Comenius University

Faculty of Mathematics, Physics and Informatics

Real-time Lighting Effects using Deferred Shading

2012 **Michal Ferko**

Comenius University

Faculty of Mathematics, Physics and Informatics

Real-time Lighting Effects using Deferred Shading

Master Thesis

Study program: 1113 Mathematics

Study field: Computer graphics and geometry

Department: Department of algebra, geometry and didactics of mathematics

Supervisor: RNDr. Michal Valient

Code: b52bb9ed-b405-438d-be5b-71a6dd0e096c

Bratislava, 2012 **Bc. Michal Ferko**

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

PRIHLÁŠKA NA ZÁVEREČNÚ PRÁCU

Meno a priezvisko študenta: Bc. Michal Ferko
Študijný program: počítačová grafika a geometria (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: 9.1.1. matematika
Typ záverečnej práce: diplomová
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: Real-time Lighting Effects using Deferred Shading

Cieľ: Cieľom práce bude vytvoriť systém schopný renderovať 3D scény v reálnom čase za pomoci metódy Deferred Shading. Tento systém má podporovať plne dynamické scény a nemá sa spoliehať na žiadne predvýpočty. Metóda Deferred Shading prináša mnohé výhody čo sa týka generovania osvetlenia, ale prináša aj niektoré nevýhody, ktoré sa budeme snažiť odstrániť. Súčasťou systému bude taktiež možnosť generovania tieňov ako aj útlmu ambientnej zložky svetla.

Vedúci: RNDr. Michal Valient
Katedra: FMFI.KAGDM - Katedra algebry, geometrie a didaktiky matematiky

Dátum schválenia: 28.10.2010

podpis študenta

I hereby declare that I wrote this thesis by myself with the help of the referenced literature.

Bratislava, May 4, 2012 .....

Acknowledgement

I would like to thank my supervisor RNDr. Michal Valient for his guidance and very useful suggestions regarding my work on this thesis. We also thank Marko Darbovic for providing the Sponza and the Sibenik models and the Stanford 3D scanning repository for providing the Dragon model. Special thanks goes to Tomáš Kovačovský, Martin Madaras, Andrej Ferko and Stanislav Fecko for providing some of the testing hardware.

Abstrakt

V tejto práci popisujeme renderovací systém schopný zobrazovať plne dynamické scény pri interaktívnych hodnotách FPS. Náš systém je založený na koncepte Deferred Shading, ktorý oddeľuje renderovanie geometrie a počítanie tieňovania do dvoch prechodov. Náš grafický engine beží na hardvéri podporujúcom OpenGL 3 a vďaka Deferred Shading-u umožňuje naraz stovky svetiel v scéne. Náš systém priamo podporuje HDR osvetlenie a výstupom sú obrázky spracované tone-mapping technikami s pridaným efektom Bloom. Na prekonanie problémov vzniknutých použitím metódy Deferred Shading sú v systéme rôzne riešenia. Anti-aliasing je nahradený celo-obrazovkovým prechodom, ktorý vyhladzuje hrany. Priesvitné objekty sa zobrazujú pomocou Stencil Routed A-Buffer techniky, ktorá nezávisí od poradia renderovania objektov. Na ďalšie zlepšenie zobrazovaných obrázkov poskytuje náš systém tiene za pomoci Shadow Mapping-u a dvoch rozdielnych Ambient Occlusion techník, ktoré bežia v reálnom čase. Systém je jednoducho rozšíriteľný a do budúcnosti plánujeme pridať veľa ďalších techník.

Kľúčové slová: Real-time Rendering, Deferred Shading, OpenGL, High Dynamic Range, Tone-mapping, Bloom, Screen-space Ambient Occlusion, Ambient Occlusion Volumes, Stencil Routed A-Buffer

Abstract

In this thesis, we describe a real-time rendering system capable of displaying fully-dynamic scenes at interactive frame rates. The system is built on the concept of Deferred Shading, which separates the rendering of geometry and evaluation of shading into two rendering passes. Our graphic engine runs on OpenGL 3 capable hardware and allows hundreds of lights to affect the scene thanks to Deferred Shading. We provide support for HDR rendering and the engine itself outputs tone-mapped images with added Bloom. To overcome problems introduced by Deferred Shading, we provide several solutions. Anti-aliasing is replaced by a full-screen post-process and transparent objects are forward-shaded using Stencil Routed A-Buffer - an order-independent technique. To further improve the realism of displayed images, the engine features shadows with the help of Shadow Mapping and two different real-time Ambient Occlusion techniques are used. The system is open to extensions and we plan to incorporate many other techniques in the future.

Keywords: Real-time Rendering, Deferred Shading, OpenGL, High Dynamic Range, Tone-mapping, Bloom, Screen-space Ambient Occlusion, Ambient Occlusion Volumes, Stencil Routed A-Buffer

Contents

List of Abbreviations

OpenGL - Open Graphics Library
GLSL - OpenGL Shading Language
GPU - Graphics processing unit
CPU - Central processing unit
API - Application programming interface
SM - Shadow Mapping
AABB - Axis-aligned bounding box
FOV - Field of view
FPS - Frames per second
SSAO - Screen-Space Ambient Occlusion
AOV - Ambient Occlusion Volumes
SR - Stencil routing

List of Tables

List of Figures

List of Listings

Chapter 1

Introduction

Computer games try to mimic reality and thus immerse the user into the virtual environment. If successful, the player is convinced that he is actually there. That's where real-time rendering comes into play. The visual appearance has to be as realistic as possible, otherwise the player will be distracted by the poor quality of the image and the immersion level will be reduced. Also, the rendering has to be done in real-time, meaning that it produces 30 - 60 frames per second (FPS) depending on how fast the camera/objects in the scene move (60 FPS or even more for displays with frequency higher than 60Hz is crucial for first person shooters such as Counter-Strike).

Ever since the graphics hardware offers better computation power than the CPU, all possible graphics calculations are moved to the GPU. Despite the increased speed, there are still rendering algorithms such as ray-tracing which can hardly be real-time due to the computational complexity.

In this thesis, we analyze techniques used in real-time rendering and propose a system that supports fully dynamic environments without the need to pre-generate lighting or execute other time-consuming pre-processing operations. We start with forward shading, which is the typical straightforward approach to real-time rendering. We then explain modifications and optimizations of this algorithm.

Furthermore, we introduce the concept of deferred shading, which shades

only visible fragments and in a more optimal way then the approaches used with forward shading. *Deferred shading* is not a new concept, it was introduced in [Deering et al., 1988], even though the article never mentions the name deferred shading. In the article, the authors propose a system with two pipeline stages. One rasterizes the geometry and the other shades the pixels, which is the main idea of deferred shading.

Data stored for shading computations is usually called a G-buffer (or geometric buffer), since they store geometric information (color, normal, position/depth, specular exponent, ...). This name was proposed in [Saito and Takahashi, 1990] and is being used ever since.

Deferred shading became a viable real-time rendering technique with the introduction of multiple render targets in the year 2004. Multiple render targets is a hardware-supported feature of DirectX and OpenGL that allows shaders to render into multiple textures at once. This approach allows the generation of the G-buffer in a single rendering pass, which was the necessary speed-up for it to be a real-time technique. After the G-buffer pass, the final scene rendering is performed as a series of post-processing effects that access the information stored in the G-buffer.

The multiple render targets feature is currently supported by many video cards, therefore, deferred shading is a viable rendering option for modern computer games. More and more games are using deferred shading, especially games that benefit from the advantages of this type of rendering.

A good example is the real-time strategy Starcraft II©, whose deferred rendering engine is described in [Filion and McNaughton, 2008]. The game contains two different modes: the in-game mode in which hundreds of units in an exterior are rendered, and the story mode, in which only a few characters and interiors are rendered. The in-game mode benefits from the possibility of having many lights, where the light from special effects can affect the surroundings. They also describe a method for rendering transparent shadows in the in-game mode for effects like fire or smoke. The story mode takes advantage from Screen-Space Ambient Occlusion (SSAO), which simulates

ambient occlusion in real-time and increases realism of the scene. Real-time ambient occlusion techniques will be discussed later.

Another example is the first-person shooter S.T.A.L.K.E.R.. The authors of [Shishkovtsov, 2005] describe the deferred renderer used in this game. Their main aim is optimizing the shadow mapping algorithm by determining what kind of algorithm to use (different for transparent shadows, different for lights casting a projective image, etc.). They also discuss the optimal G-buffer layout, while trying to save as much video memory and performance as possible and yet retain good quality images. An edge-only anti-aliasing algorithm is described as well, since anti-aliasing with deferred shading is tricky and has to be simulated with shaders. The document describes only static ambient occlusion in light maps, whereas the Starcraft II© engine provides a fully-dynamic solution.

There are many more games and game engines that take advantage of deferred shading. The most notable are: CryENGINE (Crysis 2), Killzone 2 and 3, Frostbite Engine and Leadwerks Engine.

The goal of this thesis is to create a system similar to the rendering systems used in the mentioned games while avoiding any kind of pre-computation such as light map generation or BSP tree construction. This will allow our system to display fully-dynamic environments at interactive frame rates while retaining a realistic look.

In Chapter 2, we explain the basics of shading, local illumination models and the Blinn-Phong illumination model. We also explain how OpenGL and The OpenGL Shading Language works, which is necessary to understand the concept of deferred shading. Furthermore, we describe how forward shading works and explain modifications of the algorithm that are more optimal. In Chapter 3, the concept of deferred shading is explained in-depth together with similar techniques. In Chapter 4, we examine certain post-processing effects that are easily integrable into a deferred shading pipeline, such as High Dynamic Range Lighting or Screen-Space Ambient Occlusion. In Chapter 5, we provide an implementation of the deferred rendering system described

in Chapters 3 and 4. The implementation uses C++, OpenGL and The OpenGL Shading Language (GLSL). Our program is a complex rendering system using deferred shading, that produces high quality images at plausible frame rates. In Chapter 6, we show some results and discuss the performance and quality of the implemented techniques.

Chapter 2

Lighting and Shading

For the explanation of the deferred shading concept, it is crucial to have an understanding of the rendering pipeline used by OpenGL. Since version 3 of OpenGL, the fixed function pipeline is deprecated and all rendering must be performed via shaders. Therefore, we explain the rendering pipeline with the integration of shaders, specifically shaders created in the OpenGL Shading Language.

2.1 OpenGL

We will be using OpenGL for implementing all the algorithms described in this thesis. OpenGL stands for the Open Graphics Library, an open standard for GPU-accelerated rendering of primitives (points, lines, triangles, ...). OpenGL was developed by Sillicon Graphics International (SGI) in 1992. In the past years, OpenGL has evolved quickly and the version at the time of writing is 4.2. OpenGL contains numerous extensions (some are GPU-vendor specific, some are OS specific) which allow modifications of the rendering pipeline.

OpenGL provides a hardware-independent application programming interface (API) that simplifies 2D and 3D rendering. For OpenGL to run (render to a window), an OpenGL context has to be created for the window

we are going to render to. The rendering context is platform-specific, since OpenGL does not provide a way to create a window in the operating system's window manager.

Using the fixed-function pipeline of OpenGL, it is possible to render complex geometry in real-time on almost any graphics hardware. With a smart view frustum culling algorithm and other optimizations (Vertex Arrays, Vertex Buffer Objects), it is possible to render tens of thousands of triangles every frame. The newest OpenGL specification, as well as specifications for older versions and the OpenGL Shading Language can be found on the official OpenGL web page [OpenGL, 2011].

2.2 OpenGL Shading Language

Some of the OpenGL extensions introduced a way to write user-programs that will be executed on the GPU when rendering primitives. These programs are called *shaders*. First shaders were just assembler instructions for the GPU, but as time passed, a few high level shading languages emerged. For DirectX, it is the High Level Shading Language (HLSL), for OpenGL it is the Cg (C for graphics, developed by NVidia) and OpenGL Shading Language (GLSL). All three are similar in syntax, but there are certain differences. We will be using GLSL, which is part of the OpenGL core since version 2.0. OpenGL version 3 deprecated the fixed-function pipeline and all rendering should be performed via shaders.

We will only introduce the basic capabilities of shaders. For more complex algorithms and understanding, there are a lot of great books that are dedicated to shaders, such as the "Orange Book" [Rost, 2005].

There are 3 types of shaders: vertex, geometry and fragment. All of these shaders are easily understandable with the help of Figure 2.1. The figure is actually a data-flow diagram, showing how the image is generated in the frame-buffer from drawing calls like `glDrawElements`. In OpenGL 4, the Tesselation Control and Tesselation Evaluation stages were introduced,

but these will not be discussed here.

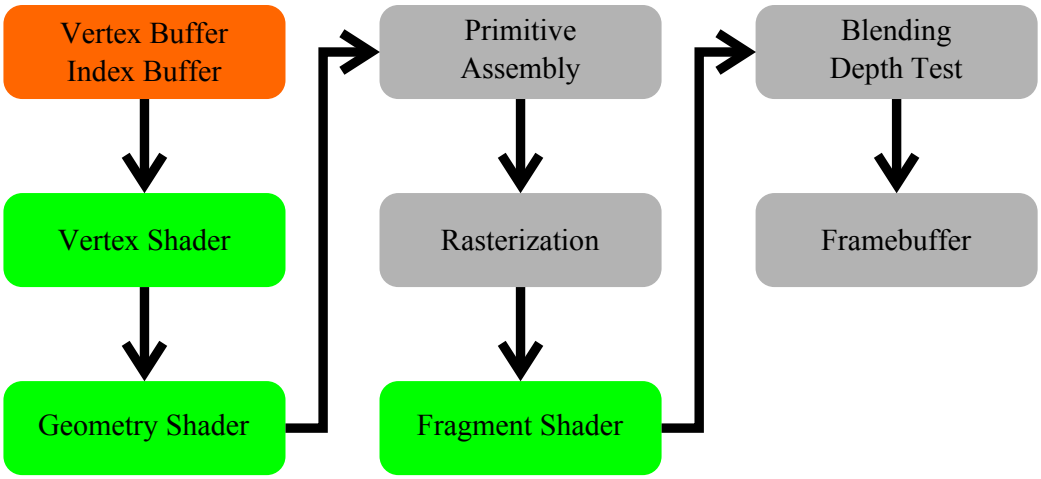

Figure 2.1: OpenGL Shader Rendering Pipeline

At the beginning, data from vertex buffers (vertex positions, texture coordinates, ...) are sent to the vertex shader (by calling one of the primitive drawing functions). Input to the vertex shader are therefore, vertex attributes such as vertex colors, positions, etc. But a vertex shader is tied to a single vertex, meaning that for each vertex, it is executed once. Furthermore, it has no information on the primitive it belongs to or other vertices in the primitive. A vertex shader transforms the vertex positions from object space (these are the exact same values as the application specifies when calling a function like `glDrawElements`) into clip space, using the modelview matrix (sometimes split into the model matrix and the view matrix) and the projection matrix. Texture coordinates are transformed as well, and there are many more calculations that can occur in the vertex shader (such as character skinning, lighting calculations, ...).

The output from vertex shaders is sent to geometry shaders. Output type of the geometry shader is same as the input type. A geometry shader can emit new primitives (e. g. duplicating triangles). If a geometry shader is not present, the data from vertex shaders is sent directly into the fragment shader. Primitives are handled as they are interconnected, since a geometry

shader is always set to operate on a certain kind of primitives (points, lines, triangles). Therefore, a geometry shader is executed once per primitive. The geometry shader can also change the type of primitive being rendered and many more.

After geometry shaders, the primitive assembly combines the transformed vertices into primitives in the clip space. These are then *rasterized*, meaning that they are converted into pixels. For each pixel rasterized, a fragment shader is executed (this is the reason why fragment shaders are sometimes called pixel shaders). The output of a fragment shader is the pixel color that should be displayed and also the depth value for depth testing. It is also possible to render to an off-screen buffer (OpenGL Framebuffer Object - FBO) which can have multiple color buffers attached and then the output of the fragment shader are colors that should be written to each of these buffers.

Finally, a depth test occurs on each fragment, discarding fragments that do not pass the test. Once a fragment is ready to be written into the framebuffer, it is blended into the image based on blending settings and the new color is written.

There are special types of variables in GLSL. For each type of shader, *uniform variables* are global, read-only variables whose value is set outside the shader (by the OpenGL application). Uniform variables are specified using the `uniform` keyword. There are certain limitations concerning the maximum number of bytes taken by uniform variables. These variables can be used to control shader execution without the need to change the shader. Typically, these can be settings like enabling texturing, lighting or setting light and material parameters. Every call to `glDrawElements` or a similar function, the values set to uniform variables remain unchanged throughout the whole rendering process and all shaders. This also means that one uniform variable can be shared between all shader types.

Another type of variables are *input variables* (specified by the `in` keyword). These variables are read-only and differ in meaning in each shader.

In geometry shaders, these are arrays of vertex attributes (length is specified by the type of primitive used - 1 for points, 2 for lines, 3 for triangles). In vertex shaders, input variables are vertex attributes (but not arrays of them, since the vertex shader "sees" only one vertex at a time). For fragment shaders, input variables are interpolated values of output variables from vertex shaders. These can be texture coordinates, colors, etc..

Finally, there are *output variables* specified by the `out` keyword. A shader usually has some output, for instance a fragment shader's output is the color of the fragment. For vertex shaders, these are vertex attributes that are sent to the respective geometry shader. Output of vertex shaders can be different, but as mentioned before, these are usually colors, coordinates or normals. These values are then interpolated between vertices of the same primitive (using bilinear interpolation for triangles) and hence the values for input variables of the fragment shader are calculated. The output of fragment shaders is color and/or depth, but multiple output variables can be defined when using FBO with multiple color attachments (aka. multiple render targets).

2.2.1 Example shaders

We will now provide one example for each type of shader and describe it's functionality.

