Karlsruhe Institute of Technology

Rendering Decals with Ray Tracing Acceleration Structures

Bachelorarbeit von

Sidney Hansen

An der Fakultät für Informatik
Institut für Visualisierung und Datenanalyse,
Lehrstuhl für Computergrafik

21. September 2020

Reviewer:	Prof. Dr.-Ing. Carsten Dachsbacher
Second reviewer:	Prof. Dr. Hartmut Prautzsch
Advisor:	Dr. rer. nat. Christoph Peters

Contents

1 Introduction **3**

2 Background **5**
2.1 Mip Mapping . . . 5
2.2 Order Independent Blending . . . 6
2.3 Frostbite BRDF . . . 7
2.3.1 Specular Component . . . 8
2.3.2 Diffuse Component . . . 9
2.4 GPU Ray Tracing . . . 9
2.4.1 Bounding Volume Hierarchy . . . 10
2.4.2 Ray Queries . . . 10
2.5 Forward Rendering . . . 10
2.5.1 Standard Pipeline . . . 10
2.5.2 Early z . . . 11
2.5.3 Depth Prepass . . . 11
2.6 Deferred Shading . . . 11
2.6.1 Geometry Pass . . . 12
2.6.2 Rendering Lights . . . 12
2.7 Decals . . . 13

3 Related Work **15**
3.1 Mesh Decals . . . 15
3.2 Deferred Decals . . . 15
3.3 Tiled and Clustered Shading . . . 16
3.3.1 Tiled Shading . . . 16
3.3.2 Clustered Shading . . . 17
3.3.2.1 Decals in Clustered Shading . . . 18
3.4 Deferred Texturing . . . 18
3.4.1 The Visibility Buffer . . . 18
3.4.2 Decals in Deferred Texturing . . . 19

4 Ray-Query Decals **20**
4.1 Loading and Sampling Decals . . . 20
4.1.1 Decal Loading . . . 21
4.1.2 Material Loading . . . 21
4.1.3 Sampling Decals . . . 22
4.2 Decal Enumeration with Ray Queries . . . 23
4.2.1 Acceleration Structure Construction . . . 23
4.2.2 Ray Query Loop . . . 24
4.3 Blending . . . 26
4.3.1 Per Channel OIT Blending . . . 27

5 Results **29**
5.1 Implementation 29
5.1.1 Forward Shading with Depth Prepass 30
5.1.2 Deferred Shading 30
5.1.2.1 Deferred Decals 31
5.1.3 Deferred Texturing with Visibility Buffer 31
5.2 Evaluation 32
5.2.1 System Specifications 32
5.2.2 Few Blended Decals 33
5.2.3 Many Decals with many Lights 35
5.2.4 Very many Decals 37
5.2.5 Very many Lights 38

6 Conclusion **40**
6.1 Future Work 41

7 Acknowledgements **42**

Bibliography **43**

Zusammenfassung

In dieser Arbeit beschreiben wir eine neue Technik zum Rendern von Decals auf Ray-Tracing Hardware. Das Ziel ist es eine flexible Technik zu entwickeln, die sich unabhängig von der Rendertechnik realisieren lässt. Hierfür entwickeln wir einen Renderer in welchen wir unsere Technik implementieren. Dieser Renderer unterstützt Forward Shading, Deferred Shading und Deferred Texturing.

In der ersten Hälfte unserer Arbeit betrachten wir bisherige Ansätze zum Rendern von Decals. Die betrachteten Ansätze sind Mesh Decals, Deferred Decals und Decals in Clustered Shading. Wir betrachten diese Ansätze jeweils mit Blick auf die unterschiedlichen Shading Techniken und stellen fest, dass Decal implementierungen meist von der verwendeten Shading Technik abhängen. Im Hauptteil beschreiben wir unsere Technik, welche wir Ray-Query Decals nennen. Der Kern dieser Technik ist der Gebrauch von Ray-Queries zum enumerieren aller überlappender Decals während der Fragment Shader Ausführung. Hierbei speichern wir die AABBs aller Decals in einer Beschleunigungsstruktur. Während der Fragment Shader Ausführung schießen wir einen Strahl der Länge 0 um alle überlappende AABBs zu finden. Zur Evaluierung von Ray-Query Decals mit den implementierten Rendertechniken benutzen wir unterschiedliche Testszenarien.

Zu Vergleichszwecken implementieren wir Deferred Decals für Deferred Rendering.

Aus den Testergebnissen schlussfolgern wir, dass Ray-Query Decals für alle Testszenarien unter normalen Bedingungen anwendbar sind. Deferred Decals sind dennoch etwas schneller. Weiterhin stellen wir fest, dass wir selbst für eine sehr hohe Anzahl an Ray-Query Decals eine gute Performanz in Deferred Shading und Deferred Texturing erhalten.

Als Anwendungsgebiet für Ray-Query Decals sehen wir vor allem Deferred Texturing, da wir hier keine Deferred Decals benutzen können und somit eine Alternative brauchen. Am Ende unserer Arbeit befassen wir uns zudem mit dem Rendern von lokalen Lichtquellen, basierend auf der selben Technik welche wir auch für Ray-Query Decals benutzen.

Abstract

In this thesis we describe a novel technique for rendering decals on ray tracing hardware. The intent is to provide a flexible solution that can be used independent of the underlying render technique. We therefore create an application, where we implement and evaluate this technique for forward shading, deferred shading and deferred texturing.

In the first half we discuss previous approaches for rendering decals, such as mesh decals, deferred decals and decals in clustered shading. We view them in the context of different rendering methods and see that decals often rely on the underlying render technique. In the main part, we describe our method, which we call ray query decals. The key point in our method is to use ray queries in the fragment shader to enumerate all overlapping decals. To accomplish this, we store the AABBs of all decals into an acceleration structure. We then shoot a ray of length zero to obtain all overlapping AABBs for a fragment. To evaluate our technique with the implemented renderers we use different test scenarios. For the evaluation of ray query decals with deferred shading, we also implement deferred decals as a comparison.

From the results we conclude that ray query decals are applicable for all evaluated render techniques under normal conditions. Deferred shading performs slightly better with deferred decals. Also our technique remains stable for a remarkably high number of decals with deferred shading and deferred texturing.

As an application area for ray query decals we primarily see deferred texturing. Here we cannot use deferred decals and therefore rely on an alternative technique to render decals. In the end we also discuss using the same technique to render local lights.

1. Introduction

While the move towards real-time ray tracing has been kicked of by NVIDIA's Turing technology, it is now being backed up by Intel's Xe GPU, AMD's Big Navi and NVIDIA's second generation RTX cards. With the release of the RTX 30 series this month and the upcoming releases of Intel's and AMDs GPUs, accelerated ray tracing is well on its way of becoming mainstream for high end systems. The graphics API's Vulkan and DirectX12 already offer a tight integration of ray tracing functionality with the existing standard rasterization pipeline. While the primary function of ray tracing hardware is computing light transportation, the underlying hardware accelerated BVH traversal can also be used for other purposes. For graphics programmers this allows for many new possibilities.

In this thesis, we make use of ray tracing functionality to efficiently render decals. Decals are design elements used by most renderers to add details to surfaces (see Section 2.7). By projecting a texture in a limited volume, we modify the surface properties of the contained geometry. This may also be used for dynamic elements such as player tags, bullet holes or footprints. Figure 1.1 shows a decal applying to an uneven surface. Our focus lies here on providing an implementation of decals with good performance and high flexibility, that is visually equal to competing techniques.

