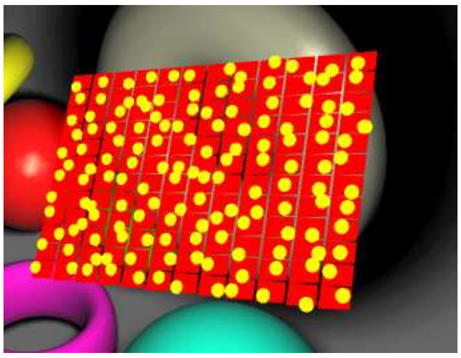


Figure A.4: Stochastic sampling

`recomputeOffsets();`

Function recomputes offset coefficients used for texture lookups (shadow mapping). Area light is not strictly square. Ratio of side lengths has to be taken into account (shadow is softer on the edges in the direction of the longer light edge).

A.5 Framework - gpuShadows.cpp

This file contains mainly functions concerning basic GLUT settings (callbacks, main display loop) and also CgFX initialization. Further description of the most important ones follows:

`initCgFX();`

Function takes care of CgFX instantiation (context creation, effect file parsing), techniques validation and setting of all shader-framework bindings (parameters, textures) using Cg libraries.

Code A.1: initCgFX() extract

```
//create context and set optimal compiler options
context = cgCreateContext();
cgGLSetOptimalOptions(CG_PROFILE_VP40);
cgGLSetOptimalOptions(CG_PROFILE_FP40);

//parse effect file
```

```
effect = cgCreateEffectFromFile(context, "shadows.cgfx", NULL);
if (!effect) {
  fprintf(stderr, "Unable to create effect!\n");
  exit(1);
}

//validate techniques
technique = cgGetFirstTechnique(effect);
while (technique) {
  if (cgValidateTechnique(technique) == CG_FALSE)
      fprintf(stderr, "Technique %s did not validate.  Skipping.\n",
              cgGetTechniqueName(technique));
  technique = cgGetNextTechnique(technique);
}

//create bindings and set correct values
CGparameter flatsize = cgGetNamedEffectParameter(effect, "flatSize");
assert(flatsize);
cgSetParameter1i(flatsize, flatSize);

...

//initialize sampler for shadow maps
shadowMap = cgGetNamedEffectParameter(effect, "shadowMap");
cgGLSetupSampler(shadowMap, shadowMapHQ);
```

`Display();`

This is the main rendering loop function, called after each swapping of frame buffers. It sets basic transformations, calls required rendering pipeline according to the actual technique and takes care of performance computation and displaying.

Code A.2: Display() extract

```
//get current technique number (type)
CGannotation ann = cgGetNamedTechniqueAnnotation(technique, "type");
if (ann) {
  const int *vals = cgGetIntAnnotationValues(ann, &count);
  assert(count == 1);
  type = *vals;
}

//call appropriate function for shadow computation
switch(type) {
   case MAP_VISUALIZATION : pipelineMapVisualization(); break;
   case MAP_HARD : pipelineMap(true); break;
   case MAP_FAKE : pipelineFakeMap(); break;
   ...
   default : basicScene(); break;
}
```

`main(int argc, char *argv[]);`

Heart of the program, initialization of GLUT, GLEW, CgFX, texture creation. Registers all GLUT callbacks, creates context menu, precomputes data for a static scene (shadow maps, projection matrices, offsets). Because of the need to use high resolution shadow maps (1024x1024) a frame buffer object is created and verified. Also the lighting settings are done here. At the end the main rendering loop is entered. After its termination some memory cleanup (textures, context, etc.) is scheduled.

A.6 Framework - projectiveShadow.h

This header file contains mainly functions concerning projective shadows (creation, drawing). Further description of the most important ones follows:

`shadowMatrix(float shadow[4][4],float floor[4],float light[4]);`

Calculates matrix for projection of the geometry into the floor plane. More information can be found in the chapter 2.2.2.

`cgProjective(float position[], int lightNum);`

This function encapsulates all the necessary calls for shaders, sets variable bindings and calculates shadow.