In Listing 2.1, you can see a vertex shader. It reads the model transformation matrix values specified by the program from the uniform variable `model`. There are two input vertex attributes, position and color. Color is written to the output variable `vColor` and the position of the vertex is transformed into clip space. Notice that `gl_Position` is a built-in output variable from the vertex shader that must be written to, otherwise the rasterizer would have no information on where to render the primitive. We transform vertex positions and normals from object space into world space with this vertex shader. Transformation to clip space is left for the geometry shader to do.

In Listing 2.2 is a simple geometry shader, that recieves 3 input param-

```
#version 330            // Use version 3.3 of GLSL
uniform mat4 model;     // Application set model matrix
in vec3 aVertex;        // Input vertex attribute - position
in vec2 aCoord;         //   - texture coordinate
in vec3 aNormal;        //   - surface normal
out vec2 vCoord;        // Output variable texture coordinate
out vec3 vNormal;       // Output surface normal
void main()
{
   vCoord = aCoord;     // Direct copy of values
   vNormal = model * vec4(aNormal, 0.0);     // Object -> World
   gl_Position = model * vec4(aVertex, 1.0); // Object -> World
}
```

Listing 2.1: A simple vertex shader

eters. The first is `gl_in`, which is a built-in array. This array is used as the geometry shader input containing vertex positions that were stored into `gl_Position` in the respective vertex shaders. The second and third are custom vertex attribute arrays corresponding to output from the vertex shader.

The shader transforms the data into clip space and sends the data it received into the fragment shader. Furthermore, it creates duplicates of triangles and extrudes them in the direction of the surface normal while retaining the texture coordinate values.

Finally, we have a fragment shader in Listing 2.3. This fragment shader receives interpolated texture coordinates from the geometry shader through the variable `coord`, as well as the specified texture unit from which the texture should be taken (through the uniform variable `diffuseMap`). All it does is read the texture color based on texture coordinates and store it into the output variable (which is attached either to the frame-buffer or a color attachment of a FBO).

2.3 Shading

Shading is a set of algorithms that simulate lighting in scenes. These algorithms take only the local (other objects in the scene do not affect the

```
#version 330                    // Use version 3.3 of GLSL
layout(triangles) in;           // Input primitives are triangles
layout(triangles, max_vertices = 6) out;  // Output triangles,
   together max. 6 vertices
uniform mat4 viewprojection;    // View-Projection matrix
in vec3 vCoord[];               // Input from vertex shader
in vec3 vNormal[];              // Input from vertex shader
out vec2 coord;                 // Output to fragment shader
void main()
{
   // Iterate all vertices of this primitive
   for(int i = 0; i < gl_in.length(); i++)
   {
      coord = vCoord[i];                    // Set output coord
      vec4 pos = gl_in[i].gl_Position;
      gl_Position = viewprojection * pos; // World -> Clip space
      EmitVertex();                         // Add vertex
   }
   EndPrimitive();                          // Finish 1. triangle
   // Iterate all vertices once again
   for(int i = 0; i < gl_in.length(); i++)
   {
      coord = vCoord[i];                    // Set output coord
      vec4 normal = vec4(vNormal[i], 0.0);// Get normal
      vec4 pos = gl_in[i].gl_Position + normal; // Extrude
      gl_Position = viewprojection * pos; // World -> Clip space
      EmitVertex();                         // Add vertex
   }

   EndPrimitive();                          // Finish 2. triangle
}
```

Listing 2.2: A simple geometry shader

resulting lighting) properties of the object's surface and compute the amount and color of lighting reflected into the camera. According to [Akenine-Möller et al., 2008], *shading* is the process of evaluating a shading equation to compute the outgoing light from a surface in the direction of the viewing ray (or into the camera). This approach is often referred to as *local illumination*, the opposite of global illumination. Local illumination computes only the direct illumination of the surface, whereas global illumination computes indirect illumination as well (light rays reflected from surfaces onto other surfaces, light rays refracted into a transparent material, ...). Global illumination sim-

```
#version 330

uniform sampler2D diffuseMap;    // Represents a 2D diffuse map

in vec2 coord;                   // Input texture coordinate
out vec4 outColor;               // Output fragment color

void main()
{
   outColor = texture(diffuseMap, coord); // Read value from
       texture at the texture coordinates and set as output color
}
```

Listing 2.3: A simple fragment shader

ulates the real physical behavior of light and therefore, the produced images are more photo-realistic than those produced by local illumination.

The problem with global illumination is performance. Despite the huge advancements in CPU and GPU speed, algorithms that compute global illumination like ray tracing or radiosity cannot be real-time on consumer hardware. A good example are famous computer-generated movies, that were rendered *weeks* on supercomputers. Therefore, software that requires real-time rendering has to use local illumination instead. There are various number of extensions to local illumination that increase the realism of the produced images. These extensions usually do not correspond to physical behavior of light, but are simplifications of it that simulate lighting effects accurately enough.

Local illumination has much greater performance. A local lighting model usually does not have lots of parameters and computing the color of one pixel can be done with a few vector operations.

2.3.1 Phong Shading

Phong Shading is a simple reflection model. It was introduced by Bui Tuong Phong in [Phong, 1975]. It is an illumination model that takes into account material parameters, surface orientation, light parameters, light position and

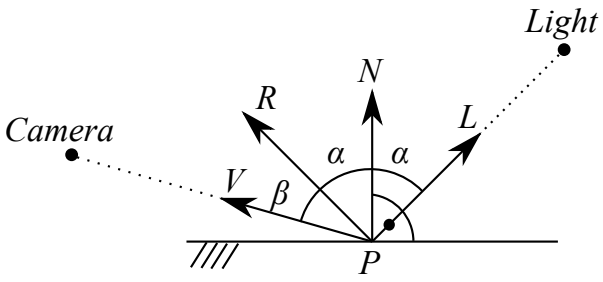

Figure 2.2: Phong shading model for one light source. All vectors are normalized for the dot products to directly measure angles.

camera position. The Phong reflection model is widely used in real-time applications, together with the Blinn-Phong model that is derived from this one.

The following equation is applied to every pixel to compute it's color based on lights affecting this pixel.

$$I_p = k_a i_a + \sum_{\mathrm{m} \in \{0, \cdots, \mathrm{n}-1\}} (k_d i_{m,d}(L_m \cdot N) + k_s i_{m,s}(R_m \cdot V)^{k_e}). \tag{2.1}$$

In this equation, I_p is the output color (or intensity) calculated from light contribution. k_a is the material's ambient color. i_a is the global ambient term. These terms combined together determine the ambient lighting of the pixel. Ambient light is defined as the light incoming into the point from all directions, therefore, this term is constant throughout the whole object (in a physically correct model, no ambient term is present).

k_d is the material's diffuse color and $i_{m,d}$ is the diffuse color of light m. Diffuse light is defined as the light incoming directly from the light source and reflected into the camera. However, due to the local nature of the model, even if there are objects occluding the object being rendered, the color remains the same (shadowing techniques are needed to take care of this unnatural behavior). The amount of reflected light depends on surface orientation, which is determined by the normal N. The direction of incoming light from light source m is L_m and the dot product $cos\ \alpha\ =\ N \cdot L_m$ directly corresponds

to the amount of light reflected from the surface (of course, N and L_m need to have unit length). As seen in Figure 2.2, if $\alpha = 0$ (N and L are parallel), then $N \cdot L = 1$ and the diffuse term is maximal. If $\alpha = \frac{\pi}{2}$ (N and L are perpendicular), the diffuse term is all zeros. With a bigger α, less rays actually hit the surface (since it has a smaller area from the light's point of view) and therefore the amount of reflected light is smaller.

k_s is the material's specular color and $i_{m,s}$ is the light's specular color. The result of the specular term are specular highlights, which are shown only when the viewer is at a specific position. Figure 2.2 shows that the term $R \cdot V = cos\ \beta$ (again, both vector need to be normalized). Therefore, if the reflected vector R points directly into the camera, the specular term is maximal. k_e is the material's specular exponent, that further controls the specular term. The higher the exponent is, the smaller the highlights and thus the object appear more shiny (therefore, it is sometimes called "shininess", for instance in the OpenGL lighting model). For a perfectly diffuse material, the specular color is $(0, 0, 0, 0)$ which removes the specular term completely.

There can be different types of light in the shading equation. The usual lights are point lights, spot lights and directional lights. A point light is similar to a light bulb. It emanates light from a point in the scene uniformly into all directions. A spot light is similar to a flash-light, where the part of the scene lit by it is reduced to a cone. A directional light corresponds to sunlight, where the light rays come from an almost infinitely far away point light source. All the rays share the same direction, hence the name.

For point and spot lights, we also specify an attenuation function as follows:

$$att(d) = \frac{1}{a_c + da_l + d^2 a_q}, \tag{2.2}$$

where d is the distance between the shaded point and the light's position. Constants a_c, a_l and a_q are the constant, linear and quadratic attenuation factors for a light. The diffuse and specular terms are multiplied by this term to account for attenuation.

2.3.2 Blinn-Phong Shading

Blinn-Phong shading [Blinn, 1977] is a slight modification of the Phong shading that is used in the fixed function pipeline of OpenGL and DirectX . The only difference between Phong and Blinn-Phong shading is the evaluation of the specular color. First of all, the *halfway vector* H is a unit vector that is halfway between the viewer vector V and the light vector L_m. Therefore:

$$H_m = \frac{L_m + V}{|L_m + V|} \tag{2.3}$$

The term $(R_m \cdot V)^{k_e}$ from Equation 2.1 is replaced with $(N \cdot H_m)^{k_e}$.

Blinn-Phong shading produces results that are closer to the physical behavior of many materials, and is therefore a better approximation than Phong shading. Due to this fact, we use the Blinn-Phong shading in our engine.

2.4 Forward Shading

With the introduction of vertex, fragment and geometry shaders, a wide variety of effects became possible to render. One of the mostly used is per-pixel shading of rendered objects. A fragment shader is input the position and color of light (through uniform variables) and using a simple lighting equation (e. g. Blinn-Phong shading), the color of the fragment is computed. If there were hundreds of lights, the fragment shader would have to go through every light and compute the light contribution. That would result in a shader that performs thousands of operations and it would take too long to execute, making the rendering so slow that the 30 - 60 FPS limit could not be reached.

Deferred shading partially solves this problem, but we will describe forward shading first as a method that precedes deferred shading. Forward shading suffers from the number of lights limitation described.

Forward shading is the process of rendering all primitives without intermediate steps. This means that the output image is generated on the fly and fragments are shaded right ahead. There are a few possibilities of how forward shading can be done, we will discuss the most used in this section.

Forward shading is what the pipeline in Figure 2.1 was designed for. Without programmable graphics hardware, there was no choice but to render using forward shading.

2.4.1 Single-pass Lighting

In the single-pass lighting method, which was the first forward shading technique used, the final image is synthesized in one pass. The image generated is the final image shown in the framebuffer. Without shaders, the fixed-function pipeline would shade the fragments, but only using per-vertex lighting calculations.

With the introduction of shaders, the application programmer is given a possibility to perform lighting calculations per-pixel. The single-pass method will then contain a vertex shader that does nothing special, except prepare the variables for lighting computation for the fragment shader. The fragment shader that iterates through a light list (stored in uniform variables) and accumulates lighting affecting the fragment. A pseudo-code is shown in Listing 2.4. This approach has $O(m \cdot n)$ complexity, where m is the number of objects and n is the number of lights.

The function `shade` uses a shading equation to calculate the color of the fragment, while accessing material properties from uniform variables and textures. Objects usually have a diffuse texture assigned to them that represents the diffuse term (and mostly the ambient term as well) in each point. Additionally, an object can have a specular color map, a specular roughness map, a normal map etc. that are used to determine material properties at every surface point.

```
for (each object)                                  // Application
-------------------------------------------------------------
   for (each light)                                // Shader
      shade(point, light)                          // Shader
```

Listing 2.4: Single-pass Forward Shading

There are several things that cause this approach to do unnecessary computation. First of all, we render all objects and shade every single fragment as it comes. But if newer fragments overwrite the old ones, we wasted time with calculating the color of the old fragments. For a rendered image, the *depth complexity* is the average number of overlapping objects per pixel. This number is also the number of fragment shader executions (pixel color calculations) per pixel. The lowest possible depth complexity is 1, where for each pixel, the fragment shader is executed exactly once. This is one of the advantages of deferred shading and will be discussed later. For scenes, where depth complexity reaches much higher values than 1, we waste a lot of GPU cycles on shading occluded fragments.

A simple fix of the depth complexity issue is to render all geometry twice and take advantage of the *early depth test*. It is a special optimization that can occur if a fragment shader does not modify the depth value (write to the built-in `gl_FragDepth` variable) that was automatically computed. The GPU can then perform the depth test before executing the fragment shader, and if the test fails, the fragment shader is not executed at all.

In the first pass, only the depth values are stored and no shading occurs (writing to the color buffer will be disabled). If the conditions are satisfied, the depth test stage is moved before the execution of the fragment shader (see Figure 2.1). The second pass is rendered as we would normally render, with the exception that the depth function is set to equal, allowing the hardware to discard all fragments except the one that will be shown in the final image. Since we have generated depth values in the first pass, the second pass discards all invisible fragments and therefore reduces depth complexity to 1. But we render the scene twice.

Another wasted computation occurs during the light calculation. We iterate over all lights for each pixel, even those lights that do not contribute to the pixel's color (the lights are too far away from the object). Another problem that occurs is the memory and instruction limit for shaders. All the light data has to be stored in uniform variables, and if there were hundreds

of lights, the memory limit would be exceeded and the shader could not be used. The same thing would happen with the instruction limit, hundreds of lights means hundreds of dot products, vector multiplications etc. Multi-pass lighting takes care of these problems.

2.4.2 Multi-pass Lighting

As the name suggests, this approach uses multiple passes to render the final output. For each light, one pass occurs, during which the scene is rendered as if there was only the one light. This takes care of the memory and instruction limits in shaders, since the shader only calculates the light contribution for one light. The output is then additively blended together for the final result. A very simple pseudo-code is shown in Listing 2.5. The rendering complexity remains $O(m \cdot n)$ as it was with single-pass lighting. A new performance issue pops out: each object is rendered n times, n being the total number of lights. This increases batch count in comparison to single-pass lighting. *Batching* is a problem that occurs when rendering thousands of objects. For each object, one `glDrawElements` call is performed (one such call is referred to as a batch). The call has to be sent to the GPU, which takes an insignificant amount of time, but when there are thousands of batches, the performance drops. There is a section on optimizing these in [Watt and Policarpo, 2005], such as grouping objects with the same material together or merging chunks of geometry into larger chunks.

```
for (each light)                                    // Application
   for (each object)                                // Application
-------------------------------------------------------------
      shade(point, light)                           // Shader
```

Listing 2.5: Multi-pass Forward Shading

A few simple optimizations will increase the performance rapidly. Each object has to be rendered only once for each light that affects it. Also, lights whose light volume (the volume in which the light affects objects) is outside

the viewing frustum should not be taken into account. The same counts for objects. This results in a optimized multi-pass approach, as shown in Listing 2.6.

```
for (each visible light L)                    // Application
   for (each visible object affected L)       // Application
------------------------------------------------------------
      shade(point, L)                         // Shader
```

Listing 2.6: Optimized Multi-pass Lighting

Chapter 3

Deferred Shading

Single pass lighting cannot handle lots of lights effectively and the multi-pass approach might need to render one object many times. Rendering one object multiple times increases batch count and can limit performance. Deferred shading is a rendering method that removes certain limitations of the single-pass and multi-pass approaches, but introduces new problems. We will describe the main idea of deferred shading and analyze how to overcome the problems that emerge.

```
// Geometry pass
for (each object)                                  // Application
-------------------------------------------------------------
   store material properties into G-Buffer         // Shader

// Lighting pass
for (each light)                                   // Application
   render(light shape)                             // Application
-------------------------------------------------------------
      shade(point, light, G-Buffer)                // Shader
```

Listing 3.1: Deferred Shading Pseudo-code

Deferred shading was introduced in [Deering et al., 1988]. The main idea is to split rendering into two rendering passes. The first one, called the geometry pass, renders the whole scene while storing material properties into several textures. These textures are called the geometry buffer (G-buffer).

The second pass (lighting pass) performs a series of post-processing effects on the G-buffer and displays the shaded result. These post-processing effects actually perform the lighting and shading. The main point is to apply lighting only where it is needed. This is done by rendering the light volume and every pixel that is rendered accesses the necessary data from the G-buffer, computes the final pixel color and additively blends into the final image. A simplified deferred shading pipeline is shown in Listing 3.1.

3.1 The G-Buffer

The G-buffer is an important intermediate step. Since the graphics hardware supports multiple render targets, the G-buffer can be generated in one pass. The layout of the G-buffer is an important part of the renderer optimization. For applying lighting, we require certain information that has to be stored in a G-buffer. Albedo (or diffuse color) of the pixel has to be stored. If we also want to have per-pixel specular color/exponent, the specular color/exponent needs to be stored as well. For these colors, 3 bytes per pixel (24bpp) is enough. We might need more storage when handling High Dynamic Range textures, but HDR will be discussed later. Our G-Buffer images are shown in Figure 3.1.