To cover the basics, we first analyze existing methods for rendering decals. We discuss mesh decals, deferred decals and decals with clustered shading(Section 3.1, 3.2, 3.3.2.1). The most common method for rendering decals is deferred decals. While deferred decals are very efficient, we discover that they have some limitations. The main limitation is that deferred decals are only applicable in deferred shading. Though deferred shading is still used in most game engines, there has been a shift towards less bandwidth heavy render techniques, such as deferred texturing. Therefore, we see the need for a more flexible technique for rendering decals with different renderers.

In this context we propose ray query decals, a technique that uses the ray tracing acceleration structure to render decals efficiently. For that we form the decals AABB and store it to an acceleration structure (Section 4.2.1). To iterate over decals during shading we traverse the acceleration structure with ray queries (see Section 4.2.2). We here shoot a ray of length 0 to enumerate all decals which contain the current fragment in their AABB.

While we are limited to GPUs with ray tracing capabilities, our technique is easily integrated within different renderers. To prove this, we successfully implement ray query decals with 3 different render techniques. Of these render techniques we cover the fundamentals in the sections 2.5, 2.6 and 3.4. During the implementing of ray query decals,

we treat different topics such as: mip mapping, order independent transparency and the BRDF. For these topics we cover the basics in the sections 2.1, 2.2 and 2.3. In Chapter 5 we evaluate ray query decals with 3 test scenarios. In these scenarios we compare our technique to deferred decals under different conditions. In the process we consider the visual correctness and assess the performance for each renderer.

We find that ray query decals archive similar visual results to deferred decals, while being slightly slower. However ray query decals are more flexible and broadly applicable then deferred decals. At the end of this thesis we employ the same method for rendering local lights (Section 5.2.3), which we discuss in future work 6.1.

Figure 1.1: Decal applied to an uneven surface.

2. Background

In the scope of this work, we explore previous methods for rendering decals. We also discuss different render techniques to understand why some decal implementations only work for certain renderers.

All decal techniques which we discuss in this work require accessing textures. Therefore, we cover mip mapping in the first section (Section 2.1). Since ray query decals cannot rely on alpha blending, we discuss methods for order-independent blending in the following section (Section 2.2). Next, we explain the physically based material definition we use for all implemented renderers (Section 2.3). In Section 2.4 we discussed ray tracing functionality that we use for ray query decals. In the sections 2.5 and 2.6 we cover forward and deferred rendering. Deferred texturing is discussed later in Section 3.4 At the end of this chapter we offer a more detailed descriptions of decals and their use in modern renderers (Section 2.7).

2.1 Mip Mapping

Mip mapping is a popular technique used to reduce aliasing when sampling from textures. When a texture gets minified, (e.g. a textured object moving away from the camera) this causes several texels to be mapped to one pixel. Sampling only one texel would introduce aliasing artifacts such as jittering and Moiré patterns. To avoid this, we must correctly calculate the contribution of every texel to the pixels color. Directly iterating over such an amount of texels requires too much bandwidth and processing time. Hence it is not applicable in real time rendering.

As a solution to this, we use mip mapping [1]. This technique works by precalculating several filtered versions of each texture. These textures are obtained by repeatedly down-scaling the original texture by the factor of 2 in width and height, then storing the result. An easy way to go about this, is to apply a box filter to every 2x2 set of pixels. Other low pass filters, such as the Gaussian filter are better at eliminating high frequencies. Tho, they also come at a higher computational cost. The mip map hierarchy can be processed in advance, using the GPU or the CPU.

The output is a pyramidal data structure where each level holds a filtered version of the texture, half the size of the level below. The lowest level (level 0) holds the original texture, while the highest level only contains one texel, holding the average color. This is illustrated in Figure 2.1

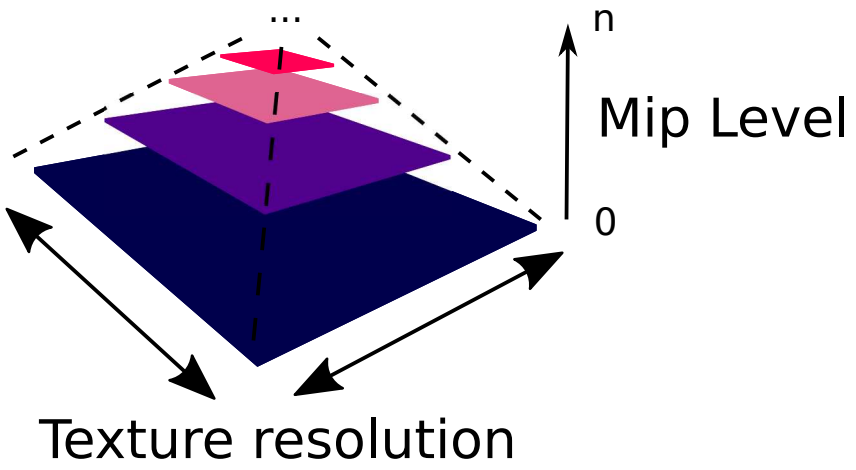


Figure 2.1: *Illustration of the mip map pyramid.*

To sample from a mip map we use three coordinates u, v and d. d is the level of detail. We use it to determine the next larger and lesser mip level. We sample these with bilinear interpolation, using the u and v coordinate. Then we use d again to interpolate those values.

The value of d is calculated by taking the derivative of the uv-coordinates in screen space. The numerical values of $\frac{du}{dx}$, $\frac{dv}{dx}$ and $\frac{du}{dy}$, $\frac{dv}{dy}$ can be approximated by comparing to neighbor fragments. Modern hardware uses single instruction multiple data (SIMD) to process fragments. When a triangle is rasterized, fragments are always generated in sets of 2x2. If the texture fetch is always performed for the entire set, the uv-coordinates of these fragments can be compared to obtain the derivatives [2]. We later explore a case where this scheme does not work (see Section 4.1.3).

Overall, mip mapping is cache friendly as low-resolution textures are more easily cached. Thats wy it significantly increases performance on bandwidth limited systems. For our application we only support isotropic mip maping, which is described in this section. Anisotropic mip mapping allows to individually filter textures along both axes. This helps to create a sharper image when viewing a surface from a flat angle.

2.2 Order Independent Blending

When rendering opaque objects, we use depth testing to handle occlusion per fragment. This works fine if only one fragment contributes to the final color of the pixel. Rendering translucent objects is more complicated and requires blending of multiple layers.

The common approach is to sort translucent objects back to front and render them after all opaque geometry. Thus, he fragments color is blended with the current color in the framebuffer. Blending with the framebuffer happens during the merge stage (see Section 2.5). This process can be controlled by using a fixed set of blending operations. The blending operation for regular alpha blending is given by Porter and Duffs OVER operator[3]:

$$C_f = C_1 + (1 - \alpha_1)C_0$$

where C_1 is the premultiplied surface color 1, covering the backround 0. The premultiplied color is obtained by scaling the rgb values with the alpha value of the color. The final color results in:

$$C_f = [C_n + (1 - \alpha_n)...[C_2 + (1 - \alpha_2)[C_1 + (1 - \alpha_1)C_0]]]$$

While alpha blending delivers correct results in most settings, it is not applicable to every use case. Order independent transparency (OIT) is a set of techniques that do not require the blending operations to be performed in the correct order. We only discuss weighted blended order independent transparency, as this is well suited for blending decals [4].