Code A.3: cgProjective() extract

```
//set transformation matrix for shadow projection
cgSetMatrixParameterfc(shadowP,(float*)&floorShadow[lightNum]);

//call required passes of current technique
CGpass pass = cgGetFirstPass(technique);
while (pass) {
   cgSetPassState(pass);
   drawSolids(true);
   cgResetPassState(pass);
   pass = cgGetNextPass(pass);
}
```

`pipelineProjective(bool hard = false);`
Pipeline for a projective shadow computation and drawing consists of the following steps:

1. draw the scene
2. fill stencil buffer with 1, where floor is (to avoid drawing shadow away from the floor polygon)
3. calculate + draw shadows (using shaders)

`pipelineFakeProjective();`
Pipeline for a fake projective soft shadow consists of the following steps:

1. project shadow into a texture
2. draw the scene
3. draw the fake soft shadow using the captured texture (with PCF7x7)

A.7 Framework - shadowVolume.h

This header file contains mainly functions for a shadow volume computation. Further description of the most important ones follows:

`cgVolume(float position[]);`
Contains calls for appropriate shaders and variable bindings. Was tested also with `GL_STENCIL_TEST_TWO_SIDE_EXT` extension, which enables only one pass while setting stencil buffer (with different stencil functions for back facing and front facing polygons). It didn't make a big difference in performance and could not be implemented in CgFX technique directly (state settings are defined but they are not working correctly[2]).

`pipelineVolume(bool hard = false);`
Pipeline for a shadow volume consists of the following steps:

1. draw the scene
2. calculate + draw shadows (using shaders)

[2]bug will be hopefully removed in later versions of Cg

A.8 Framework - shadowMap.h

This header file contains functions designated for a shadow maps computation (basic implementation as well as midpoint shadow maps) and resulting shadow computation. Further description of the most important ones follows:

`createJitterLookup(int size, int u, int v);`

As the random number generation is not GPU implemented yet, all noise functions have to be precomputed on th CPU side and stored as a texture. This function creates 3D texture with jittered sampling of a disk. We create *size* different disks of $u * v$ samples (to avoid the same pattern for texture lookups in neighboring fragments).

`generateMidpointShadowMap(float light[], int sample, bool highRes);`

Generation of midpoint shadow maps is slower than the basic approach but they are more universal (self shadowing artifacts removed almost completely) and their usage increases quality of the resulting shadows. Function creates two shadow maps (front faces, back faces) and than averages their depth values into the resulting midpoint shadow map.

`generateShadowMap(float light[], int sample, bool highRes);`

Basic shadow map computation. Scene is displayed from the light point of view, depth buffer is stored into the texture. *highRes* variable toggles usage of the frame buffer object to create textures with resolution higher as is the main window resolution. More information can be found in the chapter 2.3.2.

`cgMap(float position[], int sample, bool highRes);`

This function encapsulates all the necessary calls for shaders, sets variable bindings and calculates shadow.

`pipelineMap(bool hard = false);`

Pipeline for a shadow mapping consists of the following steps:

1. draw the scene
2. calculate + draw shadows (using shaders)

`pipelineFakeMap();`

Pipeline for a fake blurred soft shadow map consists of the following steps:

1. draw the precomputed central shadow map (+PCF)
2. copy PCF shadow into a texture

3. blur the given texture (gaussian, separable = horizontal + vertical blur)
4. store the blurred shadow texture
5. mix the scene with the computed soft shadows texture + fill stencil buffer with 1, where floor is (to avoid drawing shadow away from the scene)

A.9 Shaders - shadows.cgfx

This file is in CgFX format. It consists of variable declarations, function definitions (shaders) and techniques. Each technique defines multiple passes, each comprising of OpenGL state settings and used vertex, fragment shader.

Code A.4: hardProjectiveShadow technique

```
//projective shadow matrix
float4x4 shadowP;

struct fpDataProj {
   float4 position : POSITION;
   float4 texCoord : TEXCOORD0;
   float distCoef : TEXCOORD1;
   float4 color : COLOR;
};

//projective convolution texture
sampler2D projTex = sampler_state {
   minFilter = Linear;
   magFilter = Linear;
   WrapS = Clamp;
   WrapT = Clamp;
};

//vertex shader for projecting scene to the floor plane
float4 projectiveShadowVertex(
         uniform float4x4 modelViewProj,
         uniform float4x4 shadowP,
         uniform float4 shadow,
         float4 P : POSITION,
         out float4 Color : COLOR
         ) : POSITION
{
   //set shadow color
   Color = shadow;

   return mul(modelViewProj,mul(shadowP,P));
}
```