An important part of the Blinn-Phong shading equation are surface normals. Without normals, we cannot compute lighting properly. For correct operation of the lighting equation, the pixel position has to be stored as well. The ideal would be to store normals and positions into two 3-channel 32-bit float texture. This would result in a very memory-consuming G-buffer. For instance a 1024x768 G-buffer with 24bpp color, and float precision of normals and positions would take 3 + 12 + 12 = 27 bytes per pixel, which makes the whole G-buffer over 20MB large. In Full HD resolution it is over 50MB, which takes a lot of space and does not leave enough memory left for other textures on older graphic cards. For such a large G-buffer, the video memory is not the only part that suffers. The fill-rate will also be encumbered by

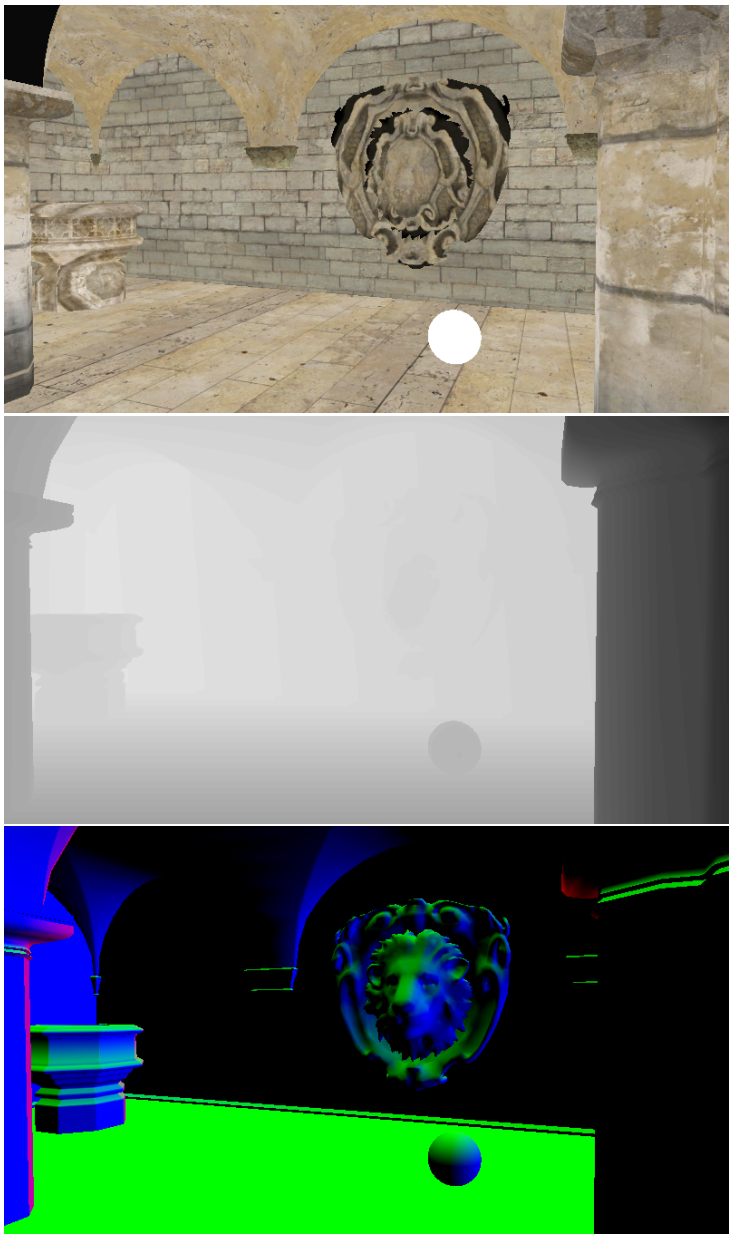

Figure 3.1: G-Buffer data - Albedo, Depth and Normal

performing too many operations per-pixel.

Using half-precision (16-bit float) for normals is unacceptable, it produces unwanted artifacts due to the low precision. To improve the precision, we

can use 16-bit fixed point numbers instead. OpenGL directly provides such a type for textures. OpenGL image formats that end with `_SNORM` represent signed normalized integer values. These values are fixed-point in range $[-1, 1]$, but are represented by a signed integer. The signed integer is in range $[-INT_{max}, INT_{max}]$ (just like classic integer values) and when accessed by a shader, it is divided by INT_{max} and results in a floating-point value.

With the help of these fixed-point numbers, we can represent the normal (whose coordinates are within range $[-1, 1]$ thanks to their unit length) with three fixed-point values. These values are uniformly distributed in the $[-1, 1]$ interval and all bits are fully utilized, therefore the precision of stored normals is much higher than we would achieve with floating-point numbers.

Furthermore, we can only use two channels instead of three by storing only the x and y coordinates of the normal. The authors of [Kircher and Lawrance, 2009] describe a transformation that allows to reconstruct the z coordinate from the other two coordinates. We store the x and y coordinates of the following normal:

$$\bar{n}' = \frac{\bar{n} + (0, 0, 1)}{|\bar{n} + (0, 0, 1)|}. \tag{3.1}$$

We store $\bar{n}'_x$ and $\bar{n}'_y$ for further processing. When we need the original normal again, the inverse mapping is:

$$z = \sqrt{1 - |(\bar{n}'_x, \bar{n}'_y)|^2} \tag{3.2}$$

$$\bar{n} = (2z \cdot \bar{n}'_x, 2z \cdot \bar{n}'_y, 2z^2 - 1) \tag{3.3}$$

This transformation cannot correctly transform the vector $(0, 0, -1)$, but that is not a problem since this is a vector facing directly away from the viewer (in view space, which we use when storing the normal). Such a vector is therefore always on back-face and will never be rendered.

With this approach, we saved 8 bytes per pixel in comparison to the three 32-bit float normals, at the cost of a few more instructions in the shader. The precision is fully retained.

Thanks to the depth buffer, the other large texture storing pixel positions becomes unnecessary. During the G-buffer generation, we have to use a

depth buffer, and thankfully, the depth value can be used to reconstruct pixel position. We take pixel coordinates $(x, y) \in [0, 1]^2$ and depth $z \in [0, 1]$ and transform these back into clip space by mapping the target range into $[-1, 1]^3$. We then apply the inverse of the projection matrix used when generating the G-Buffer. The result is the point's view-space position, which is then used as the position.

3.2 The L-Buffer

If we intend to perform only lighting calculation using additive blending, we can render directly into the window's framebuffer. However, there are still many post-processing effects that can be applied after the lighting, and therefore it is useful to store the illuminated scene into a texture. This texture is called the *L-Buffer* or the lighting buffer. This intermediate step becomes necessary when using High Dynamic Range Lighting, which will be further discussed in Section 4.1.

Not taking into account how we will use the output, the L-Buffer generation is crucial. We have to decide which pixels are lit by a certain light and shade them accordingly.

3.2.1 Rendering light volumes

Determining which pixels are affected by which light can be extracted from rendering the light volumes using the same view-projection matrix as was used during G-Buffer generation. During this rendering, the fragment shader determines if the pixel being rendered actually is inside the light volume and should be lit by the light. This calculation depends on the shape of the light volume.

For point lights (represented by a sphere), the fragment shader has to reconstruct the pixel's 3D position and check if this position lies inside the sphere. This is simply a check if the distance between two points is lower than the radius.

If we are bounding a point light with a sphere, we expect that this light's contribution outside this sphere should be 0. Otherwise, we get a feeling of cutoff lighting which is desired when using spot lights, but not when using point lights. However, if the result of the attenuation function in Equation 2.2 should be zero means that the distance must be $+\infty$. Since light has quadratic attenuation in the form

$$att(d) = \frac{1}{1 + d^2}, \tag{3.4}$$

and a more straightforward and artist-friendly way to define light attenuation is to define the light's radius and let the attenuation factor be derived from the radius.

This brings the need to design an attenuation function that resembles the quadratic attenuation, but has $att(r) = 0$, where r is the light's radius.

Let us derive such a function by starting from Equation 3.4. We want the function to have values very similar to the standard quadratic attenuation, but instead of retaining a small non-zero value at distance $d = r$, the function should reach zero.

We use the following attenuation function:

$$att(d) = \frac{1 - \left(\frac{d}{r}\right)^{20}}{1 + d^2}. \tag{3.5}$$

We subtract an expression that is close to zero for $d << r$ but reaches 1 for $d = r$. This ensures an attenuation factor closely resembling the original, but smoothly attenuating to zero at the sphere's boundary. The difference between the old attenuation function and our new attenuation function is shown in Figure 3.2.

For spot lights (represented by a cone), we have the spot direction and spot angle (or cutoff angle). The test that occurs for pixels belonging to a cone is very simple. We take the vector between the shaded point and the cone position (the tip of the cone) and the spot direction vector. If the angle of these two vectors is less than or equal to the cutoff angle, the point is inside the spot light's volume. This can be seen in Figure 3.3. During runtime, we test if $\beta \leq \alpha$, where $\cos\beta = P_{dir} \cdot S_{dir}$ if $|P_{dir}| = |S_{dir}| = 1$

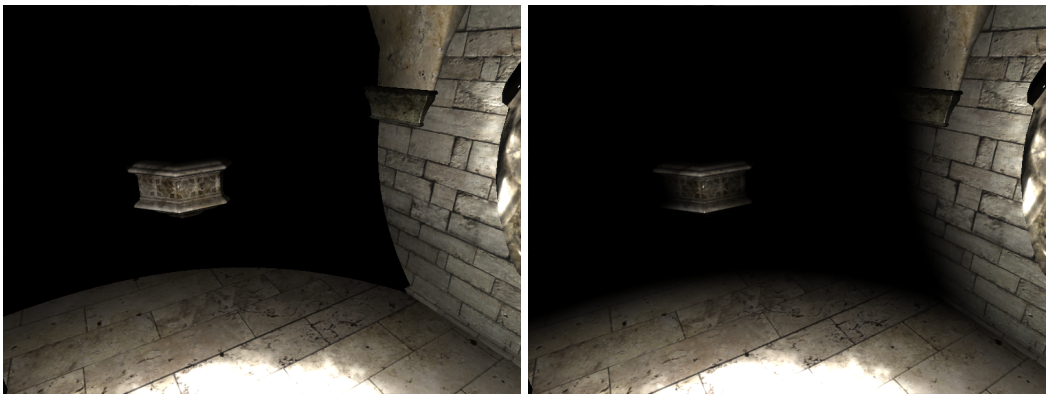

Figure 3.2: Our modification of the attenuation function. Showing cut-off lighting on the sphere boundary (left) and a simple fix causing smooth transitions with our new attenuation function (right).

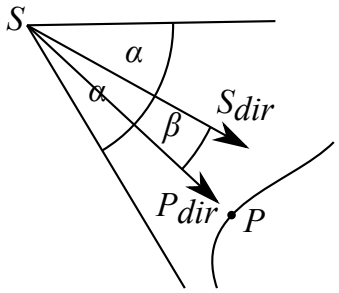

Figure 3.3: Testing if point P is inside the cone (or spot light volume) defined by cone tip S, cutoff angle α and direction S_{dir}.

For a directional light (just like sunlight), every single pixel has to be lit. This is thus performed by rendering a fullscreen quad and shading every pixel accordingly. However, this additional step increases fillrate (since all the pixels have to be rendered twice). Therefore, the authors of [Filion and McNaughton, 2008] proposed to render directional lights with a traditional forward rendering pass (possibly generating the results during the G-Buffer pass).

Due to this limitation, deferred shading does not handle many directional lights well. On the contrary, many point lights and many spot lights are

handled much more efficiently than in forward shading.

Rendering light shapes as simple geometric objects is straightforward and we do not need very precise cones and spheres, a few hundred vertices is enough. We want to tightly encapsulate the light volumes to avoid overdraw in this step. However, too densely tessellated meshes waste performance by executing the vertex shader too many times.

In spite of the many advantages of deferred shading, there are certain limitations. Due to these limitations, deferred shading is not always the way to go. One limitation has already been mentioned: multiple directional lights are not handled well, resulting in lower performance than forward shading.

3.3 Anti-Aliasing

A serious issue arises from using deferred shading. Most graphic cards support hardware anti-aliasing only when rendering to the window's framebuffer. This is not the case of deferred shading, where the scene is rendered into a texture. Therefore, anti-aliasing has to be simulated.

There are a few options, the first one of them is super-sampled anti-aliasing (SSAA). We render the G-Buffer and L-Buffer at a higher resolution than the actual window and downscale it when finally rendering into the window. This approach is unacceptable, because it requires a 4 times larger G-Buffer and L-Buffer to perform 2x2 anti-aliasing (for a 800x600 image, a 1600x1200 image is rendered and each of the final pixels contains the averaged value from 4 pixels).

Multisample anti-aliasing (MSAA) is the traditional hardware supported anti-aliasing which executes the fragment shader only once per pixel but the depth and stencil values are super-sampled. Due to this fact, it clearly outperforms SSAA, but still has serious impact on performance.

More viable approaches perform post-processing of the output image. We take the final tone-mapped image as a texture, execute a full-screen rendering pass that accesses the texture and the fragment shader analyzes the texture

and outputs smoothed edges. Such an edge-blurring filter can take into account G-Buffer data as well, such as normals or depths.

Once the edges are determined, the color of the output pixel is computed as a weighted average of neighboring pixels. Thanks to the edge-detection step, only edge pixels are blurred, saving as much performance as possible while improving edge quality.

These approaches can blur sharp edges in textures as well, not only on triangle edges.

The fast approximate anti-aliasing (FXAA) [Lottes, 2009] is a widely used edge-detecting and blurring anti-aliasing method. It only uses the color buffer and performs edge-detection in several steps that quickly discards non-edge pixels. The author also provides an easily-integrable shader that performs several different versions of FXAA on the input texture.

Morphological anti-aliasing (MLAA) [Reshetov, 2009] analyzes aliasing patterns in the image between neighboring pixels and retrieves positions of L-shaped, U-shaped and S-shaped aliasing patterns. These are afterwards blurred based on the actual shape.

3.4 Transparent objects

Probably the worst problem to overcome in deferred shading is the rendering of transparent objects. The G-Buffer only holds information about the nearest pixel and therefore does not contain any information about pixels behind the closest objects. For these objects to render correctly (using alpha blending), all the triangles need to be sorted in back-to-front order, making the more distant objects render sooner. This is due to the nature of the alpha blending operation:

$$color_{final} = (1 - \alpha_{src})color_{src} + \alpha_{src}color_{dst}, \tag{3.6}$$

which is order-dependent. The $color_{dst}$ is the current color stored in the framebuffer, $color_{src}$ is the color of the currently rendered primitive that is

to be blended into the framebuffer. And finally, α_{src} is the corresponding alpha value for $color_{src}$. Replacing alpha blending with a different blending operation, where the order of operands does not affect the result, we could avoid the need to sort. However, such blending operations do not correspond to real behavior of light coming through transparent objects and thus produce non-realistic results.

Most deferred shading pipelines revert to forward shading and perform the traditional back-to-front polygon sorting and back-to-front rendering. This approach can be CPU-intensive when the transparent objects are dynamic and the triangles need to be sorted every frame.

To sort triangles back-to-front, we first collect all triangles from the scene that are transparent. We put these triangles into a BSP tree. Every frame, the BSP tree is traversed based on the camera position and triangles are rendered back-to-front.

The generation of a BSP tree is CPU-intensive, especially when there are thousands of triangles to process. If we have static transparent objects, the BSP tree can be generated just once and then traversed each frame. If we had dynamic transparent objects, the BSP tree would require rebuilding every time the objects move. Most of the CPU time would be spent on BSP tree generation and it could result in a CPU bottleneck. This is the main reason why this approach is not being used as much.

3.5 Order Independent Transparency

The problem of rendering transparent objects without having to sort triangles is referred to as *Order Independent Transparency*.

To avoid the CPU-intensive BSP tree approach, a lot of order-independent techniques were developed. There are per-pixel linked lists with the help of OpenGL 4 or DirectX 11, where we actually create one linked-list per pixel containing depth values and colors [Thibieroz and Gruen, 2010]. In the final pass, all samples in a pixel are sorted based on stored depth and blended

together back-to-front. The required OpenGL 4/DirectX 11 features are atomic read/write operations in shaders which enables us to capture multiple samples per pixel during one rendering call.

3.5.1 Depth Peeling

An older algorithm is called *Depth Peeling* [Mammen, 1989] [Everitt, 2001]. The main idea is to render the scene multiple times, "peeling" away one layer of surfaces at a time. In the first pass, the rendered image corresponds to the closest points to the camera for every pixel. In the second pass, the image corresponds to the second closest points and so on. Finally, these layers are blended together back-to-front. This approach does not require new hardware and only uses OpenGL 2 features. The problem with this method is, that transparent objects have to be rendered n times for n transparency layers.

A modified technique called *Dual Depth Peeling* [Bavoil and Myers, 2008] decreases the required number of rendering passes from n to $\frac{n}{2}$ by peeling away the closest and the farthest points in one pass.

3.5.2 Stencil Routed A-Buffer

There is a much more effective way of rendering transparent objects on OpenGL 3 capable hardware than Depth Peeling called Stencil Routed A-Buffer [Myers and Bavoil, 2007]. With the help of OpenGL framebuffer objects that can be used to render to multisample textures and a multisample stencil buffer, we can capture up to 32 layers of transparent geometry in one rendering pass. This is equivalent to the per-pixel linked list approach, but uses a multisample texture to store the samples, not one separate texture for every layer.

This algorithm's main idea is to use *stencil routing* which allows us to store output of a fragment shader into a sample whose stencil value differs from all other samples and thus allowing us to capture different fragment

values into different samples. After one pass of geometry, we have unsorted values for every pixel stored in different samples and a consecutive sorting and blending pass displays the correct transparent result.

At first, we initialize the alpha buffer (A-Buffer) by clearing color and depth values identically for all samples, but the stencil buffer has to be cleared with different values for each sample. By setting the sample mask, we limit rendering only into the k-th sample (this functionality is provided by the OpenGL sample mask feature). Afterwards, we set the stencil operation to always replace with the reference value set to $k+2$. We then need to perform rendering of a full-screen quad, since the sample masking is ignored by the `glClear` function and would rewrite values in all samples.

We repeat this step for every sample and after this initialization step (after drawing N full-screen quads), we have in each pixel N samples with stencil values set to $k + 2$ for the k-th sample.