When blending multiple layers with alpha blending, lower layers are partially occluded by the upper layers. Weighted blending modulates partial occlusion by employing weights. These weights control how much a blended surface influences the final color. A low weight for example suggests that the surface is situated behind other surfaces. Blending takes place by forming the weighted sum over all color values:

$$C_f = \frac{\sum_{i=1}^{n} C_i \cdot w(z_i)}{\sum_{i=1}^{n} \alpha_i \cdot w(z_i)}(1 - \prod_{i=1}^{n}(1 - \alpha_i)) + C_0 \prod_{i=1}^{n}(1 - \alpha_i)$$

Again we use premultiplied colors C_i and weight them with $w(z_i)$. $w(z_i)$ calculates the weight on the depth z_i. For the correct outcome it is important to normalize the result. This we do by dividing with the term $\sum_{i=1}^{n} \alpha_i \cdot w(z_i)$. One last thing to consider is the total coverage of the background. This can be calculate using the product $\prod_{i=1}^{n}(1 - \alpha_i)$.

For calculating the weight function there exists different approaches. Weights are normally calculated based on the depth value of the fragment. For this we must define a function which maps the depth value to a blending weight. Depending on the scene different functions yield better results. McGuire and Bavoil present different polynomial functions for different scenes [4]. They also use the alpha value as a parameter to linearly decrease the weight based on transparency.

Other approaches calculate the weight function dynamically, based on the current frame. Moment-based order-independent transparency, works by using a prepass to calculate moments, based on the depth of the fragment [5]. These are then used in a second pass to construct the weight function per pixel.

In our case we use static weights, as our blending order is not based on depth values.

2.3 Frostbite BRDF

For representing materials under various lighting conditions we adopt the standard material model used by the Frostbite engine [6]. The model employs a bidirectional reflective distribution function (BRDF) used for physically based rendering (PBR) to approximate light reflection on dielectric and metallic surfaces. Further on, this model is limited to isotropic surfaces. Given this material model different lighting techniques may be employed, such as point lights, area lights and image-based lighting. To simplify lighting, we only use point light sources.

The general reflectance equation is given by [7]:

$$L_o(v) = \int_{l \in \Omega} f(l, v) L_i(l)(n \cdot l) dl$$

Integration is performed over the unit hemisphere. This covers the contribution of light over all incident angles. Backlit surfaces are therefore ignored. As we only allow point lights, we can substitute the integral by using a sum over all light sources. $L_i(l)$ is the intensity of the light source. We scale this value with $n \cdot l$, which represents the falloff in intensity when light hits the surface from an angle. Often this term is expressed as $\cos(\theta)$, where θ is the angle of incidence.

What remains to be calculated is the BRDF f. The BRDF represents the property in which the surface material reflects light. f is calculated as the sum of the specular component

f_r and the diffuse component f_d. To be properly physically based, f must also follow the laws of energy conservation. These implicate that the total reflect radiance over all angles cannot surpass the incoming radiance.

2.3.1 Specular Component

Most PBR materials use a specular component based on a microfacet model [8]. The specular BRDF is described by the equation [9, 7]:

$$f_{spec}(l, v) = \frac{F(h, l)G(l, v, h)D(h)}{4|n \cdot l||n \cdot v|}$$

Here G and D modulate the microfacet structure, while F is the Fresnel term used for reflection in optics. The vectors used are:

- n: Surface normal,
- l: Outgoing light direction,
- v: Outgoing view direction,
- h: Halfway vector.

The microfacet model treats a rough surface as a large set of tiny flat surfaces (facets). These facets are represented by the orientation of their normal vectors. Every facet acts as a perfect mirror. This causes it to reflect light in the direction of the reflected light vector. The surface structure is illustrated in Figure 2.2. For a given a light vector v and a viewing direction l, only facets aligned to the half vector h contribute to the specular brightness. The half vector is introduced as:

$$h = \frac{v + l}{||v + l||}$$

.

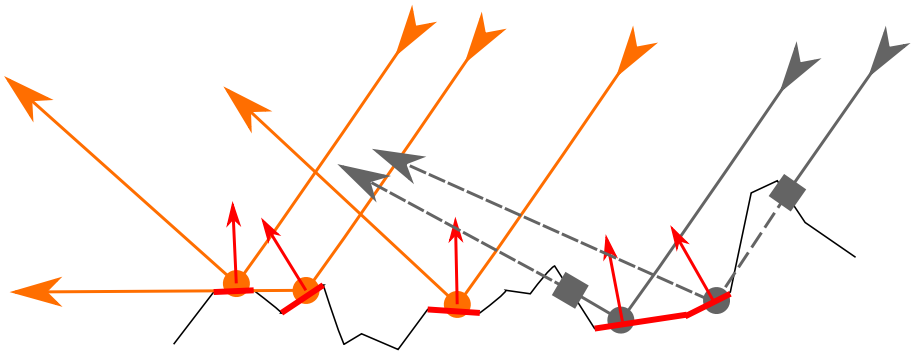


Figure 2.2: *Microfacet surface. (Red) Facets and facet normals. (Orange) Reflected ligth rays. (Grey) Blocked Ligth rays for masked and shadowed facets.*

To evaluate the reflected brightness of a surface area, we need to calculate the proportion of facets, of which the normal is aligned to the halfway vector. We do this by using a normal distribution function (NDF) $D(h)$. The roughness value of the surface material is used to calculate the distribution of the microfacet normals. For example, a roughness value of 0 means all facets are perfectly aligned to the surface normal. The form of the curve described by the NDF, visually affects the size and intensity of the specular highlight.
$G(l, v, h)$ is called the geometry function or masking-shadowing function. It is used to

calculate the proportion of facets that are actually visible from view direction as well as light direction (see Figure 2.2). This is used together with the NDF to obtain the coefficient representing the fraction of facets that actually reflects light from the light source to the viewer. Frostbite uses the hight correlated version of the Smith masking-shadowing function, recomended by Heitz [10, 11, 6, 7]. The implemented function also contains the denominator $4|n \cdot l||n \cdot v|$.
$F(h, l)$ is the Fresnel reflectance. It is used in optics and represents the amount of light that gets reflected from a surface under a certain angle. In real world physics the Fresnel term also takes into account the wavelength of the light. In real time computer graphics this influence is generally neglected. When approaching an angle of 90 degrees F always converges towards 1. F remains rather linear for small angles and takes the shape of an exponential function when approaching 90 degrees. For an angle of 0 degrees we specify a constant value. This value ranges from 0.02 to 0.05 for most dielectric materials and is between 0.5 and 1 for most metals. [7] For simplification we use a constant value of 0.02 for all non metals. While metals have a high specular reflection coefficient they do not have a diffuse component. To simulate this effect, we interpolate the Fresnel value at 0 degrees with the base color, by using the metallicity of the material. Then we scale down the diffuse color to ensure energy conservation. The Fresnel term we use the Schlick approximations [12].

2.3.2 Diffuse Component

For the diffuse component we use a renormalized version of Disney diffuse term [13].

The diffuse term describes the amount of light that is scatterd by the surface. In this case the model is based on empirical observation. The equation presented by Burley is given by the equation:

$$f_d = \frac{baseColor}{\pi}(1 + (F_{D90} - 1)(1 - n \cdot l)^5)(1 + (F_{D90} - 1)(1 - n \cdot l)^5)$$

with $F_{D90} = 0.5 + \cos(\theta_d)^2\alpha$
As this model is empirical, we do not discuss it any further. The original paper is listed in the bibliography [13].