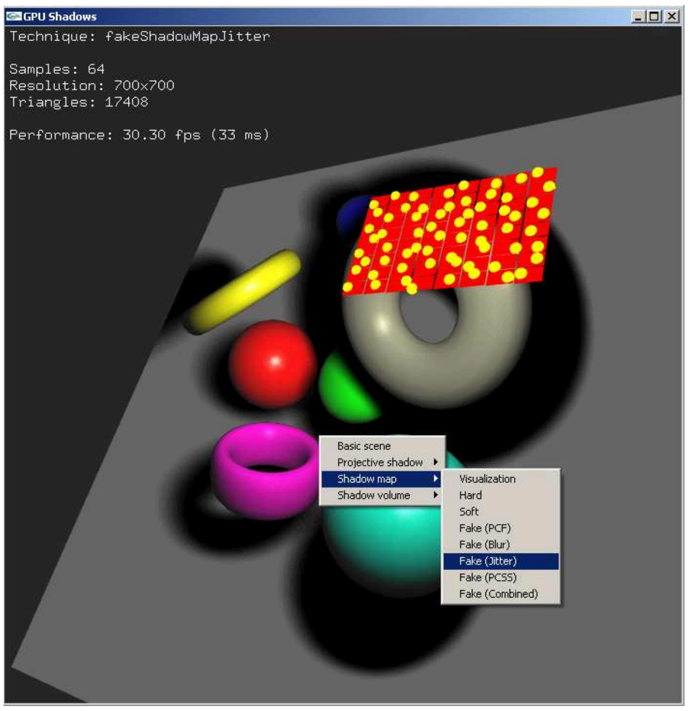
```
//technique
technique hardProjectiveShadow < int type = 5; > {
   pass projectShadow {
      //state settings
      PolygonOffsetFillEnable = true;
      PolygonOffset = float2(-2.0,-10.0);
      DepthMask = false;
      CullFaceEnable = true;
      CullFace = Back;
      StencilTestEnable = true;
      StencilFunc = int3(Equal,1,0xffff);
      StencilOp = int3(Keep,Keep,Zero);
      BlendEnable = true;
      BlendFunc = int2(SrcAlpha,OneMinusSrcAlpha);

      //shaders
      VertexProgram = compile vp40 projectiveShadowVertex(mvp,
                                        shadowP,hardShadow);
      FragmentProgram = NULL;
   }
}
```

Appendix B

Shadow framework

B.1 Overview

The shadow framework is implemented in C++ with the use of OpenGL [RSWJ04] and Cg Toolkit [Fer03]. Its main purpose is to show various methods of shadow computation in an interactive environment and to be able to compare their performance.



```
Technique: fakeShadowMapJitter

Samples: 64
Resolution: 700x700
Triangles: 17408

Performance: 30.30 fps (33 ms)
```

Figure B.1: Shadow framework

Statistics shown on the screen:

- **Technique** – name of the currently used technique (it defines multiple passes with shaders)
- **Samples** – number of light samples used to compute soft, blended versions of shadows (in the brute force algorithms)
- **Resolution** – window resolution
- **Triangles** – number of faces in the scene
- **Performance** – actual performance of the given method in fps and ms

Other information about initialization, parsing of the scene, times taken to recompute dynamic parts of the variables (projection matrices, shadow maps, etc. after light movement) and average performances can be seen in the command line output.

B.2 Controls

Useful controls:

- **LMB:** rotate the scene
- **MMB:** move the light source (height stays constant)
- **RMB:** open context menu
- **Space:** select next technique
- **Return:** start/stop scene rotation
- **Page Up:** move light source up
- **Page Down:** move light source down
- **+:** zoom in
- **–:** zoom out
- **Esc:** exit
- **i,I:** toggle info texts on/off
- **l,L:** toggle light model drawing on/off

- **s,S:** take a snapshot
- **c,C:** camera position info
- **m,M:** toggle midpoint shadow maps usage on/off

B.3 Implemented methods

Type	**Method**	**Note**
Basic scene		*scene without shadows*
Projective shadow	Hard shadow	
	Soft shadow	*blending of the projected shadow from each light sample*
	Fake shadow	*sampling of the texture created from projected shadow*
Shadow map	Visualization	*simple demonstration of depth map*
	Hard shadow	
	Soft shadow	*blending of the shadow maps generated for each light sample*
	Fake shadow (PCF)	*neighborhood sampling and averaging*
	Fake shadow (blur)	*separable Gaussian blurring of hard depth map shadow*
	Fake shadow (jitter)	*PCF supersampling with precomputed random offset lookup texture*
	Fake shadow (PCSS)	*fake soft shadows based on PCF with dynamic penumbra region computation*
	Fake shadow (combined)	*PCF supersampling with random offsets with dynamic penumbra region computation*
Shadow volume	Visualization	*simple color coded displaying of shadow volumes*
	Hard shadow	
	Soft shadow	*blending of the shadows generated from each light sample*

Table B.1: Implemented techniques overview

Note: *all these methods are available from context menu (RMB[1])*

[1] right mouse button

B.4 Requirements

Tested on the configuration: AMD Athlon 3200+, 1024MB RAM, nVIDIA GeForce 7600GT.

Because some newer OpenGL extensions (`EXT_framebuffer_object`) and shader profiles (`fp40`) are used, it is essential to have a graphics card GeForce6 or newer...

List of Code examples

List of Figures

List of Tables