During the A-Buffer generation step, we set the stencil operation to decrease whenever a fragment is being rendered - this will affect stencil values in all samples. The stencil function is set to "equal" with a reference value of 2. This setup allows the following: when a fragment is being rendered, the stencil reference value 2 is matched only with the first sample (other samples have values $3, 4, ...$). The computed fragment color and depth are therefore stored only in the first sample, and the stencil operation decreases values in *all* samples. This causes the second sample to have a value of 2 and the next fragment will be stored into the second sample. This process is shown in Figure 3.4. The 4 samples for a pixel after initialization step ready to write to the sample with stencil value $S = 2$ (top-left). When the first fragment is stored all stencil values are decreased by one, color $C = (R, G, B, A)$ and depth D of the first fragment are written into the first sample (top-right) and after next two steps (bottom-left, bottom-right) we have three valid samples that can be sorted afterwards. The sort simply takes the depth values and orders them from the largest number (farthest point) to the smallest number (closest point).

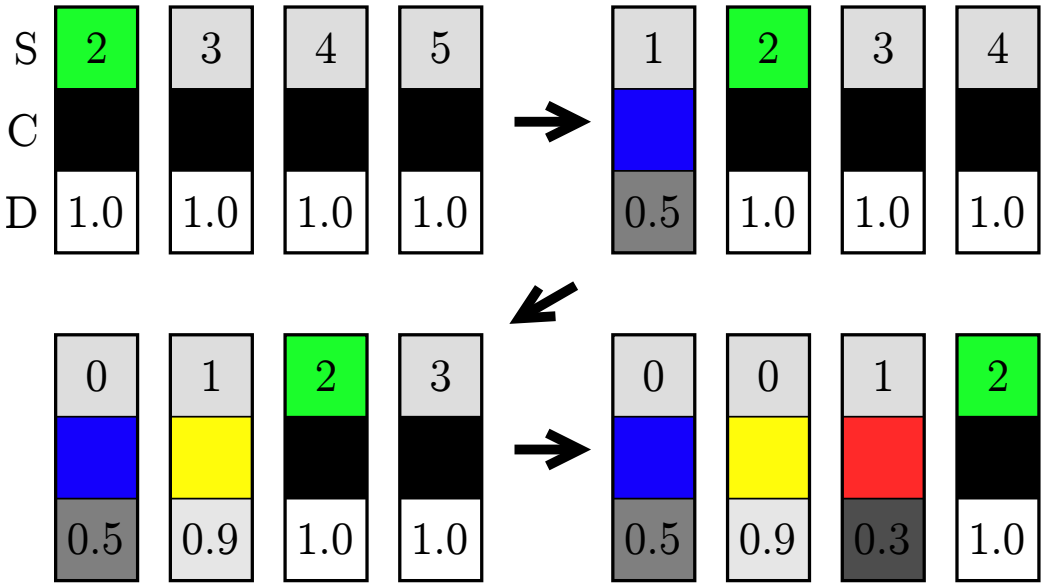

Figure 3.4: Stencil Routing for 4 samples per pixel

During the A-Buffer rendering step, we can directly access pixel samples one by one in the fragment shader. Once we have all color and depth values stored in an array, we sort the color values based on their corresponding depth. We take the sorted colors and perform back-to-front blending by iterating over all sorted values and evaluating the blending equation in the fragment shader.

There are several extensions that can be integrated into this solution. We always have the depth difference (and therefore distance) between two transparent samples and these distance values may be used for advanced lighting evaluation such as Beer's law, refraction, or other.

Chapter 4

Lighting Effects with Deferred Shading

Deferred Shading allows for a various number of effects. All these effects could be applied to a forward rendering engine as well, but the deferred approach has its advantages, especially those concerning light calculation.

4.1 High Dynamic Range Lighting

Current displays can only display a limited luminance range (usually RGB in range $[0, 1]$), however in real life, luminance has a much wider range, from very high values (bright sun) to almost no luminance at all (stars during night). The human visual system allows us to perceive both of these extreme luminance values thanks to our eyes' *light adaptation*, a process that adjusts our visual system to various levels of light or darkness. *High Dynamic Range Rendering* (HDR-R) is a series of algorithms that take into account the wide luminance range and try to simulate light adaptation. This allows for more realistic display of virtual scenes.

4.2 Tone Mapping

Tone mapping is a method of mapping luminance values of a HDR image (in range $[0, \alpha)$, where $\alpha >> 1$) to the displayable Low Dynamic Range (LDR) - $[0, 1]$.

Since we want to operate directly on luminance values, the RGB color space is not suitable due to the fact that the luminance value has to be calculated from all three components. For instance the Y component of the XYZ color space directly corresponds to luminance and luminance is unaffected by the XZ components. When applying tone mapping to a High Dynamic Range RGB image, it is customary to convert the image into the XYZ color space, perform tone mapping (by altering only the Y component) and then convert back to RGB.

There are many different tone mapping operators, divided into two categories:

- **global** - these operators take a single luminance transformation and apply it to every single pixel. This means that if two pixels have the same luminance value, their new luminance will be the same.
- **local** - these operators depend on the local properties of the image, applying a different luminance transformation in every pixel. As a result, two pixels with the same luminance value can be transformed into two different values.

We will now introduce two global operators that will be used in our deferred rendering engine. Probably the simplest global (apart from clamping luminance values) tone-mapping operator uses a direct compression of the $[0, \infty)$ interval into $[0, 1]$ using the following equation:

$$L_{tonemapped} = \frac{L_{input}}{L_{input} + 1}. \tag{4.1}$$

This transformation compresses higher luminance values more than lower luminance values and ensures that the output will be LDR. However, it desaturates colors and the output images have a grayish look.

4.2.1 Reinhard's operator

Reinhard's tone mapping operator [Reinhard et al., 2002] is a popular tone-mapping operator that analyzes how light or dark the target image is and tries to modify luminance values to retain information in dark as well as light parts of the image. We use only the global part of Reinhard's operator, the original paper also describes consecutive local steps that further improve image quality.

The *key value* of an image is a quantity determining whether the image is subjectively light or dark. The higher the key value, the brighter the input image is. A typical approximation of the key value used by many other similar methods is the *log-average luminance* of the image. It is computed as follows:

$$lum_{log} = \exp\left(\frac{1}{N}\sum_{x,y}\log\left(\delta + L_w(x,y)\right)\right), \tag{4.2}$$

where N is the total number of pixels in our input image, $L_w(x,y)$ is the luminance of the input image at pixel (x,y), and δ is a small offset value to avoid evaluating a zero logarithm.

Once we have the log-average luminance representing the scene's key, we want to map average values to the target key value, using the following:

$$L(x,y) = \frac{a}{lum_{log}} L_w(x,y). \tag{4.3}$$

a is a user-defined value controlling the targeted key value after processing. Values in range $[0,1]$ are acceptable, and as we can see, lower values result in a darker image.

This gives us intermediate values that we further process by compressing high luminance values using the simplest tone-mapping from Equation 4.1.

The original paper suggest modifying this simple equation to provide better quality and avoid desaturation by introducing the value L_{white}, which is the smallest luminance value that will be mapped to a pure white color.

The final tone-mapped luminance is then computed as:

$$L_d(x,y) = \frac{L(x,y)\left(1 + \frac{L(x,y)}{L_{white}^2}\right)}{1 + L(x,y)}, \tag{4.4}$$

which reverts into Equation 4.1 when $L_{white} = \infty$. L_{white} is usually set to the highest luminance value found in the image.

The authors further suggest to apply automatic local adjustments, however we do not use this part in our system.

4.3 Bloom

When looking at a dark object that is partially occluding a bright light source such as the sun, the light rays appear to "bend" at the object's edge and a fringe is visible (the brighter the light, the larger the fringe effect). This is caused by visual systems that include a lens, such as the human eye or a camera. This effect is called *bloom*, a very popular effect used in almost all new 3D computer games. It goes hand in hand with HDR, since only very bright lights produce such an effect. This effect can be observed in real life when looking directly into the sun through tree-tops or similar objects. A bloom fringe caused by a specular reflection is shown in Figure 4.1.

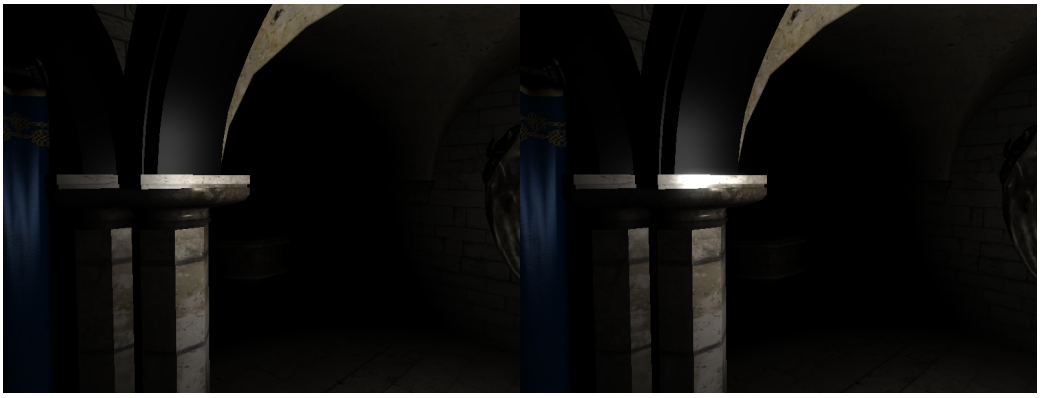

Figure 4.1: An image with specular reflection without showing bloom (left) and with bloom enabled (right)

Simulating bloom is straightforward and relatively easy to implement. Firstly, a luminance threshold is set and pixels that have a higher luminance value will produce a bloom effect. The threshold value is subtracted from each pixel's luminance. The next step is to perform an adaptive blur filter on these pixels, while taking into account the fact that brighter pixels get blurred with a wider kernel (resulting in a bigger bloom effect). Finally, the output is additively blended into the scene.

We use a separable gaussian filter which requires only $m + n$ texture reads to blur a pixel with a $m \times n$ kernel. Performance can suffer if the bloom effect is supposed to be hundreds of pixels in size. To avoid such problems, we create mipmaps before calculating the actual bloom effect and then compose the final image from different sized textures. If we were to blur the highest resolution mipmap, using a smaller kernel on one of the mipmaps with lower resolution can reduce the computational time significantly. The mipmaps are created using simple downscaling and bilinear interpolation, which makes the faster approach less precise. However, since the bloom effect is additively blended into the scene, the difference in visual quality is usually not noticable.

4.4 Shadow Mapping

Shadows are a crucial part of the scene lighting. They provide depth information, as well as helping us estimate the size of objects. For shadow generation, two viable real-time techniques are used: Shadow Volumes and Shadow Mapping.

Shadow Volumes [Crow, 1977] detects the silhouette of objects as seen from the light and then expands these edges away from the light. By doing so, mesh representations of the shadows are created. With the help of a stencil buffer, the shadows can then be added into the scene. The determination of silhouette edges is required to run every time a light or object moves, making it a CPU intensive approach for a highly dynamic scene. Shadow Volumes

will not be discussed further due to the CPU-bound operations.

Shadow Mapping [Williams, 1978], on the other hand, fully utilizes the GPU. From the light's point of view, a depth map (usually called the shadow map) is rendered and the depth information is then used to determine lit and unlit surfaces when rendering from the camera's point of view. It requires one additional rendering pass per light, but only the depth values need to be stored during this pass. When performing shading, the position in the shadow map (corresponding to the current point in world coordinates) is calculated and compared to distance from the light. If equal, the point is lit, otherwise the point is in shadow because there was an object in front of it when the shadow map was rendered. A scene with shadow mapping is shown in Figure 4.2.

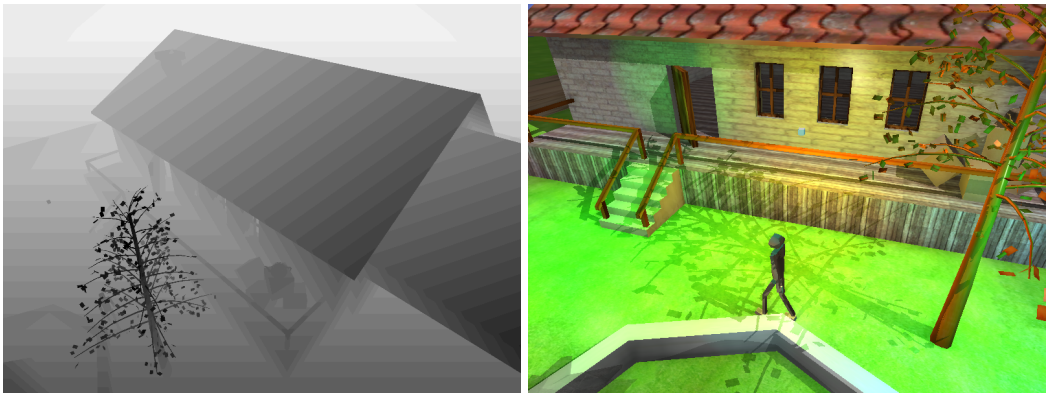

Figure 4.2: A shadow map from the light's point of view (left) and the shadowed scene (right).

In a single-pass forward rendering approach, all the shadow maps need to be stored in GPU memory at once since the shading evaluation accesses them all in one shader. However, in a multi-pass and a deferred approach, where lights are processed separately, only the shadow map required for the current light needs to reside in the GPU memory at that moment. This allows for a simple yet powerful memory optimization, where one depth texture can be reused multiple times for different shadow maps, overwriting the contents of the buffer every time we need the next shadow map.

The quality of results depends highly on the resolution of the shadow map as well as filtering techniques used when accessing the shadow map. A necessary optimization to reduce aliasing due to low resolution is to ensure that shadow maps are rendered only for parts of the scene that will be visible in the final image. There are several improvements of standard shadow mapping which address these problems.

4.4.1 Light types

Depending on light types (directional, point, spot), different projections and methods are used to generate shadow maps. For a directional light, an orthographic projection is used to generate the shadow map. For a spot light, a perspective projection is used while matching the spot cutoff angle to ensure correct results.

Point lights are trickier, since we want to generate omni-directional shadows. Multiple shadow maps are required per light. Usually, a shadow cube map (cube map composed of 6 shadow maps) is created for a point light by rendering into 6 different shadow maps and then the fragment shader accesses the correct shadow map based on light direction.

However, rendering the scene 6 times for one light is not optimal. Using two hemispherical projections instead reduces the number of scene renderings required to generate an omni-directional shadow map to 2 per point light, which is much more desired. This approach called *Dual Paraboloid Shadow Mapping* is described in [Brabec et al., 2002].

4.5 Ambient occlusion

Ambient occlusion is a technique that estimates the shadowing from ambient light in the scene. It provides much better depth information than simple shading, and the overall image quality is much higher, as can be seen in Figure 4.3. Ambient occlusion and shadows are both necessary for realistic display of scenes. Since ambient light illuminates surfaces uniformly from all directions,

the ambient occlusion term does not depend on the light's direction or light sources in the scene. This means that for a static scene (and dynamic lights), the ambient term can be pre-computed. For a dynamic scene, however, ambient occlusion has to be re-calculated every time an object moves.

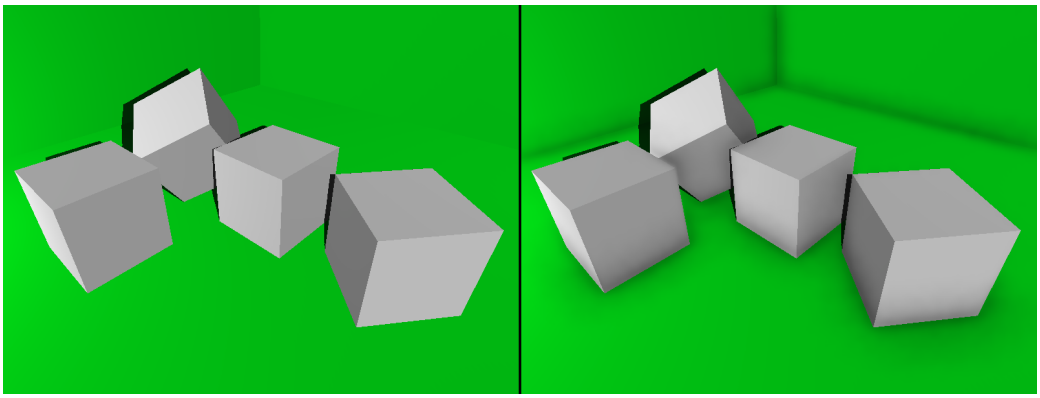

Figure 4.3: The effect of ambient occlusion. A scene without displaying ambient occlusion (left) and a scene with ambient occlusion displayed (right).

The computation takes into account nearby objects, therefore it is considered a global illumination technique. However, it can be approximated in real-time, as we show later.

Analytical ambient occlusion is computed by integrating the visibility function over a hemisphere centered at the surface point P and oriented by the surface normal $\hat{n}$ at P. The corresponding equation is:

$$A_P = \frac{1}{\pi} \int_{\Omega} V_{P,\hat{\omega}}(\hat{n} \cdot \hat{\omega}) \, \mathrm{d}\,\omega. \tag{4.5}$$

Ambient occlusion determines for a surface point how nearby objects block the incoming ambient light. Instead of integrating the visibility function, numerous rays are sent out into the scene from P in a hemisphere Ω oriented by the surface normal at this point. If a ray does not hit any object in the scene, it means that ambient light is reaching this point along this ray. The ambient occlusion factor is then equal to the percentage of rays that did not hit any surface. This process is shown in Figure 4.4.

We will be using a slightly modified version which includes distance checking. This is referred to as *ambient obscurance*, where we compute how many of the rays sent from the surface point do not hit a surface within a certain distance. Instead of rays "escaping" the scene, it is enough if the rays travel far enough without hitting a surface.