2.4 GPU Ray Tracing

Ray tracing relies on intersecting each ray with the geometry in the scene. This process can be accelerated using bounding volume hierarchies. As a result, the asymptotical complexity of intersection testing is reduced down to $log\ n$ for n primitives. Also, each ray may be processed individually leading to a large speedup on parallel architectures.

While ray tracing scales well with the amount of geometry in the scene, it poses challenges for the underlying hardware. The recursive nature of ray tracing, as well as the difficulty of managing incoherent memory access, have prevented it from being viable in real time rendering systems until recently.

With NVIDIA's release of the RTX Graphics Cards, ray tracing is again becoming relevant in the cosmos of real time graphics programming. With the Turing chip generation, NVIDIA has released the first consumer GPU that offers hardware capabilities to support efficient ray tracing. These are accessible to programmers through APIs such as Vulkan and DirectX12. The tight integration of ray tracing features within the current standard, allows hybrid renderers to combine rasterization with ray tracing techniques.

2.4.1 Bounding Volume Hierarchy

The bounding volume hierarchy (BVH) is a spatial tree structure. Every node stores a bounding volume, such as a bounding sphere or an axis aligned bounding box (AABB). Leave nodes, hold geometrical primitives. The bounding volume of each node must contain all geometry of its subtree. Hence, a ray that does not intersect the bounding volume of a node, does not intersect any primitives in its subtree.

This allows us to calculate ray-primitive intersection, by performing a depth-first traversal of the BVH. The required computations scale with different factors, such as the dimensionality and the depth of the tree. The time required to construct and update the BVH for dynamic scenes are also important metrics to be considered. We do not discuss BVH construction here in detail as there exist many different strategies. In our case we use the native implementation exposed by the Vulkan API to create the acceleration structure, which in its implementation use a BVH [14].

2.4.2 Ray Queries

Ray tracing in Vulkan can be implemented through the ray tracing pipeline, or through ray queries. We focus on the latter, as it offers us more flexibility and can be well integrated within existing rasterization pipelines. Ray queries allow us to perform ray intersections in any shader stage. They are initialized via shader commands, and require an acceleration structure to query against. Further on we use flags and a culling mask to control how and which geometry is intersected. This enables us to make complex decisions base on ray intersections. Ray queries are used in a loop, in which we iteratively obtain all potential hits. What happens during this loop is determined by the programmer. An example for the structure of the ray query loop is shown by the following pseudo-code

```
1   initialise ray query (...)
2   while(Traverse acceleration structure (ray query))
3       // returns true if a potential hit is found
4       // otherwise loop terminates
5       get hit data (hit)
6       process ( hit data )
7       if( some condition ) {confirm intersection}
8       // to be used for closest hit determination
9   }
10  handle result()
11  // for example fetch triangle data for the closest hit
```

Listing 2.1: Ray query loop in pseudo-code

2.5 Forward Rendering

Unmodified forward rendering uses the standard GPU pipeline for rendering. Geometry processing and shading are here done in the same render pass.

2.5.1 Standard Pipeline

Here we discuss the standard graphics pipeline that is used in unmodified forward rendering.
Rendering a frame on the GPU is carried out through several stages. Here we only discuss stages that are directly relevant for our implementation:

- **Vertex Shader:** The Vertex shader is executed for every vertex in our scene. Here we transform our geometry to clip space and extract vertex data used for shading individual fragments. Here we also transform vertex normals, tangents and bitangent to world space. These are used for interpreting normal maps during shading.

- **Rasterization:** Our triangles are rasterized into fragments, holding the interpolated vertex data. Fragments are generated and processed in 2x2 sets forming a quad. These are then evaluated by the fragment shader.
- **Fragment Shader:** Here we shade the fragment for every light source, respecting its material properties. Shading data is obtained from interpolated vertex attributes and texture fetches. We use implicit screen derivatives to access mip maps. Normals are transformed using the tangent frame, described by the vertex attributes. If we do not use a depth prepass, we must perform an alpha test to render cutout materials.
- **Merge Stage:** Before writing to the framebuffer, a depth test is performed. Occluded fragments are discarded. Others write their depth value to the depth buffer. Then we write the final output color to the framebuffer during the merge stage. If a blending option is selected we instead blend the color with the current color in the framebuffer.

The full time it takes to render a frame is determined by several factors. Very often a bottleneck is caused by operations during fragment shading. Here the bottleneck may be caused by factors such as register pressure, complex shading operations or bandwidth limitations. While the fragment shader can be optimized in itself, reducing the overall number of fragment shader invocations, definitely increases the performance.

2.5.2 Early z

Given a complex scene, many fragments are fully shaded and then discarded later on, due to depth testing. If the fragment shader does not discard any fragments or change the depth value, the depth test may be performed before the fragment shader execution. This reduces the number of fragments that need to be shaded. This is also known as early z and is an optimization performed automatically by most modern hardware.

2.5.3 Depth Prepass

The depth test is processed in triangle submission order. This means, unless the geometry is ordered roughly front to back, there will be a lot of overdraw. An effective way to reduce overdraw is to perform an early depth pass. In a first render pass all opaque geometry is rendered without shading. Only the depth value is stored. If we do not use an alpha test the fragment shader is not even required. In a second pass the scene is rendered another time. Before the fragment shader invocation, a depth test (early z) is performed against the final depth buffer from the first pass. If we want to render cutout material we must select the equal operation for this test. The depth buffer here does not need to be written, leading to further optimisations. Now, only fragments are shaded that are not discarded by the depth test.

Still, some fragments are unnecessarily shaded. As we already know, fragment shading takes place in 2x2 tiles. This causes overdraw to happen along the edges of triangles, were background tiles and foreground tiles of different triangles overlap. While the depth test significantly reduces the number of shaded fragments, it also comes with an additional cost, as the entire geometry is now processed 2 times. Other render techniques such as deferred shading solve this problem in a different way.

2.6 Deferred Shading

Deferred shading was introduced in 1990 by Saito and Takahashi and has become the most widespread render technique [15]. It is employed in most game engines such as the Frostbite engine and is also partially integrated into hybrid renderers as in Doom (2016)

[6, 16]. Deferred shading is based on using a temporary buffer to store shading data per pixel between render passes. It is primarily used for its efficient way to deal with many local lights. It is also often considered for its powerful way in dealing with decals. Decals are discussed in a later chapter. The basic concept of deferred shading is to separate geometry and visibility calculations from shading.

2.6.1 Geometry Pass

Deferred shading uses an initial render pass to rasterize all opaque geometry. This render pass is also known as the geometry pass. The fragment shader obtains the interpolated vertex data, and fetches textures to gather data for shading. Unlike in forward rendering, the fragment shader does not contain code for lighting calculations. Instead, the shading data is written to a set of render targets after passing the depth test. These render targets form the geometry buffer, called the g-buffer [15].The type of data that is stored in the g-buffer, depends on the material model. In most cases this includes the diffuse albedo and the normal. How the data is stored is an important point to consider. The bandwidth used by accessing the g-buffer scales linearly with its size. An oversized g-buffer will decrease the applications performance on bandwidth limited systems. By storing data in limited precision and applying compression, the size of the g-buffer can be reduced, to save bandwidth [7]. On this subject there exists extensive literature, as it is vital to most modern-day renderers.

While overdraw does occur during the geometry pass, it does not stress the performance very much. In this pass the fragment shader does not need to perform any heavy calculations. In return we have completely eliminated overdraw for the shading pass. This is chiefly beneficial when lighting calculations get complex.

2.6.2 Rendering Lights

To render lights there are different strategies. One is to render a full screen quad and sample the g-buffer per pixel. The obtained information is then used to properly shade each pixel, iterating over all light sources. Instead we may also render a full screen quad for every light, trading bandwidth usage for shader complexity. While these methods may be reasonable for a small amount of lights, especially if they affect large parts of the scene, there exist more elaborated lighting schemes.

When rendering scenes with multiple local light sources, not every light source significantly affects the entire scene. Only fragments close to that source of light actually need to be evaluated. To do this, we generate meshes for our light sources, approximating the area of influence. For example we use rough sphere shapes for point lights [7]. This geometry is rasterized during the shading pass. For each generated fragment we use viewport coordinates to sample the g-buffer and depth buffer. Then we shade the pixel using the retrieved data. Additionally, we can use the depth value of the depth buffer to determine if the fragment is contained inside the sphere, and thus needs to be shaded. The result is blended with the current color in the framebuffer. Later on, we take a similar approach to render deferred decals.

While deferred shading lets us render many local lights, it also comes at its costs. As already discussed, using a large sized g-buffer will strain our bandwidth capacity. As modern systems are mostly bandwidth limited, this may cause performance loss. This concern has led developers to explore new strategies, such as deferred texturing (Section 3.4). Also, selecting the right g-buffer layout must be considered closely when creating a material system. Preserving a small g-buffer may constrain what kind of materials can be rendered.

Besides that, there is the obvious limitation to opaque and cutout materials. To render semi-transparent objects we need to employ a separate forward render pass. Lastly deferred shading does not support anti-aliasing schemes such as MSAA. Temporal aliasing is often used as an alternative.

2.7 Decals

Figure 2.3: Here we use decals to place puddles on the street of the Amazon Lumberyard Bistro [17]. The affected surface areas are marked in red.

A Decal is a common design element used in computer graphics. In simple terms it can be thought of as a texture projected onto a surface. This is illustrated in Figure 2.3. It is used to create different visual effects and add fine grained details to objects. Id tech 7 demonstrates the extent of what decal systems are capable of in their recent title 'Doom Eternal' [18].Here we see intricate objects all around the map, with numerous details added through the extensive use of decals. The influence of a decal is not limited to the color but allows us to modify plenty of surface parameters (see Figure 2.4). For instance, dents and protrusions may be simulated through modification of the surface normal.

Another use for decals is to apply dynamic modifications to the surface. We can use this to create effects such as bullet holes, footprints or tire marks. Most implementations allow us to dynamically add decals while preserving scene geometry and textures in graphics memory.

Layering decals on textured objects also allows us to create different unique surface appearances, while reusing the same base texture over many objects. This helps us to reduce the memory footprint, especially when using high resolution textures. A lower count of base textures also reduces the overall disk space required for our application.

A Decal may modify surface parameters according to the material system. For our implementation of the Frostbite standard model this includes: diffuse albedo, surface normal, roughness, metallicity. A decal may apply to any subset of these parameters, as shown in figure 4.5.

These values may be overwritten or blended using the decals stored alpha values. As we see decals offer a lot of flexibility and are definitely worth considering when developing a rendering engine. In order to be viable, we need an efficient way to render decals. In most modern renderers we see variations of two widespread approaches. Deferred decals is a technique used in deferred renderers. It allows us to layer many decals while only having to compute lighting calculations once.

The other technique is sometimes called geometry or mesh decal. It relies on creating a textured mesh close over the surface geometry. It is easily integrated within most rendering pipelines, but also comes with several drawbacks.

We discuss both techniques in the following chapter. There also exist alternative approaches to render decals, that we do not look closer at. Most of these suffer from limitations and restrictions to the engine, or do not perform well. For instance, renderers that use a megatexture, such as Doom 2016 may directly edit the megatexture in graphics memory to apply modifications to surfaces [19]. Another option that has been explored is rendering an object for every decal on the surface and blending the result [7].

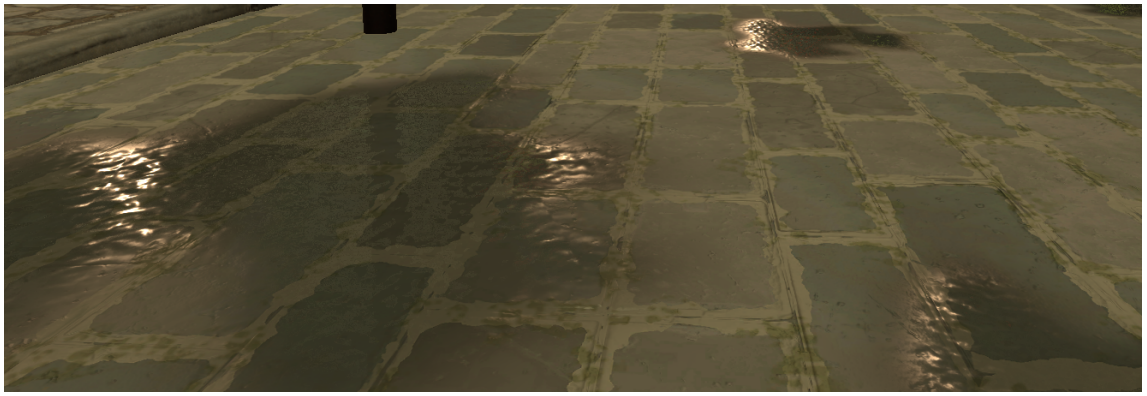


Figure 2.4: Decals may apply to the surface in different ways, completely changeing the surface appearance. In this example we use ray traced reflections.

3. Related Work

3.1 Mesh Decals

Mesh decals offer a cheap way to render decals that can be well integrated within existing render pipelines, without increasing shader complexity. Here a decal is rendered as a mesh placed closely over the background geometry.

This mesh is generated in advance and can be placed to cover the underlying surfaces. As with other techniques we define the decal as an oriented box. This box can be thought of as an orthographic projector. One side of the box is mapped to a set of textures. We then create a cutout version of the geometry, on which we project our decal. This can be done by finding all triangles intersecting the box. Then we crop triangles that are not fully contained in the box. The result is a mesh that fits the underlying surface geometry. This mesh is then textured, by projecting the textures of the decal orthographically to the surface [20].

The cutout geometry also may be further processed. For example, we may apply tessellation to add more vertices. This offers us a convenient way to apply local displacement maps, which require a high vertex density of the underlying surface. If using non-binary alpha values the decal needs to be handled like any other kind of semi transparent geometry. As already discussed semi-transparent objects are rendered after opaque geometry, and require back to front ordering assumed that no OIT technique is used.

While this implementation is relatively easy to be handled by most renderers, it has several draw backs and limitations. For instance, layering many decals will cause a lot of overdraw, as every decal is shaded independent of the underlying surface. Also, the z-bias used to render the decal in front of the underlying surface can be a source of z-fighting artifacts.

3.2 Deferred Decals

A popular way to render decals with deferred shading are deferred decals [21]. As opposed to mesh decals, the orthographic texture projection takes place in the fragment shader. Rendering deferred decals works similarly to how we render lights. As deferred decals apply to the g-buffer, we may use them to modify any parameters stored within. That provides limited freedom on how decals may influence the surface material, based on the information stored in the g-buffer.

As with Mesh decals, we represent decals with oriented boxes, simulating orthographic projectors. One side of the box is mapped to a set of textures containing the surface parameters, that the decal applies to. These boxes also represent the bounding volume, in which fragments are affected by the decal. In the fragment shader, textures are orthographically projected down onto the surface.