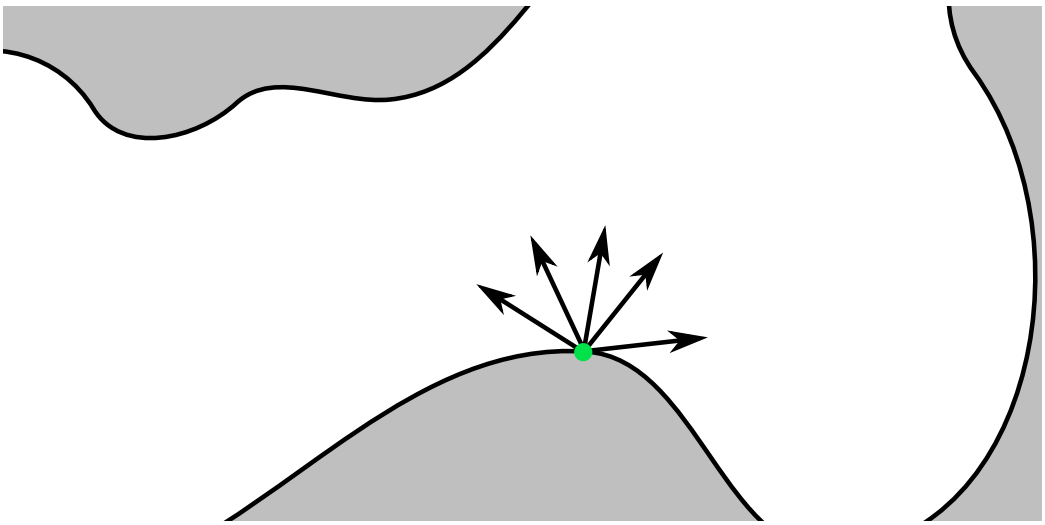

Figure 4.4: Computing ambient occlusion as an approximation by casting rays from the green point and checking which rays do not hit any surfaces.

The computed value is in range $[0, 1]$ and the ambient term (in the Blinn-Phong shading equation) is simply multiplied by the value when performing shading. The visual difference is seen in Figure 4.3.

4.5.1 Offline generation

Pre-computing the ambient occlusion term for static scenes is usually performed by using ray tracing. For every surface point, rays are cast into all directions based on the hemisphere and the nearest hit is stored. This distance information from every ray is then used to calculate the ambient occlusion term, where closer objects contribute more to the occlusion factor. If none of the rays intersect any object, there is nothing blocking the ambient light and the term remains 1 for a bright ambient light.

These results are usually computed for points uniformly distributed on the surface. The values computed are then stored into textures and can be used in a real-time application. This is similar to computing global illumination of static scenes and then storing these into light maps, but we do not store results of shading, only one of the input parameters for shading.

4.6 Real-time ambient occlusion

There are lots of techniques that produce real-time ambient occlusion, usually divided into two categories.

4.6.1 Screen-Space methods

These methods use the depth buffer and other screen-space data to estimate scene geometry and thus compute occlusion. One of the first methods was Screen-Space Ambient Occlusion [Mittring, 2007], from which most of these methods originated. We use a slightly modified version of this method that incorporates surface normals.

Amongst these techniques are those derived from the original SSAO either by changing the space in which computations occur or by using different sampling approaches or falloff functions.

There is the Horizon-Split (or Horizon-Based) Ambient Occlusion (HBAO) [Dimitrov et al., 2008] [Bavoil et al., 2008] which performs steps in tangent space and estimates the actual horizon as seen on the hemisphere from the evaluated point.

The Screen-Space Directional Occlusion (SSDO) [Ritschel et al., 2009] extends the original SSAO and as a result can evaluate one indirect diffuse bounce of light, allowing color leaking of nearby objects onto other objects.

The Alchemy SSAO implementation [McGuire et al., 2011] is derived from HBAO. Based on a different derivation of the ambient obscurance term, it allows for a more efficient and less noisy evaluation of occlusion.

Due to low sampling rate and the fact that the depth buffer contains no information about occluded geometry, all these methods produce significant noise and incorrect occlusion due to lack of information.

However, these methods have stable performance since every frame the number of evaluated samples and pixels remains the same.

4.6.2 Geometry-based methods

Contrary the previous methods, geometry-based methods rely heavily on the currently displayed scene. In [Bunnell, 2005], the authors generate disks from polygonal geometry and compute ambient occlusion between these disks (which is simpler than for general meshes).

The Ambient Occlusion Fields [Kontkanen and Laine, 2005] method provides a similar approach, where fields encapsulating the possibly occluding regions for meshes are generated and then evaluated during runtime. However, this method does not handle situations with deformable objects, since the fields are pre-computed.

An extension of Ambient Occlusion Fields allowing for fully dynamic occlusion is Ambient Occlusion Volumes [McGuire, 2010]. This method is part of our system and will be described later.

4.6.3 Screen-Space Ambient Occlusion

Screen-Space Ambient Occlusion (SSAO) was introduced by Crytek in [Mittring, 2007]. This approach utilizes the depth buffer of the scene (this is our G-Buffer depth) and based on depth values approximates scene geometry. This approximation is then used to evaluate the occlusion factor.

The main idea is to read the depth value of the current pixel, reconstruct it's 3D position, add several offset vectors to this position and reconstruct the screen-space coordinates of these points (as well as their depths). Finally, based on depth values stored in the depth map for these surrounding points, the occlusion factor is computed.

There are many different implementations that carry the label SSAO, and most of them differ quite significantly from each other. The term SSAO has been used in this context as an occlusion estimation algorithm that uses the depth buffer and screen-space to do so.

Certain implementations use only the depth buffer, comparing and combining depth values around the pixel. Other implementations use the depth buffer to actually reconstruct 3D points around the current pixel and derive occlusion based on the position of those points.

A nice improvement when using deferred shading is to take into account the normals as well (taken from the G-Buffer normal texture), thus improving the occlusion calculation. When we try to estimate occlusion at a point, the point's normal tells us how the sampling hemisphere around the point should be oriented. Thanks to that, we do not need to sample points in a sphere around the point (and probably taking into account points that would never have contributed to the occlusion), but we can take a hemisphere and thus sample points at twice the density while performing the same amount of operations.

It is relatively easy to implement in a fragment shader:

1. Read the pixel's depth value from the depth buffer.

2. Reconstruct the 3D position of this point.

3. Construct a fixed number of points around this point in 3D, reconstruct their screen-space position by projecting onto the viewing plane. We get a position in the depth map as well as a new depth value for this point.

4. For every point, if the depth value is higher than the value stored in the scene's depth map, increase occlusion factor based on distance from the center point.

When the depth value is higher than the value stored in the depth map means that there is an object in front of the point and we assume that this

point belongs to the object and thus occludes P. This process is shown in Figure 4.5.

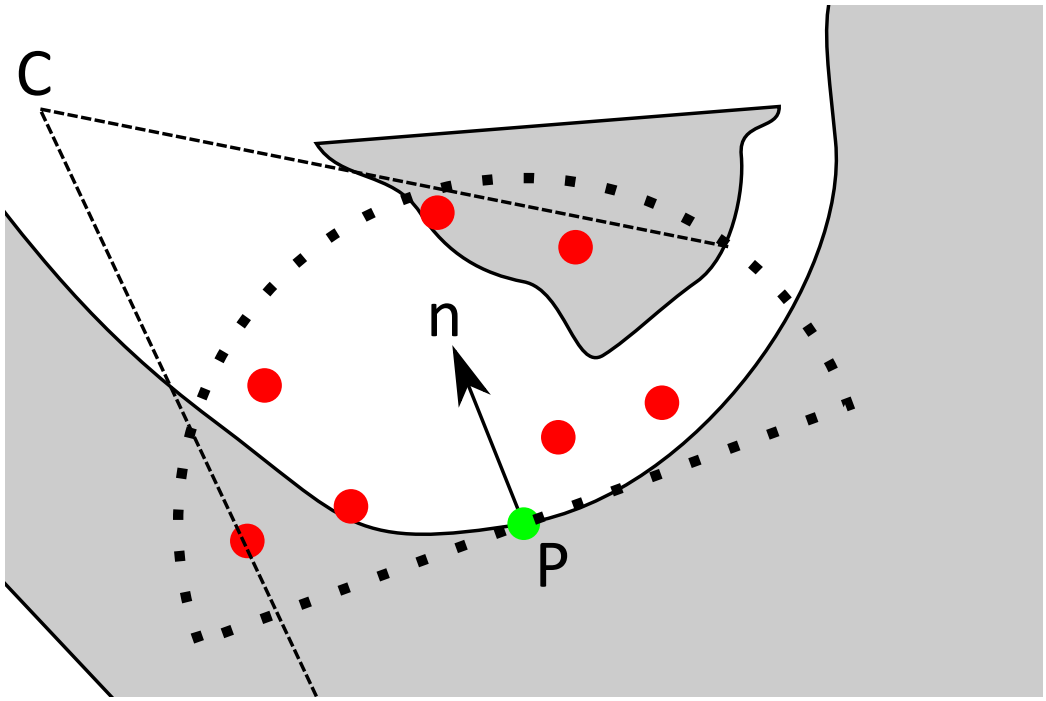

Figure 4.5: The process of evaluating SSAO

We use a falloff function based on distance of the occluding point, since objects that are closer contribute more to the occlusion than distant objects. Our function looks like:

$$f(d) = \frac{1}{1 + d^2}, \tag{4.6}$$

where the input d is the object's distance.

This approach produces significant ghosting in the occlusion buffer as seen in Figure 4.6 on the right image. Firstly because it is a very rough approximation of actual ambient occlusion and secondly because we need to keep the amount of sampled pixels to a minimum, otherwise it will take too long to compute. This ghosting can be avoided by randomizing offset vectors per pixel. We replace ghosting artifacts with high-frequency noise (Figure 4.6 left) but we will remove this noise with a blurring pass.

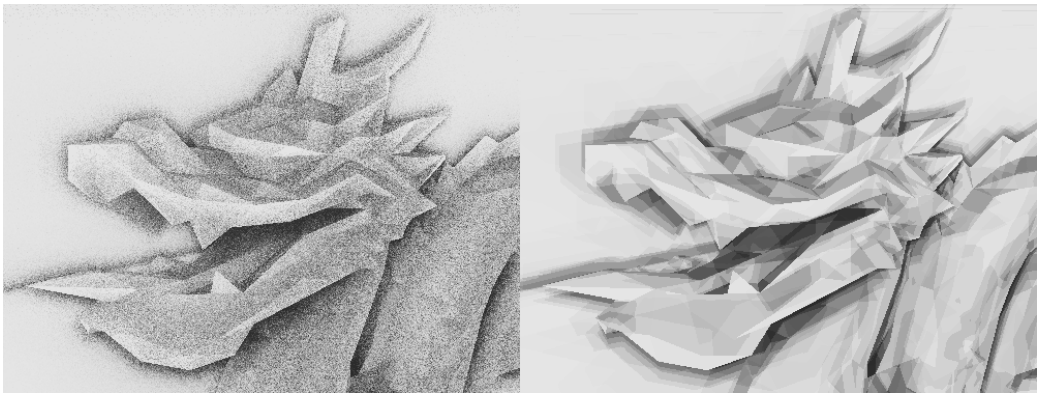

Figure 4.6: SSAO without randomized offset vectors produces ghosting artifacts (right). If we randomly rotate offset vectors in every pixel, we get a noisy result instead which will then get blurred (left).

Finally, the most important noise reduction is smoothing with a separable blur. Smoothing can reduce the noise to a minimum (Figure 4.7), but we have to be aware that smoothing can produce blurry artifacts between two objects. Imagine an object with ambient occlusion and a few pixel away from this object is an object that is actually hundreds of meters away. A simple gaussian blur would cause the occlusion of the close object to "leak" into the distant object. This is shown in Figure 4.8.

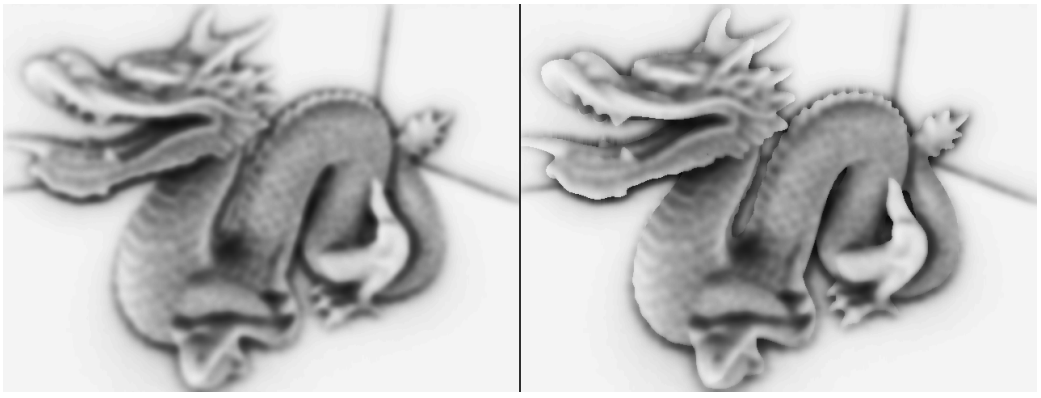

Figure 4.7: Smoothing of the SSAO buffer with a gaussian blur (left) and a cross-bilateral blur (right).

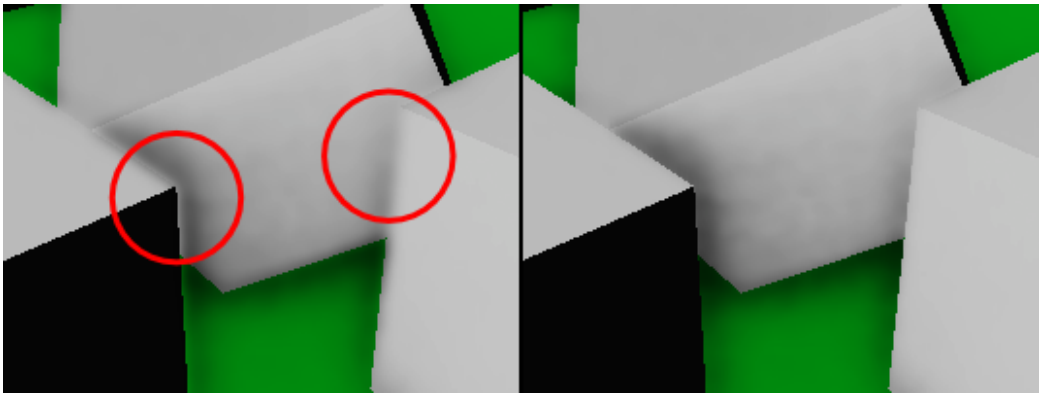

Figure 4.8: Errors caused by gaussian blurring of the SSAO buffer (left) and correction of the occlusion leaking by using a cross-bilateral filter (right).

Therefore, we use a cross-bilateral blur [Tomasi and Manduchi, 1998]. The depth buffer is used to analyze depth discontinuities when blurring the image and adjusts gaussian weights based on depth difference between the center pixel and other pixels. We multiply all gaussian weights with this expression:

$$w(current) = \frac{1}{\delta + |d_{center} - d_{current}|}, \tag{4.7}$$

where d_{center} is the depth of the center sample, $d_{current}$ is the current sample's depth and $\delta > 0$ is a small constant to avoid division by zero.

While computing all sample weights, we sum them together to re-normalize the values so that the final sum of all pixel weights is 1.

We perform a horizontal and a vertical blurring pass as we normally would with a separable gaussian blur.

When the camera moves, pixel neighborhoods change and it can result in unwanted jittery artifacts when moving the camera. For static images the cross-bilateral blur usually reduces the noise to an acceptable level but such a movement issue can still persist if we use a small blur kernel or not enough samples per pixel.

Despite the need to solve the noise problem, SSAO is a widely used effect which has quite good performance, but sometimes produces unrealistic artifacts (even after noise reduction).

The ideal number of samples should be as many as possible. However, we need to execute non-trivial operations including a texture read in the fragment shader for each sample. For low quality but high performance, we use 8 samples per pixel. For average quality, we use 16 and for high quality 32. Higher values are usually not practical due to performance issues.

Performing a full-screen SSAO with 32 samples per pixel is a challenge for newer hardware as well, and since the occlusion buffer will be blurred anyway, most implementations used buffers of size $\frac{w}{2} \times \frac{h}{2}$ or even $\frac{w}{4} \times \frac{h}{4}$, where w is the output image width and h is the output image height.

4.6.4 Ambient Occlusion Volumes

A different approach to real-time ambient occlusion is called Ambient Occlusion Volumes (AOV) [McGuire, 2010]. The main idea is to construct an *ambient occlusion volume* for every triangle and by rendering the triangle's ambient occlusion volume, we can cover objects that are potentially occluded by the triangle and evaluate occlusion contribution from this triangle.

Firstly, an ambient occlusion volume is similar to a shadow volume from the shadow volumes algorithm [Crow, 1977], but for ambient light. It is a triangular prism that is extruded from the triangle. This triangular prism is constructed by calculating outward facing normals e_1, e_2, e_3 of triangle edges - these vectors lie in the plane of the triangle. Afterwards, we normalize these vectors and multiply them by a constant δ which determines how far the occlusion will reach.

For every vertex of the triangle, the outward facing normals (corresponding to edges that contain the vertex) of length δ are added to the vertex position. This operation produces three new vertices V_1, V_2, V_3 which will be the base of the occlusion volume. The other three vertices V_4, V_5, V_6 are then constructed by adding the triangle normal (rescaled to length δ) to each of the three vertices. This process is depicted in Figure 4.9.