Deferred decals are rendered in a separate pass, between the geometry pass and the shading pass. In order to drive the fragment shader invocation, we rasterize the decals boxes. This works the same way as we have done with light volumes. For the generated fragments we then fetch the depth buffer to determine the location of the underlying surface. Next, we compute the fragments relative position in the box through multiplication with the inverse basis matrix of the box (see figure 4.1). The obtained vector then can be used to determine whether the fragment is located in the box. Only fragment inside the box are further processed. If the fragment is not rejected (false positive), we use the xy-components to sample the decals texture at the correct location. When rendering decals this way we must consider the case where the camera is located in the box of the decal. In our implementation of a deferred decal system, we detect this case in the vertex shader. There, the box is transformed into a screen filling triangle. Thus, every pixel is tested for this decal. For blending decals we use alpha blending. The blending order can be defined by initially sorting all decals back to front.

While blending the decal data directly to the g-buffer comes with many advantages, it also imposes some restrictions. As the g-buffer forcefully only stores opaque geometry, we cannot apply decals to transparent surfaces this way. Also, we are limited to blending only the same material types, as the g-buffer can generally only hold parameters for one material type at a time.

3.3 Tiled and Clustered Shading

Tiled and clustered shading are techniques that can be applied to each, forward and deferred rendering pipelines. They aim to reduce the performance overhead, created through the use of many local lights. To do this we approximate the bounding volume of each local light source. The goal is to only evaluate fragments that are contained within this bounding volume. The same approach may also be applied to decals, as their influence area is contained within the box defining the decal. There already exist some renderers that use this technique to apply decals.

3.3.1 Tiled Shading

Both, tiled shading and clustered shading work by forming light lists. These are evaluated in the fragment shader, to shade individual pixels.

The general concept was first introduced by Zioma[22]. He proposed rendering proxy meshes that encompassed the influence area of local light sources. The data required for shading is then stored for overlapping pixels. Multiple overlaps are handled through depth peeling.[7] Another approach suggests only storing the index in a light buffer for each pixel. This requires performing a depth prepass for the geometry [23].These indices are compressed to the RGBA channels of a texture and read during shading. Thus, this approach is also limited by the amount of lights that can affect one fragment.

Tiled shading is based on the assumption, that local regions of pixels are mostly affected by the same set of light sources. This technique was first employed in the game Uncharted: Drake's Fortune [24]

A tile is a square of pixels on the screen. For example, this may encompass 32x32 pixels. For each tile we store a list of lights affecting pixels in this square. During the shading

pass, each fragment is shaded for every light source contained in the light list of its tile. To construct the light list, we intersect the bounding volumes of our light sources with the frustums of all tiles. For point lights this can be calculated efficiently on the CPU or in the compute shader. [7] As the frustums tends to be long and thin we will often get many light intersections along the z-axis. Many of these are too far away to affect any fragment rendered in that tile. This causes false positives, decreasing our overall efficiency. In forward renderers with depth prepass, and in deferred renderers, we can use the depth value to further cull lights. For this we calculate the depth range for each tile by storing the smallest and largest fragment depth value as z_{min} and z_{max}. Lights that only intersect the frustum outside of this range do not affect any fragment and can be culled. This approach offers good results for tiles with a small depth range, but performs bad when large parts of the screen display borders between foreground and background geometry. This problem can be solved by categorizing lights by their depth. Then, for each fragment, only lighting computations are evaluated for lights that fall in the depth category of the fragment. An easy way to do this is to simply split the view frustum into a near and a far part. This is often called HalfZ. It can also be applied multiple times, creating an arbitrary amount of depth splits [25]. A more advanced solution is 2.5D culling [26]. Here, we split the depth range in to a fixed number of cells, represented by bits in a bitmask. By ANDing the geometrys bitmask with the light bitmask, we determine if the light can affect the geometry.

3.3.2 Clustered Shading

Clustered shading works similarly to tiled shading, as it creates lists of local lights. Instead of classifying lights by screen regions, we subdivided the entire view frustum into 3D cells. For each of these cells we test all light sources for intersection. The dimensions of this subdivision may variate based on the implementation. For instance, the Unreal Engine uses cells that span a screen region of 64x64 pixels while using 32 depths slices.[7] The light lists for these clusters can be computed independent of the geometry. Some implementations even suggest using a world space grid to compute clusters.

As with tiled shading, we require an efficient way to construct such a light list per cluster. Here there exist multiple CPU and GPU driven solutions. Ollson et. al suggests a fully hierarchical approach, using a BVH [27]. First, lights are grouped by their position using Morton codes [28]. The BVH tree is then constructed featuring 32 children for each inner node. Each node holds an axis aligned bounding box (AABB) containing all lights in its subtree. For each light, a bounding sphere is stored in its leaf node. The resulting shallow BVH can then be efficiently traversed on the GPU.

Pearson uses an iterative strategy to obtain all frustums affected by a light source [29]. He successively narrows down the set of affected clusters for each light source. If clustering is performed on the GPU, geometry information may be used to further cull lights per cluster. Ollson et. al uses the g-buffer normals present in deferred shading to cull lights based on normal cones [27].

One advantage of clustered shading over tiled shading is the increased stability in performance. This is mainly due to the reason that tiles with large depth discontinuities are treated more efficiently.

Doom 2016 uses a variation of clustered forward shading, employing techniques discussed by Ollson and Pearson [16] . In this case the view frustum is clustered into a low-resolution grid of 16x8 with 24 depth slices. Early out tests are then performed during shading to further cull false positives. The Depth slices are calculated in view space, exponentially increasing in spacing.

3.3.2.1 Decals in Clustered Shading

As already seen in deferred shading, we can use the same technique we use to render lights, to also render decals. Id tech uses clustered shading to efficiently render decals in DOOM 2016 [16]. Next to lights, decals and environement probes are clustered in the same grid. Every item is represented by its oriented bounding box or frustum. Then we iterate over every grid cell in clip space (which are effectively AABBs) to obtain all intersecting items. These are stored per cluster using their index. During a forward shading pass we iterate over all items, performing early-out fragment checks to further reduce their number. Decals are here iterated and blended in a loop, similar to how we handle ray query decals. Petino uses a similar approach [30].

3.4 Deferred Texturing

Saving bandwidth has become more important in recent years. While bandwidth and computing power are both increasing, the latter is rising faster. This shifts the bottleneck towards bandwidth limitation [7]. Especially deferred shading suffers from this limitation, as repeated g-buffer access consumes a lot of bandwidth. To address this behavior, modifications to deferred shading have been developed, that work around bandwidth limitations [30, 31]. Here, we view techniques that focus on reducing the g-buffer size, as well as omitting texture fetches in the geometry pass. Due to overdraw, texture fetches are especially costly in this pass.

In deferred texturing, texture fetches are deferred to the shading pass. This implies using a different g-buffer layout. Petino proposes a g-buffer layout that stores depth, material id, uv-coordinates, depth gradients and optionally also uv gradients [30]. Using this data, all texture fetches are done during the shading pass. Still, storing the data requires a medium to large size g-buffer.

3.4.1 The Visibility Buffer

Another approach is described by Hunt and Burns and uses a visibility buffer [32]. The visibility buffer is a minimal g-buffer and only stores data to identify the triangle. For this we use the triangle id and the instance id. We also provide buffers that hold vertex attributes for the shading pass. The geometry pass instead only requires the buffer containing the vertex positions. During shading we use the triangle id and the instance id to read the buffer data for the visible triangle. Next, we manually interpolate the vertex attributes for our fragment position. For other render techniques, these calculations are performed efficiently during rasterization.

First, we perform a ray triangle intersection to calculate the barycentric coordinates inside the triangle. The view ray is derived from the screen coordinates of the pixel. Next, we use the barycentric coordinates to interpolate the vertex attributes. At last, we use the interpolated data to access textures and perform shading operations.

Compared to Petinos approach, using a visibility buffer saves bandwidth by virtue of its small size. However we require additional computation to obtain the barycentric coordinates. While the visibility buffer solution reduces bandwidth, it does not manage rendering semitransparent objects. These we must handle by using a forward pass, as with deferred rendering. Also, rendering partially transparent objects is not addressed with the visibility buffer. Partially transparent objects, such as foliage, require us to perform an alpha test during the geometry pass. That requires early texture access, thus needing material information and uv-coordiantes.

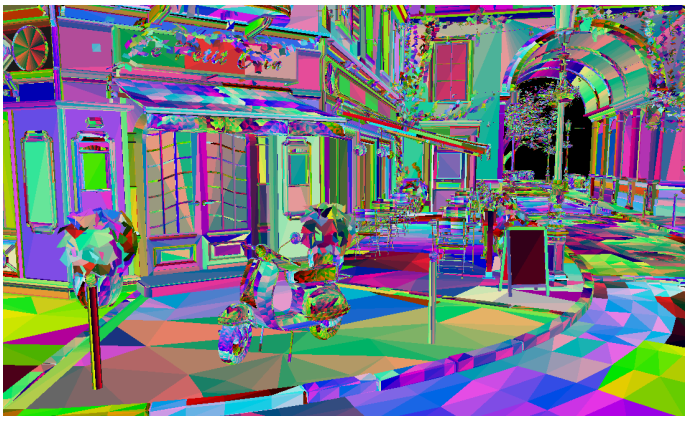


Figure 3.1: *Visualization of a visibility buffer for the bistro exterior [17]. The colors represent different triangle id's.*

3.4.2 Decals in Deferred Texturing

As we do not store any material parameters in the g-buffer, using deferred texturing requires us to reevaluate techniques that rely on this information. For instance, deferred decals do not work anymore, as we cannot blend material parameters to the g-buffer. Petino suggest an approach comparable to decals in DOOM 2016 [30, 16]. In his approach we use clustered deferred texturing to create a list of decals for each cluster. We then iterate over this list to apply decals during shading. With DirectX12 and Vulkan we can store all decal textures in a bindless texture array. Hence, we use texture arrays for globally storing decal data. A similar solution could be also thought of using tiled shading. Our approach builds on the same ideas, but uses a different method to obtain decals per fragment.

4. Ray-Query Decals

In this chapter we describe a system for rendering decals. For that we introduce a new method for retrieving decals per fragment. This method requires us to rethink how we blend decals. On the GPU side, this method is fully implemented in the shading pass. Therefore, it works with all standard render techniques. For demonstration purposes we use a forward renderer, a deferred renderer and visibility buffer renderer.

The key point of our method is the usage of ray queries. We use ray queries to retrieve decals during shading. Until now, ray queries are only available to Vulkan and DirectX12. Also, ray queries are only supported on GPUs with dedicated ray tracing hardware. To-date this applies exclusively to NVIDIAS GeForce RTX series.

In the following, we discuss the main parts of our decal rendering system. Here we present an overview of the components.

- **Loading and Sampling Decals**
  We use a system to load decals and store them into our internal representation. This representation holds all data required to apply decals during shading. It also stores information to construct an acceleration structure for the use of ray queries. For shading we implement the Frostbite standard material. This means decals apply to roughness, metallicity, diffuse albedo and normals. To access textures, we use a bindless texturing systems, which allows us random access to our decal's material textures.
- **Decal Enumeration with Ray Queries**
  We use ray queries to conservatively obtain all decals that overlap a fragment. For this we construct an acceleration structure from the AABBs of all decals.
- **Blending:**
  To blend decals, we use a hybrid blending system. The system holds 4 channels. Channels are blended with alpha blending. Decals that are applied to the same channel are blended with weighted order independent blending.

4.1 Loading and Sampling Decals

To represent decals, we store the following data:

- Inverse transformation,

- Inversed transposed transformation matrix (deferred shading),
- AABB,
- uv-coordinates,
- Blending weight,
- Blending channel,
- Vertices (deferred decals),
- Indices (deferred decals).

A decal is defined by a set of textures mapped to an orthographic projector. The orthographic projector is modeled by an oriented bounding box (OBB) with one projecting side.

To represent the projector, we store its inverse transformation matrix. We use this matrix during shading to obtain a fragments position relative to the decal (see Section 4.1.3). The matrix is constructed from the scale, orientation and position of the decal. Additionally, we store the decals inversed transposed matrix. We us it in deferred shading to transform the decals normals to world space. We store an index for the material we use for the decal. All textures indices are stored by the material data. During the loading process, the AABB is constructed around our OBB. We use it to build the acceleration structure. To map the textures to the projector we store 4 pairs of uv-coordinates. These belong to the 4 vertices of the face that is being projected. Finally, we store the weight and the channel. We use this information for blending. The vertex and the index list represent the box geometry. These are soley used to construct boxes for deferred decals and are not needed for shading. Hence, we use a reduced form for shading. That form only holds the required data and respects memory alignment rules.

4.1.1 Decal Loading

To store and load scene data, we use the .obj format. Decals are represented by a custom definition. We use a name prefix in the 'o' field to identify decals. The projector is stored as a box mesh. We treat the first surface as the projecting side and store its uv-coordinates. By evaluating vertex positions we construct the transformation matrix. To obtain the weight and channel we parse the 'o' field. These values are stored therein as strings. To store geometry and handle transformation we use a scene graph. For a loaded mesh we store all data to a node. Decals are also stored to this node.

The decal representation in .obj format is primarily for convenience. It allows to easily add and manipulate them with 3D modeling software such as Blender.

4.1.2 Material Loading

In our renderer we use the Frostbite standard material definition. Similar material definitions are common among most renderers [6]. Therefore, common test assets can be used by our renderer. For evaluation we use scenes provided by the ORCA library [33]. We use Blender to export scenes to .obj form. Also, all material parameters are stored in textures. When loading a scene, we retrieve textures for all materials and store them to a global texture array. We do not differentiate between materials used for decals and materials used for regular assets. For each material we store a fixed set of indices into the texture array.

With Vulkan's descriptor-based system we can use a bindless approach for handling texture access. For each renderer we specify a descriptor set for each render pass. For the shading pass we store a sampler descriptor for each texture.

During shading, we use an index to access material data for the current primitive. That data provides us with texture indices. We use these to access the texture sampler array.

4.1.3 Sampling Decals

Materials and decals are stored in a storage buffers. To access the data of any decal or material we use its index. In the next chapter (Section 4.2) we discuss how we obtain the decal indices for a fragment.

To sample a decal, we need to find the location on the projected surface, that is orthographically projected onto to the fragments position. For this we transform the position vector into the decals local coordinate system. The transformation is done by multiplying the position with the inverse transformation matrix. The different coordinate systems are illustrated in Figure 4.1.