By manipulating the AOV size δ, we can easily control how far from the triangle the occlusion is generated. When the camera is inside an AOV, it

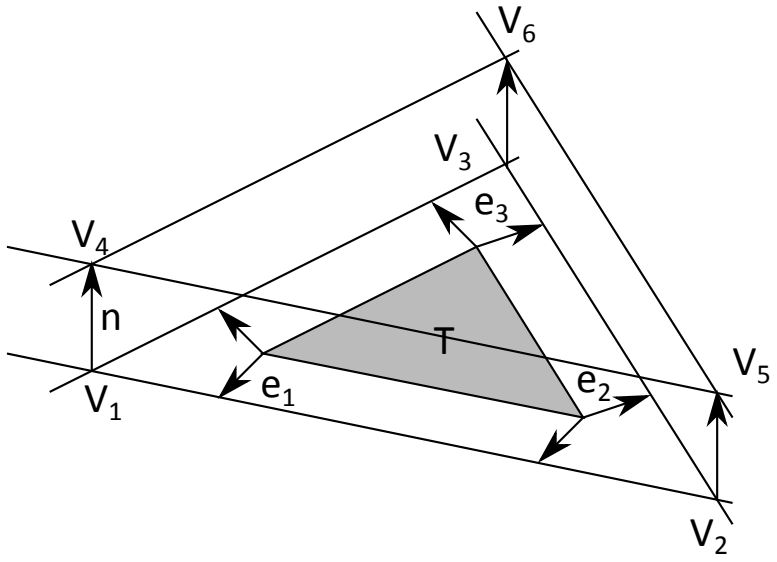

Figure 4.9: The construction of an Ambient Occlusion Volume for a triangle

has to be replaced with a fullscreen quadrilateral since all visible pixels are potentially occluded.

Due to that, setting the AOV size to a large value can cause most of the volumes to become fullscreen quadrilaterals and this can end up in rendering a fullscreen quadrilateral for every triangle in the scene. This would of course take up a lot of performance, therefore the AOV size should not be too large (and also not too small, because the effects will become much less noticeable).

By generating AOVs for the whole scene, we get a layer of AOVs on top of every mesh. Each of these AOV "remembers" it's originating triangle and it's area. When rendering these volumes, we get position (x, y, z) and normal (n_x, n_y, n_z) from the G-Buffer and calculate occlusion by calculating the form-factor between the point (x, y, z) with normal (n_x, n_y, n_z) and the AOV's originating triangle. The calculated form-factor is then attenuated by a falloff function (which determines how the occlusion looses it's effect with increasing distance from the triangle). All occlusion from neighboring triangles is then subtractively blended into the accessibility buffer, which holds information about how each pixel is accessible by ambient light (initially 1 for every

pixel). Finally, the ambient term is modulated by the accessibility buffer when performing shading.

This approach does not produce such noisy results as SSAO. Since the computation is not performed in screen-space, the jittery artifacts are not present at all. Also the noise caused by sparse sampling is not generated, since we do not sample anything. This also means that there is no need to perform a blurring step. AOV avoids all the quality problems that were introduced by SSAO but may take longer to compute since it relies on scene complexity. Despite the fact that SSAO performs better in certain situations, AOV is still a real-time technique and thanks to lower noise is more viable for the future than SSAO.

Reducing the accessibility buffer to a smaller resolution can save computational time but introduces blurry artifacts. These can be avoided by performing a cross-bilateral upsampling while accessing the full-screen depth buffer to detect edges. Rendering at full resolution always results in slower execution than SSAO. Apart from hard edges, the ambient occlusion term is usually smooth, therefore this does not reduce the quality of AO much.

A visualization of the AOVs and the resulting image is shown in Figure 4.10.

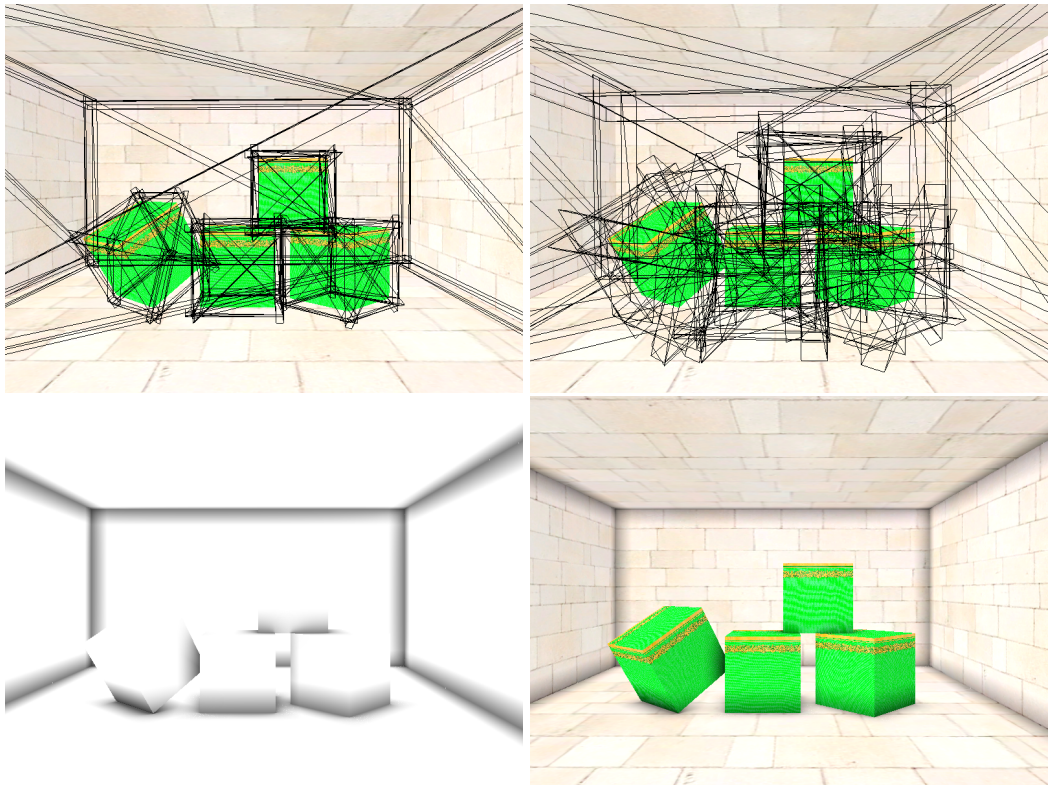

Figure 4.10: Visualization of the AOVs. Wireframe AOVs of size 0.3 overlayed in the scene (top-left), AOVs of size 1.0 (top-right), accessibility buffer for radius 1.0 (bottom-left) and the shaded scene (bottom-right).

Chapter 5

Implementation

We will now describe our implementation. The implementation was written in C++ using OpenGL version 3.3 (however, most algorithms can actually perform on older versions of OpenGL).

5.1 Forward Shading

Firstly, we will describe the forward shading approach. When compared to the deferred shading approach, it is much simpler, since forward shading is the way that OpenGL was originally designed to accelerate. This approach is compatible with much older hardware and OpenGL versions, since it only requires the usage of GLSL shaders (which are part of OpenGL core since version 2.0 and can be accessed through extensions on certain hardware not supporting OpenGL 2).

5.1.1 Single-pass Lighting

We already described how the single-pass lighting works. The scene is rendered once with our lighting shader, which has information about all the lights at once and shades each pixel during one run of the fragment shader. A simple pseudo-code is shown in Listing 5.1. The `shader` object consists of a vertex shader and a fragment shader. In the vertex shader, vertices, normals

and texture coordinates are transformed. Depending on what suits us the best, we can perform lighting computation in world space, or in view space. Either way, in the fragment shader, we require the actual position of the point currently being rendered (in the correct coordinate space), it's normal, and material properties.

The light information and material properties are stored in uniform variables. The material properties are as described by the Phong (Blinn-Phong) shading model: ambient and diffuse color (usually being the same), specular color and specular exponent (or shininess). When the objects have a surface texture, the color read from this texture usually replaces the diffuse and ambient material terms. However, by multiplying the texture color with stored diffuse and ambient color, we can provide *tint* for objects, giving them for instance a reddish look. For each light, we store similar information: light type, ambient color, diffuse color, position(, direction, intensity). Depending on the light type, we only need some of the information. For a directional light, the light's position is not necessary and it actually becomes a direction. For a point light, we require only the light's position as the light is omni-directional. For a spot light, both the position and direction (together with the spot cutoff angle) are used to compute final lighting.

The shaders for single-pass lighting are showed in Listings 5.2 and 5.3. Note that in the vertex shader, we use uniform matrices to transform vertices in multiple composed versions. This is a simple optimization of the shader execution, where matrices Model (Object space -> World space), View (World space -> View/Camera/Eye space) and Projection (View space -> Clip space) are composed once on the CPU and then sent to the GPU. Otherwise, we would have to replace `modelviewprojection` with `projection * view * model`, which wastes GPU cycles on multiplying the same matrices thousands of times every frame. We prepare all needed data for the fragment shader to compute lighting in world space - light positions and directions are sent in world space and actual position is computed in world space as well.

Notice in the fragment shader that we have lots of information for every

```
shader.Use();

for (each light)
    shader.AddLightInformation(light);

for (each object)
   shader.SetMaterialInformation(object.material);
   render(object);
```

Listing 5.1: Single-pass Lighting - CPU side

```
struct Matrix
{
   mat4 model, view, projection;
   mat4 modelview, viewprojection;
   mat4 modelviewprojection;
};
uniform Matrix matrix;

in vec3 attrVertex;
in vec3 attrNormal;
in vec2 attrCoord;

out vec3 position;
out vec3 normal;
out vec2 coord;

void main()
{
   coord = attrCoord;
   normal = vec3(matrix.model * vec4(attrNormal, 0.0));
   position = vec3(matrix.model * vec4(attrVertex, 1.0));
   gl_Position = matrix.viewprojection * vec4(position, 1.0);
}
```

Listing 5.2: Single-pass Lighting - Vertex shader

light. Therefore, due to hardware limitations on number of bytes available for storage of uniform variables, only a fixed number of lights are possible to render. And this fixed number is usually very low.

Other limitations that we might run into using this approach is a limit for number of operations in a shader. If we had hundreds of lights, the fragment shader would need to execute thousands of operations, surely exceeding this limit.

```
struct Material
{
   vec3 diffuse;
   vec3 ambient;
   vec3 specular;
   float shininess;
};
uniform Material material;

struct Light
{
   int type;
   vec3 position;
   vec3 direction;
   vec3 diffuse;
   vec3 ambient;
   vec3 specular;
   float spotExponent;
   float spotCosCutoff;
};

#define MAX_LIGHTS 8
uniform int activeLights;
uniform Light lights[MAX_LIGHTS];

in vec3 position;
in vec3 normal;
in vec2 coord;

out vec4 outColor;

void main()
{
   normal = normalize(normal);
   outColor = vec4(0, 0, 0, 0);
   for (int i = 0; i < activeLights; i++)
         outColor += shade(lights[i], material, position,
            normal, coord);
}
```

Listing 5.3: Single-pass Lighting - Fragment shader

The `shade` function simply fills the Blinn-Phong equation with the right data as supplied, and the final lighting is computed based on the light type. Thanks to the additive nature of lighting, we can sum contribution from all lights and return the color.

5.1.2 Multi-pass Lighting

We can see in the previous approach the numerous limitations as well as wasting GPU cycles when one pixel might be only affected by one light, but we have to evaluate them all the in fragment shader.

Multi-pass lighting works in a slightly different manner, but the resulting image is the same. The scene is rendered once for each light and the result is then additively blended together.

This allows for certain optimizations. First of all, the depth values generated in every single pass will be the same, so we do not need to generate the depth buffer every time. We can (during the first pass) render to the depth buffer as usual (first clear it with 1.0, then set the depth buffer function to „less than or equal" and enable writing to the depth buffer). In the consecutive passes, we disable depth writing (since we would only be writing the same values over and over) and set the depth function to „equal". This is the early depth optimization mentioned earlier, slightly modified to suit multi-pass lighting.

We can still optimize a little more, since not all lights affect all objects. If we bound every object and every light's volume with bounding volumes and then test for intersection, we can reduce the number of drawing calls. The code will look like the one in Listing 5.4.

We use additive blending thanks to the additive nature of light, and since this type of blending is not order-dependent, it does not matter in which order the objects are rendered.

The vertex and fragment shaders are the same as in single-pass lighting, except that the fragment shader has information only about one light at a time, and the summation of light contribution is executed with the help of blending.

```
// Early depth for all objects
glDisable(GL_BLEND);
glColorMask(0, 0, 0, 0);
glDepthMask(1);
glDepthFunc(GL_LEQUAL);
simpleShader.Use();
for (each object)
   render(object);

// Multi-pass lighting
glColorMask(1, 1, 1, 1);
glDepthMask(0);
glDepthFunc(GL_EQUAL);
glEnable(GL_BLEND);
glBlendFunc(GL_ONE, GL_ONE);
shader.Use();
for (each light)
   for (each object)
      if (intersect(light volume, object volume))
      {
         shader.SetLightParameters(light);
         shader.SetMaterialParameters(object.material);
         render(object);
      }
```

Listing 5.4: Multi-pass Lighting - CPU side

5.2 Deferred Shading

Deferred shading works in a similar manner than multi-pass lighting but avoids the need to render objects multiple times. Instead, the light volumes are rendered when generating lighting.

Firstly, we create a framebuffer object (FBO) for the G-Buffer and attach multiple textures, which define the layout of our G-Buffer. In the current implementation, we store this information:

1. **Albedo** or texture color - 24-bit RGB texture - 8 bits for each channel
2. **Depth** - 24 or 32-bit floating point texture
3. **Normal** - 2 x 16-bits fixed point (as described in Section 3.1)
4. **Specular color** - 24-bit RGB texture

5. **Specular roughness** - 8-bits

6. **Pixel type** - 8-bits

The hardware supports 4-channel textures, and even if we requested a 3-channel 8-bit texture, we might end up with a 4-channel texture. Therefore, we pack the data into several 4-channel textures.

Albedo and pixel type are in a single texture, the two 16-bit normal components could be packed into a 4-channel 8-bit texture, but it requires additional overhead when unpacking. The pixel type is currently used only to determine which pixels belong to the sky and should not be affected by shading.

The last texture contains specular color in three channels and specular roughness in the last channel. Several implementations revert to only monochromatic specular color and use the two free channels for other data. The Killzone 2 implementation [Valient, 2007] uses these free channels to store screen space motion vectors and then computes a motion blur.

For performance purposes, we attach the L-Buffer accumulation texture as well to forward shade the directional light(s). However, if we use SSAO or AOV, this step cannot be executed since these techniques require the G-Buffer and the lighting phase requires the accessibility buffers.

Together, it makes 15 - 16 bytes per pixel depending on the depth precision. For a 1024x768 G-Buffer, that takes only about 12MB of GPU memory. For a Full HD 1920x1080 G-Buffer, it takes almost 32MB.

Rendering into the G-Buffer requires only simple shaders. The vertex shader does only the usual - transform vertices, normals and texture coordinates. The fragment shader stores material data into output buffers, as shown in Listing 5.5, and iterates over all directional lights (we currently use only one) to shade them accordingly.

```
uniform sampler2D diffuseMap;
uniform sampler2D normalMap;
uniform sampler2D specularRoughnessMap;
uniform sampler2D specularColorMap;

uniform int objectType;

in vec2 texCoord;
in mat3 tbnMatrix;

out vec4 albedoAndPixelType;
out vec4 packedNormal;
out vec4 specular;
out vec4 lighting;

void main()
{
   albedoAndPixelType.rgb = texture(diffuseMap, texCoord).xyz;
   albedoAndPixelType.a = objectType;
   specular.rgb = texture(specularColorMap, texCoord).rgb;
   specular.a = texture(specularRoughnessMap, texCoord).r;

   vec3 normal = texture(normalMap, texCoord).xyz;
   normal = normalize(tbnMatrix * normal);
   packedNormal = packNormal(normal);

}
```

Listing 5.5: G-Buffer generation - Fragment shader

5.2.1 Generating the L-Buffer

Furthermore, we require a lighting buffer (L-Buffer). If we were to perform no post-processing on the L-Buffer result and avoid HDR rendering, the L-Buffer is not necessary and we can render directly into the window's framebuffer.

However, we want to utilize the L-Buffer data as a texture to allow reading from it, therefore we create a FBO for it as well. We attach two textures: RGB color - 16bpp, 16-32bpp depth. We use 16-bit color to allow direct HDR rendering support. This requires an additional tone-mapping step to display the result.

L-Buffer generation is performed by rendering light volumes assigned to different light types, using the same projection as was used to generate the

G-Buffer. In the fragment shader, every pixel reads data stored in the G-Buffer to generate the final color of the current point lit by the current light. The results are additively blended into the L-Buffer.

For different light types, different light shapes are rendered. When rendering directional lights, the light volume is a full-screen quadrilateral. This means that every single pixel from the G-Buffer is processed. Therefore, rendering directional lights usually adds additional overhead when using deferred shading, and several implementations revert to forward shading. Ideally, the pixels could be forward shaded when rendering the G-Buffer by attaching another texture and storing shaded values into this texture.

When rendering the sphere, we need to enable front-face culling to avoid problems when the camera is inside the sphere. For every pixel, the actual 3D position (in view space) is reconstructed from the G-Buffer depth and the distance from the light's position is evaluated. If the distance is larger than the actual light radius, the point is outside of the light's volume and should not be lit at all.

Spot lights are represented by cones. The tip of the cone is centered at the light's position, but we also need information about the spot direction and the cutoff angle. From these informations, we can again perform a test if a point is inside the light volume and apply shading based on the result.

The actual test is quite simple. We have the angle α_{cutoff} defining the cone and we have the direction of the cone $\vec{d} = (d_x, d_y, d_z)$. If we take the point to be tested and construct a vector $\vec{p}$ from the cone vertex V to the point. We just measure the angle between $\vec{d}$ and $\vec{p}$ and compare it to the cutoff angle. The situation is shown in Figure 3.3.

5.3 Tone-mapping

Lights can have intensity higher than 1. Such a light would probably cause certain values in the L-Buffer to exceed 1 (that's why we used a 16-bit RGB L-Buffer). However, we need to map these values into the displayable range

$[0, 1]$. That's where tone-mapping comes into play.

Usually, we need an additional pass to perform tone-mapping. We cannot do it during the L-Buffer phase mainly because when rendering the light shapes, we actually have no idea what to expect where. One point could be affected by hundreds of lights with huge intensity, and we can only process only one light at a time. This problem is not present in the single-pass forward shading approach, but the same approach as we will describe now is used for multi-pass lighting as well.

Tone-mapping has to occur when the L-Buffer is generated. We apply tone-mapping by rendering a fullscreen quad which then reads values from the L-Buffer.

The actual tone-mapping is executed in a fragment shader as shown in Listing 5.6.

```
uniform sampler2D lBuffer;
uniform float logAverageLuminance;
uniform float lWhite;

in vec2 pixelCoord;
out vec4 outColor;

void main()
{
   vec4 inColor = texture(lBuffer, pixelCoord);

   outColor = ApplyReinhard(inColor);
}
```

Listing 5.6: Tone-mapping shader

The simplest tone-mapping from Equation 4.1 is very easy to implement but does not produce high-quality results since colors are usually desaturated.

5.3.1 Reinhard's operator

However, Reinhard's operator requires two values that are the result of image analysis on the L-Buffer. The log-average luminance is affected by all pixels in the image and the L_{white} should be set to the highest luminance value

found in the image. Transferring the L-Buffer image into CPU memory and iterating over all pixels is not a very CPU-friendly solution and the need to transfer data back from the GPU every frame is unacceptable due to performance issues.

We use a GPU-accelerated computation of both values. The simpler of both computations is getting the L_{white} value. We perform several down-scaling passes similar to mipmap creation, but we compute the resulting luminance as the maximum of a fixed neighborhood (instead of the mean used when creating mipmaps). Once we reach a 1x1 result, we have the L_{white} value and we can either store it into an uniform value or directly read it from the created texture.

The log-average luminance works very similar. But during the first down-scaling, we first convert luminance values L_w into $\log(\delta + L_w(x, y))$ and then sum all these values together. This way, we perform piece-wise results of the sum in Equation 4.2. The other down-scaling passes only sum values in a fixed neighborhood. Again, the resulting 1x1 texture at the end of the down-scaling is used to retrieve log-average luminance of the image in constant time.

The tone-mapping shader simply fills these values into the equations and outputs the result.

5.4 Bloom

Bloom is usually coupled with HDR. We want to apply a blur filter to high luminance values and simulate a nice effect. A straightforward approach first darkens the tone-mapped image and then blurs based on intensity - the higher the intensity, the wider the blur kernel. This would mean that a point that is extremely bright should be blurred with a kernel of hundreds of pixels, and even with a separable gaussian blur, it would take too much time and would not be stable.

A faster way to generate bloom is to generate mipmaps of the darkened

image, since mipmaps can be interpreted as blurred versions of the original image. Multiple mipmap stages are then summed together in the fragment shader and blended into the scene. This is again a full-screen pass.

5.5 Shadow Mapping

Shadow Mapping can be easily integrated into single-pass, multi-pass and deferred shading. It introduces an additional rendering pass from every light's point of view. For the final light computations, only a few shader instructions are added to determine lit and shadowed pixels.

For the shadow map, we create an FBO of arbitrary size (the higher the resolution, the lower the aliasing) with up to 32-bit depth and without a color texture, which is unused.

When the shadow map is ready, we can access it in a fragment shader when generating the L-Buffer. Since we store view-space normals and use the depth values to reconstruct view-space position, all the calculations occur in view-space. However, the shadow map was rendered from a different point of view. We construct a matrix transforming from the light's clip space (in which the shadow map is) to the camera's view space.

This matrix is constructed as $B \cdot P_L \cdot V_L \cdot V_C^{-1}$, where B is the bias matrix which transforms clip space coordinates from range $[-1, 1]^3$ to range $[0, 1]^3$, P_L is the light's projection matrix, V_L is the light's view matrix and V_C is the camera's view matrix.

By multiplying a 3D vector representing a point in view-space with this matrix, we transform it into world space (V_C^{-1}), then into the light's view-space (V_L) and finally into the light's clip space (P_L). This clip space is then mapped by matrix B into range $[0, 1]^3$ and the x, y coordinates are the position to query in the shadow map while the z coordinate is used for shadow map comparison.

5.6 Screen-Space Ambient Occlusion

Integrating Screen-Space Ambient Occlusion into the pipeline is relatively simple. All that is required is to access (during the lighting phase) the *accessibility buffer*, which determines how accessible points are for ambient light. The accessibility buffer is output from the SSAO pass, and then the ambient lighting term for each pixel is multiplied by the value corresponding to that pixel.

SSAO is just a full-screen pass that in every fragment accesses the G-Buffer depth and G-Buffer normal. Since the accessibility buffer needs to be generated when we render light shapes, we perform it before the L-Buffer pass. Based on the desired number of samples per pixel, we generate random vectors on the CPU and then send these through uniform variables into the fragment shader.

5.7 Ambient Occlusion Volumes

The integration of Ambient Occlusion Volumes is exactly the same as in SSAO. Output of the AOV is a one-channel texture containing the accessibility information. However, the generation of AOV is a bit more complicated.

First of all, we create a FBO for rendering the occlusion volumes. The depth buffer needs to be the same as the G-Buffer depth (to avoid contribution from occluded AOVs), therefore we should incorporate a shared depth buffer. However, the AOV algorithm requires access to G-Buffer depth during fragment shader execution and we need to disable writing into the depth buffer. Otherwise, we might be rewriting values that are currently being read by the shader, which will probably result in undefined behavior or runtime errors. The color buffer requires only one 8-bit channel which is enough for our accessibility buffer.

In this method, the geometry shader is used as well. If we wanted to generate occlusion only on static objects, the geometry shader would not be required since all it does is recalculate the occlusion volume for every single

triangle. In a static scene, this can be done once on the CPU and then the occlusion volumes will be just sent to the GPU to render.

However, in the fully dynamic approach, the occlusion volumes change every time objects move. If we excluded from the scene changes features like skinned characters (or other non-rigid objects) that modify the vertex positions in the vertex shader in a different manner than a simple transformation matrix, the occlusion volumes can be generated during pre-processing and the only thing that changes when animating are the transformation matrices for the occlusion volumes.

When using GPU skinning, morphing, etc., the geometry shader generates the occlusion volumes every frame. This does not mean that much of a performance issue, since the occlusion volume for a triangle is simple to generate in the geometry shader and only performs simple vector additions and multiplications. Generating it during pre-processing does not reduce the number of primitives that need to be drawn, nor does it change the number of pixels that need to be shaded. It only removes the need for the geometry shader.

The AOV pass rendering occlusion volumes requires access to G-Buffer depth and G-Buffer normal. Additive blending is used to accumulate occlusion contribution from different AOVs. Depth writing is disabled since we want to render intersecting occlusion volumes.

The vertex shader is a simple pass-through shader that stores the world-space coordinates of the vertex into `gl_Position`. The geometry shader takes as input a single triangle and outputs the corresponding occlusion volume as triangle strips. It simply computes the area and normal of the triangle (using the cross product) and extrudes the triangle in the direction of the normal by a specified amount - this is the maximum distance for a triangle to contribute to occlusion of a different triangle. Sides of the created triangular prism need to be extruded as well using normals corresponding to the triangle's edges and lying in the plane defined by the triangle.

If the camera is within one of the occlusion volumes, we need to replace the

occlusion volume with a full-screen quad because all pixels can be potentially occluded by this occlusion volume. This is also done in the geometry shader.

The fragment shader receives non-interpolated edge normals, triangle normal and triangle area from the geometry shader. Based on pixel coordinates and depth, it reconstructs the 3D position in view space and applies a form-factor computation to determine how occluded the current pixel is by the triangle corresponding to the rendered AOV.

5.8 Depth Peeling

Depth peeling executes N full-screen passes over transparent geometry, capturing the closest surfaces in the first pass, the second closest surfaces in the second pass etc. This is performed with a simple algorithm. First, we render the transparent objects as usual, we receive information about the closest points. Then, we use the depth information stored in this pass to discard fragments in the shader that would belong to the first layer of transparent objects. We always use the last generated texture to discard fragments whose depth is less than or equal to the value stored in the last texture.

The reconstructed layers are sorted front-to-back, so we only need to composite them in reversed order.

5.9 Stencil Routed A-Buffer

Stencil Routed A-Buffer allows to capture multiple layers of transparent geometry in one pass. It requires OpenGL 3.3 features allowing rendering into a multisample FBO. During initialization, we create a FBO with N samples per pixel and attached color, depth and stencil. This FBO will be our alpha buffer (A-Buffer).

The associated shaders are simple. The fragment shader used when rendering into the multisample FBO only performs forward shading on several lights and stores the resulting RGBA color. The fragment shader used when

displaying the stored values reads all samples of a pixel by accessing directly the values stored in the samples. RGBA color and depth are recovered and the recovered depth is used for sorting.

Chapter 6

Results

We tested our system on several scenes, the testing machine had a Windows 7 Ultimate 32-bit, Intel Core 2 Duo E6550 2.33 GHz CPU, a NVIDIA GeForce 560Ti (1GB RAM) graphic card and 4GB of RAM. Our application is single-threaded at the moment, it was utilizing one core of the CPU. However, most of the computation should occur on the GPU and CPU speed should not be one of the main factors affecting resulting performance.

We tested the overall performance of algorithms described in this thesis and how enabling/disabling these effects has a hit on performance. We computed the average execution time of parts of our pipeline in milliseconds. These times were recorded during a guided walk-through with a camera through the scenes.

Table 6.1 shows several statistics about our test scenes. The total number of point lights in the scene, the average number of lights that affected a pixel during our guided walk-through, how many separate objects there are in the scene and finally how many triangles the scene contains. The test scene are shown in Figure 6.2.

The number of separate objects directly corresponds to the number of geometry batches, and the number of point lights corresponds to the number of batches used when rendering into the L-Buffer.

Scene	Lights	LPP	Objects	Triangles
House	7	0.86	82	11 563
Dragon	2	1.32	2	201 075
Sponza	25	3.45	385	285 583
Sibenik	1	0.14	15	75 165
	10	1.2		
	100	8.28		
	1000	10.32		

Table 6.1: General statistics about our test scenes. How many lights there were, the average number of lights per pixel (LPP), how many objects there were, and the total amount of triangles.

6.1 Shading

In the first part of our tests, we compared forward shading, multi-pass shading and deferred shading as described in Chapters 2 and 3. The results are highly dependent on how many lights were in the scene. For the tests we used a large amount of point lights. For small amounts of lights, or lights that take up big portions of the screen, the forward shading approach is the fastest one. As the number of lights increases (or the size of affected portions of the screen decreases), deferred shading performance increases up to the point where it outperforms both older techniques.

Multi-pass lighting works well for large amounts of lights and after the early-depth pass only when there are a few lights affecting an object. However, if we re-render complex objects hundreds of times, the performance drops rapidly. The optimized version of multi-pass lighting should perform without problems if there are only several lights affecting each object. But the worst case is that there will be $m \times n$ object renders, where m is the number of objects and n the number of lights. Our test nearly reached the worst case, almost all objects needed to be rendered for each light, resulting in terrible performance of the multi-pass algorithm.

Results of various tests are shown in Figure 6.1. Showing the Sibenik scene with different amount of lights and different resolutions. The corresponding number of lights that affect one pixel (LPP) can be seen in Table 6.1.

These results do not include a tone-mapping step and the L-Buffer is actually the window's framebuffer. These results also do not differ at all in terms of quality. The generated images for all three methods are identical because we evaluate the same shading equations for the same pixels, only in different order or avoiding evaluation where the result would be zero anyway.

The graphs for different resolutions are shown in Figure 6.1. Please note that there are missing values for forward shading with 1000 lights and resolution 1920x1080. Executing so many operations in a shader exceeds the 2 second execution limit and the driver crashes. Pushing more lights into a forward renderer is therefore not actually possible.

We also had to do several tradeoffs to fit the light information into the shader. We omitted several settings for lights to fit at least light positions and intensities into uniform variables. Therefore, the forward shading with 100 and 1000 lights does not allow full control of the light's properties.

Also, if we want to incorporate other techniques after forward or multi-pass shading, we need to render into an FBO and store normals if we will be utilizing them as well. This adds additional overhead, but it is not part of this evaluation.

6.2 HDR and Tone-mapping

How much overhead HDR rendering and tone-mapping adds is interesting as well. There is no direct support for forward shading or multi-pass shading, so we render the output of these methods into an FBO and then execute a tone-mapping step. We perform the same steps for all shading methods and the input is the same as well, therefore there will not be any difference in performance of this step. However, additional overhead is added to all

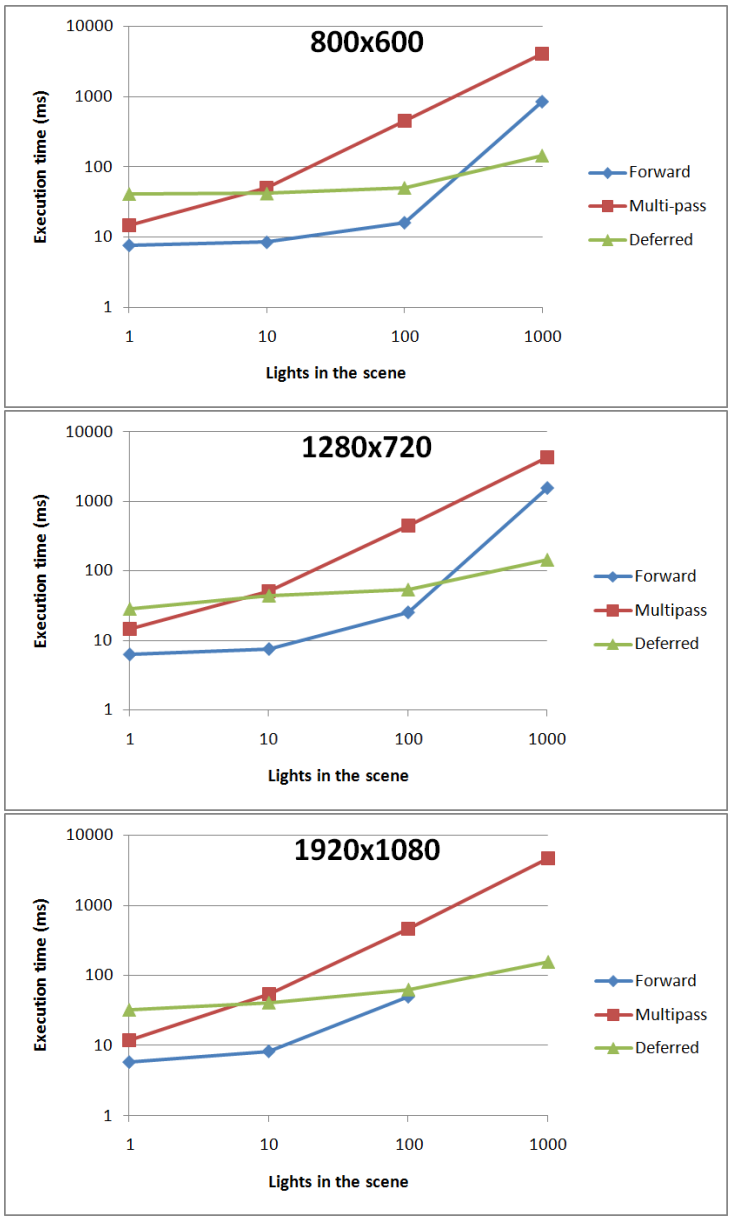

Figure 6.1: Graphs showing performance for different shading techniques and different amounts of lights.

shading methods due to the additional step that requires a texture to be generated and then the tonemapped data is displayed.

The only exception is forward shading when using a simple transform

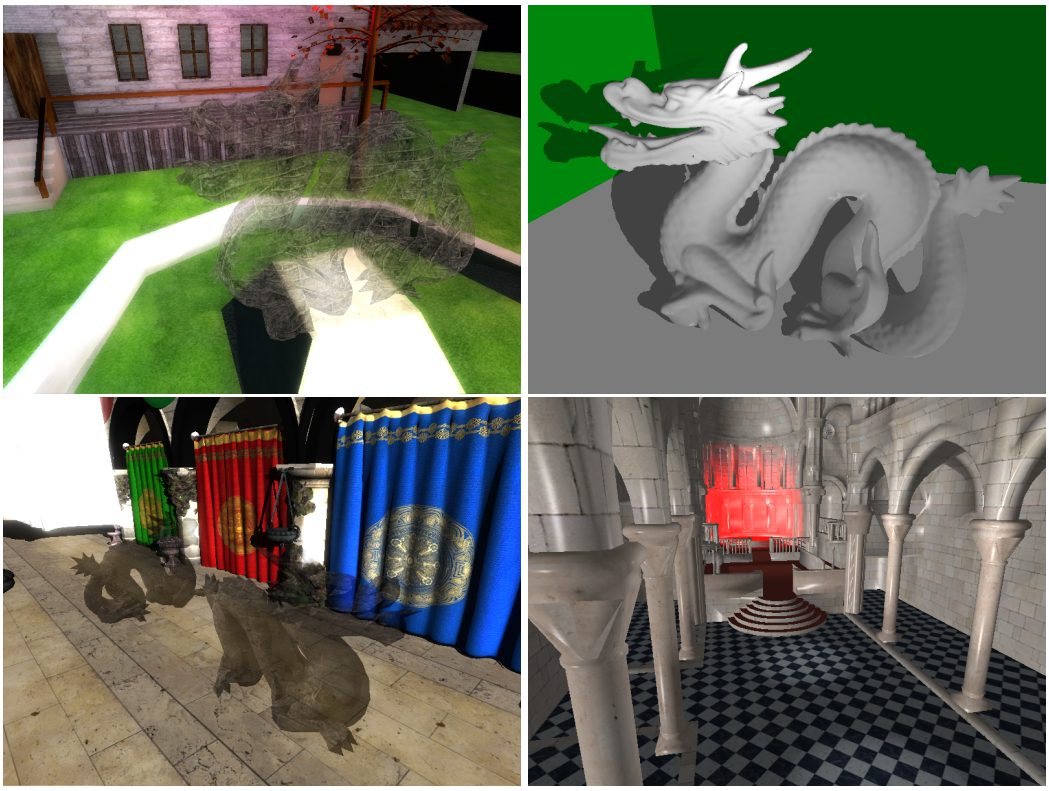

Figure 6.2: The scenes we used for testing. From left to right, top to bottom: House, Dragon, Sponza, Sibenik

that does not depend on the whole image but is pre-determined before tone-mapping. Such a tone-mapping step does not incur noticeable overhead for clamping or the simple luminance transformation in Equation 4.1.

We tested two tone-mapping operators. The simplest one from Equation 4.1 does not require any image analysis of the input image and outperforms Reinhard's operator by this step. The fragment shader evaluating the output LDR image also performs only a very small amount of operations when compared to the other two.

The execution time of these operators do not depend on the contents of the input image, only on it's size. We performed the tests on three static images from the same point of view in our scene, each of the images has a different resolution.

Method	800x600	1280x720	1920x1080
Simplest	0.16ms	0.21ms	0.34ms
Reinhard et al.	0.46ms	0.57ms	0.88ms

Table 6.2: Comparing performance of tone-mapping operators in several resolutions. The main difference in execution is the actual calculation of log-average luminance.

The visual quality of output images is different, reaching much higher realism and better looks for the Reinhard's operator. A comparison is shown in Figure 6.3.

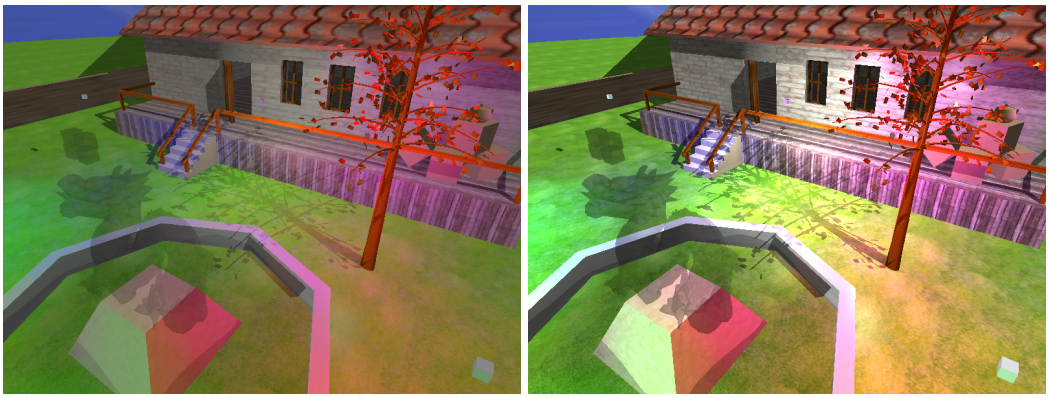

Figure 6.3: Showing different results produced by tone-mapping operators. Simplest tone-mapping (left) has desaturated colors and lower overall contrast, but Reinhard's operator (right) keeps a nice contrast and saturated colors.

6.3 Transparent Objects

The Depth Peeling approach takes quite long when there are complex scenes, since the scene is rendered N times. We set the Depth Peeling and Stencil Routed A-Buffer to correspond with the number of layers and compared

performance of the produced results. Both techniques capture the same data (unless an overflow occurs), therefore the resulting images look the same.

The Stencil Routed A-Buffer performs quite fast, however it requires a sorting step. In depth peeling, the layers already are sorted, thanks to the way we generate them. If there were a maximum number of samples in each pixel - such as a complex object very near to the camera - we would need to perform sorting for every single pixel and this results in performance spikes.

Also, the step that clears the color and depth buffers while setting the stencil sample values to ascending order takes up some of the performance. Especially due to the fact that we need to render N fullscreen quads for N layers. However, during this rendering we use the simplest possible shader and disable writing into the color and depth buffer to allow for faster rendering. There is currently no other way to generate the stencil values.

The test were done on the House scene, the walkthrough had a depth complexity of 0.67. Results are shown in Table 6.3.

Layers	**800x600**	**1280x720**	**1920x1080**
Depth peeling			
4	19.28ms	25.2ms	29.47ms
8	30.31ms	59.35ms	74.66ms
Stencil Routed A-Buffer			
4	17.28ms	23.2ms	27.13ms
8	26.45ms	43.67ms	52.51ms

Table 6.3: SSAO testing presets

On not very complex geometry, the increased performance is not that different, but if we used more layers, the difference would be much more noticeable (much like in multi-pass shading).

6.4 Ambient occlusion

Results for the two used ambient occlusion techniques - SSAO and AOV - should be very varying. SSAO is a screen-space method and therefore it is dependent mostly on the screen resolution and number of samples used. The occlusion radius also affects performance, since the smaller the radius, the closer the resulting samples will be to the center pixel and that allows the GPU to optimize texture fetches. The texture fetches close to each other are affected by the input image as well, since closer objects result in a larger hemisphere projection in screen-space.

On the other hand, AOV depends on the number of triangles in the scene, the occlusion radius, but on the camera position as well. If the camera is inside several occlusion volumes, a full-screen quad has to be rendered for each of these occlusion volumes and can result in hundreds of full-screen passes. If we keep the occlusion radius relatively small and the camera will not be inside of any of the volumes, the overdraw will not be as significant.

The AOV fragment shader performs form-factor computations and other necessary steps which is lots of steps as well. We use reduced sizes of the AOV buffer to save performance, just like with SSAO. The vertex and geometry shader stages are mainly affected by the complexity of the scene, while the fragment shader stage is affected by overdraw as well as AOV buffer resolution.

We used several settings for SSAO, their parameters are shown in Table 6.4.

The SSAO and AOV settings were set to produce results very close to each other. The AOV occlusion radius is set to 1, the AOV falloff factor is set to 1 and we only vary the resolution of the AOV buffer. For higher radii, the overdraw can be very significant. The different presets are shown in Table 6.5.

We tested all AO presets on a single scene with a guided walk-through on different resolutions. The results are shown in Tables 6.6 and 6.7.

While AOV does not produce any noise (at full resolution) and allows for

Preset	Occlusion Buffer	Samples	Radius	Blur width
SSAO1	$\frac{w}{4} \times \frac{h}{4}$	8	1	10
SSAO2	$\frac{w}{4} \times \frac{h}{4}$	16	1	15
SSAO3	$\frac{w}{2} \times \frac{h}{2}$	16	1	20
SSAO4	$\frac{w}{2} \times \frac{h}{2}$	32	1	20
SSAO5	$w \times h$	32	1	30

Table 6.4: SSAO testing presets

Preset	Occlusion Buffer
AOV1	$w \times h$
AOV2	$\frac{w}{2} \times \frac{h}{2}$
AOV3	$\frac{w}{4} \times \frac{h}{4}$

Table 6.5: AOV testing presets

Preset	800x600		1280x720		1920x1080	
SSAO1	5.95ms	1.01ms	6.13ms	1.22ms	12.92ms	1.38ms
SSAO2	8.75ms	1.78ms	15ms	2.8ms	16.19ms	1.85ms
SSAO3	8.83ms	1.74ms	9.61ms	2.24ms	11.78ms	4.75ms
SSAO4	11.5ms	7.9ms	8.67ms	3.46ms	16.6ms	9.55ms
SSAO5	12.06ms	9.48ms	17.96ms	13.88ms	41.75ms	32.75ms

Table 6.6: Comparing performance of SSAO in several resolutions and method presets. The results are shown as two values, the first one is the whole SSAO generation and the second one is only the generation of the SSAO buffer without blurring.

much more visually appealing results, the additional overhead and unstable execution time show that it is viable only for real-time display of scenes with small amounts of triangles. At lower resolutions, we sacrifice the precision to ensure fast execution.

Scene	Preset	800x600	1280x720	1920x1080
Sibenik	AOV1	62.4ms	101ms	198.2ms
	AOV2	40.5ms	64ms	142.2ms
	AOV3	26.9ms	43ms	113.8ms
Sponza	AOV1	192.4ms	253.5ms	348.2ms
	AOV2	174.9ms	212.8ms	300.1ms
	AOV3	121.4ms	141ms	231.7ms
Boxes	AOV1	5.8ms	9.54ms	14.7ms
	AOV2	3.2ms	6.7ms	10.6ms
	AOV3	1.92ms	4.6ms	7.8ms
House	AOV1	32.1ms	49ms	65.4ms
	AOV2	20.4ms	37.8ms	47.2ms
	AOV3	10.1ms	28.3ms	38ms

Table 6.7: Comparing performance of AOV in several resolutions and scenes.

The SSAO performance is very stable and in general faster than AOV. As we can see, AOV still strongly relies on the scene complexity, however this can change after view frustum optimizations. Setting the AOV radius to a plausible level is also crucial, otherwise there can be extreme spikes, ranging from 10ms to almost 800ms when inside tens of AOVs.

As we can see, full-sized AO buffers are overkill and since the occlusion is usually very smooth (except for edges), the full-screen accessibility buffer are unusable in real-time applications. A good performance/quality trade-off is to use $\frac{w}{2} \times \frac{h}{2}$ buffers with upsampling (and filtering in SSAO).

6.5 Overall performance

When putting all these techniques together, we get a complex system that executes several full-screen passes. Our goal was for the whole system to run at interactive frame rates, ideally at 60 FPS which is the standard refresh

rate of LCD displays.

For a larger scene with hundreds of thousands of triangles split into several hundred objects, and 20 lights with complex transparent objects, we measured approximately 16 FPS. We used the SSAO3 preset at 1280x720, with Stencil Routed A-Buffer at 8 samples per pixel, together with Reinhard's tone-mapping operator. The scene contained one shadow-casting directional light with a 2048x2048 shadow map as well as 25 point lights. From our tests, this setup seemed to provide high-quality shading while retaining plausible frame rates.

The target frame rate has not been reached, but the system will gain on power with increasing hardware capabilities and it requires a large amount of optimizing, especially in view frustum culling and occlusion culling, which will strongly improve the bottlenecks of our application.

Conclusion

In this thesis, we have presented a real-time renderer displaying realistic images at interactive frame rates on OpenGL 3 capable hardware. Our system does not rely on any kind of pre-computation and can display fully-dynamic scenes with animated lights and objects.

The core of our system is a Deferred Shading pipeline. Deferred Shading allows for hundreds of lights to affect the scene while saving performance by not evaluating pixels that will not be affected by a light.

Once the scene is shaded, we perform a tone-mapping step to map high luminance values into the displayable range. We use several different tone-mapping operators and our system can easily be extended with other operators as well.

To enhance visual quality and improve depth perception in the scene, we allow lights to cast shadows. Shadow are evaluated using Shadow Mapping, a fully GPU accelerated technique to render shadows. Shadow Mapping is integrated easily into a Deferred Shading pipeline and the deferred approach to Shadow Mapping even reduces memory overhead when using multiple shadow-casting lights as opposed to Forward Shading.

Deferred Shading does not directly support rendering of transparent objects, but we provide an alternative called Stencil Routed A-Buffer. It allows for up to 32 layers of transparent geometry to be acquired in one scene rendering pass.

Our system is currently not optimized and does not reach the necessary 60 FPS in most cases. However, we have shown that even for lots of dynamic

lights and lots of dynamic transparent objects, we can still achieve plausible frame rates. We expect to get 60 FPS after optimizing and on better hardware even for very complex scenes.

Bibliography

[Akenine-Möller et al., 2008] Akenine-Möller, T., Haines, E., and Hoffman, N. (2008). *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA.

[Bavoil and Myers, 2008] Bavoil, L. and Myers, K. (2008). Order independent transparency with dual depth peeling. Technical report, NVIDIA Developer SDK 10.

[Bavoil et al., 2008] Bavoil, L., Sainz, M., and Dimitrov, R. (2008). Image-space horizon-based ambient occlusion. In *ACM SIGGRAPH 2008 talks*, SIGGRAPH '08, pages 22:1–22:1, New York, NY, USA. ACM.

[Blinn, 1977] Blinn, J. F. (1977). Models of light reflection for computer synthesized pictures. In *Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '77, pages 192–198, New York, NY, USA. ACM.

[Brabec et al., 2002] Brabec, S., Annen, T., and peter Seidel, H. (2002). Shadow mapping for hemispherical and omnidirectional light sources. In *In Proc. of Computer Graphics International*, pages 397–408.

[Bunnell, 2005] Bunnell, M. (2005). Dynamic ambient occlusion and indirect lighting. In Pharr, M., editor, *GPUGems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, pages 223–233. Addison-Wesley.

[Crow, 1977] Crow, F. C. (1977). Shadow algorithms for computer graphics. In *Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '77, pages 242–248, New York, NY, USA. ACM.

[Deering et al., 1988] Deering, M., Winner, S., Schediwy, B., Duffy, C., and Hunt, N. (1988). The triangle processor and normal vector shader: a VLSI system for high performance graphics. *SIGGRAPH Comput. Graph.*, 22:21–30.

[Dimitrov et al., 2008] Dimitrov, R., Bavoil, L., and Sainz, M. (2008). Horizon-split ambient occlusion. In *Proceedings of the 2008 symposium on Interactive 3D graphics and games*, I3D '08, pages 5:1–5:1, New York, NY, USA. ACM.

[Everitt, 2001] Everitt, C. (2001). Interactive order-independent transparency. Technical report, NVIDIA Corporation. Available at http://www.nvidia.com/.

[Filion and McNaughton, 2008] Filion, D. and McNaughton, R. (2008). Effects & techniques. In *ACM SIGGRAPH 2008 classes*, SIGGRAPH '08, pages 133–164, New York, NY, USA. ACM.

[Kircher and Lawrance, 2009] Kircher, S. and Lawrance, A. (2009). Inferred lighting: fast dynamic lighting and shadows for opaque and translucent objects. In *Proceedings of the 2009 ACM SIGGRAPH Symposium on Video Games*, Sandbox '09, pages 39–45, New York, NY, USA. ACM.

[Kontkanen and Laine, 2005] Kontkanen, J. and Laine, S. (2005). Ambient occlusion fields. In *Proceedings of ACM SIGGRAPH 2005 Symposium on Interactive 3D Graphics and Games*, pages 41–48. ACM Press.

[Lottes, 2009] Lottes, T. (2009). Fxaa. Technical report, NVIDIA Developer SDK 11.

[Mammen, 1989] Mammen, A. (1989). Transparency and antialiasing algorithms implemented with the virtual pixel maps technique. *IEEE Comput. Graph. Appl.*, 9:43–55.

[McGuire, 2010] McGuire, M. (2010). Ambient occlusion volumes. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, pages 47–56, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association.

[McGuire et al., 2011] McGuire, M., Osman, B., Bukowski, M., and Hennessy, P. (2011). The alchemy screen-space ambient obscurance algorithm. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG '11, pages 25–32, New York, NY, USA. ACM.

[Mittring, 2007] Mittring, M. (2007). Finding next gen: Cryengine 2. In *ACM SIGGRAPH 2007 courses*, SIGGRAPH '07, pages 97–121, New York, NY, USA. ACM.

[Myers and Bavoil, 2007] Myers, K. and Bavoil, L. (2007). Stencil routed a-buffer. In *ACM SIGGRAPH 2007 sketches*, SIGGRAPH '07, New York, NY, USA. ACM.

[OpenGL, 2011] OpenGL (2011). `http://www.opengl.org/`.

[Phong, 1975] Phong, B. T. (1975). Illumination for computer generated pictures. *Commun. ACM*, 18:311–317.

[Reinhard et al., 2002] Reinhard, E., Stark, M., Shirley, P., and Ferwerda, J. (2002). Photographic tone reproduction for digital images. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '02, pages 267–276, New York, NY, USA. ACM.

[Reshetov, 2009] Reshetov, A. (2009). Morphological antialiasing. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 109–116, New York, NY, USA. ACM.

[Ritschel et al., 2009] Ritschel, T., Grosch, T., and Seidel, H.-P. (2009). Approximating dynamic global illumination in image space. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, I3D '09, pages 75–82, New York, NY, USA. ACM.

[Rost, 2005] Rost, R. J. (2005). *OpenGL(R) Shading Language (2nd Edition)*. Addison-Wesley Professional.

[Saito and Takahashi, 1990] Saito, T. and Takahashi, T. (1990). Comprehensible rendering of 3-D shapes. In *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '90, pages 197–206, New York, NY, USA. ACM.

[Shishkovtsov, 2005] Shishkovtsov, O. (2005). Deferred shading in S.T.A.L.K.E.R. In Pharr, M. and Radima, F., editors, *GPU Gems 2*, chapter 9, pages 143–166. Addison-Wesley Professional.

[Thibieroz and Gruen, 2010] Thibieroz, N. and Gruen, H. (2010). OIT and indirect illumination using DX11 linked lists. In *GDC San Francisco 2010*.

[Tomasi and Manduchi, 1998] Tomasi, C. and Manduchi, R. (1998). Bilateral filtering for gray and color images. In *Proceedings of the Sixth International Conference on Computer Vision*, ICCV '98, pages 839–, Washington, DC, USA. IEEE Computer Society.

[Valient, 2007] Valient, M. (2007). Deferred rendering in killzone 2. Online, accessed Feb. 20th, 2012. Develop Conference, http://www.guerrilla-games.com/publications/dr_kz2_rsx_dev07.pdf.

[Watt and Policarpo, 2005] Watt, A. and Policarpo, F. (2005). *Advanced Game Development with Programmable Graphics Hardware*. A. K. Peters, Ltd., Natick, MA, USA.

[Williams, 1978] Williams, L. (1978). Casting curved shadows on curved surfaces. *SIGGRAPH Comput. Graph.*, 12:270–274.