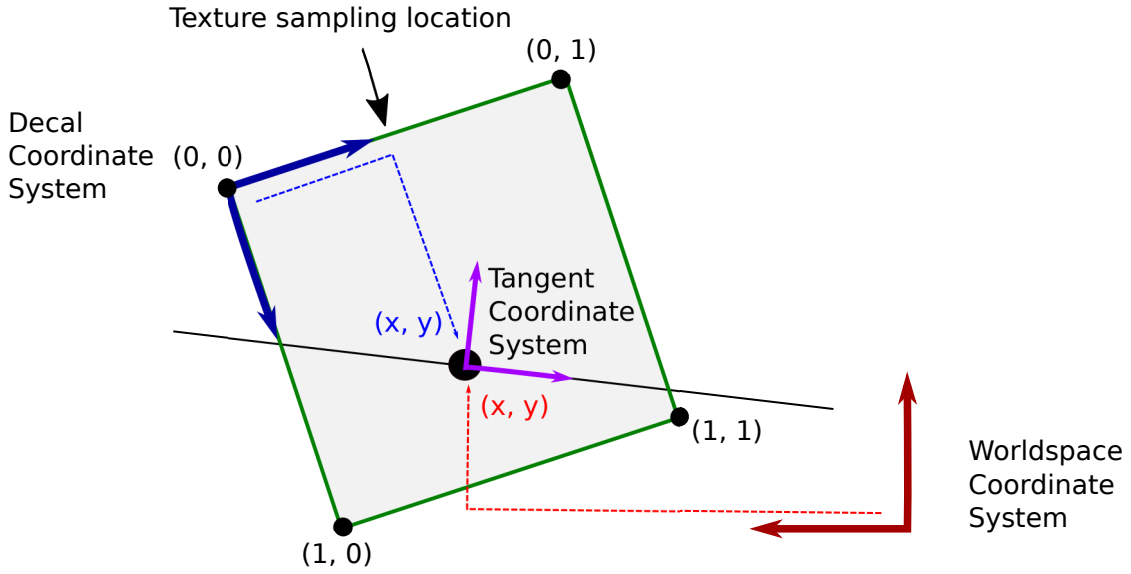


Figure 4.1: *2D visualization, showing the different coordinate systems*

Next, we check if the position is contained inside the OBB of the decal. Checking this is required, as previous checks are conservative and use the AABB. Considering we already have the fragment position inside the coordinate system of the decal, performing this check is largely simplified. We only need to check if the x, y and z components are between 0 and 1. To obtain the uv coordinates we use bilinear interpolation. We utilize the x and y coordinates to interpolate the 4 uv coordinates stored for our decal. Next, we use the material index to access the material data. The material contains indices into the texture sampler array.

Before we can sample textures, we must manually determine gradients for mip mapping. As discussed in Section 2.1 this is done by calculating screen space derivatives. These are made accessible through SIMD processing of 2x2 sets of fragments. As a result, derivatives are only available for code blocks that are always executed for the entire set of 2x2 fragments (Uniform control flow). This means, the ray query loop in which we handle decals does not allow us to use screen space derivatives. Therefore we need another way to determine the mip map level. The solution is to calculate the screen space derivatives of the world space position in advance [34]. For each decal we transform these to local decal coordinates. We use the x and y components of both derivative as gradient vectors to access textures. This solution is not optimal. For deferred shading and deferred texturing with visibility buffer this will cause artifacts to appear around the outlines of objects. Other solutions are discussed by Wronski [35].

With the interpolated uv-coordinates and the acquired gradients we can finally sample textures. Depending on which material parameters the decal applies to, we sample the diffuse texture, the normal map and the texture holding roughness and metallicity. After sampling the normal map, we need to transform the normal to world space. For forward shading and deferred texturing, we use the tangent frame for our fragment. In deferred shading we use the decals inversed transposed matrix [36]. The latter produces false normals if the decal is oriented differently than the surface. However, using the tangent frame requires storing additional information in the g-buffer, hence increasing its size. There exist efficient ways to store this information. One possibility is to use compressed quaternions [37].

4.2 Decal Enumeration with Ray Queries

The main idea behind our decal rendering system, is using ray traversal, to quickly obtain all decals that overlap a fragment. While geometry is defined by its surfaces, a decal is defined by its projection volume. When we ray trace geometry, we need to find primitive intersections along the path of the ray. When testing a fragment for overlap with decals, we only need to check for bounding volumes overlap in one point. We do this by shooting a ray of length 0 from the fragments position. The direction of the ray can be chosen arbitrarily. The information we require from an intersection is the decals index.

To develop this idea, we use functionality exposed by the Vulkan API to perform ray intersection against AABBs in the fragment shader. This method requires two steps:

- Building an acceleration structure that holds the decals AABBs,
- Traversing the acceleration structure to determine all decal overlap in the fragment shader.

4.2.1 Acceleration Structure Construction

As discussed in Section 2.4, ray tracing relies on traversing an acceleration structure, which contains all geometrical primitives. In our case the acceleration structure is a BVH and the geometrical primitives are AABBs. The actual implementation of the acceleration structure is given by the hardware provider. The Vulkan API offers an abstraction that is composed of to levels [38].

The acceleration structure (AS) is composed of a top-level acceleration structure (TLAS) and a set of bottom level acceleration structures (BLAS). The TLAS contains an arbitrary amount of BLAS instances. The instances store their corresponding BLAS and a transformation matrix. The BLAS holds the actual geometry data. This is illustrated in Figure 4.2.

To handle moving objects a BLAS instance can be updated, while preserving the rest of the AS. This makes the acceleration structure capable of handling decals on moving objects. Though we do not implement this in our application. Using multiple BLAS can also be used to classify decals into sets. These sets could be used to create different types of decal, that are only applied to certain objects. In our implementation we do not expand on this possibility. To reduce complexity, we only support static scenes with preplaced decals that can apply to all surfaces.

For every test scene we construct one BLAS and one TLAS. Our TLAS holds one instance of this BLAS. For the transformation, we use the model matrix of our scene. Building the acceleration structure requires performing a lot of memory management. This we do not cover here, as it is described in detail by the official Vulkan ray tracing tutorial [39]. Instead we present an overview of the steps taken to construct the acceleration structure for our decals.

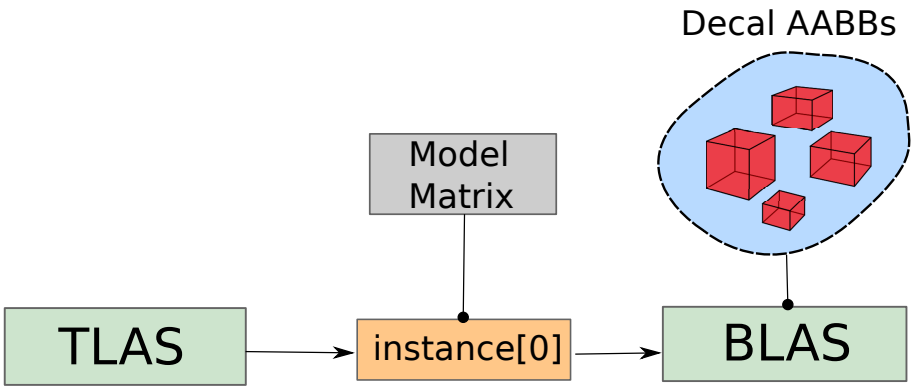


Figure 4.2: *Illustration of the acceleration structure setup used for ray query decals*

- BLAS
  - create a buffer holding the decals AABBs and upload it to the GPU
  - gather the information required to initialize the BLAS
  - use that information to create an empty BLAS
  - allocate memory and build the BLAS on the GPU
- TLAS
  - create a BLAS instance and use the model matrix as transformation matrix
  - create an empty TLAS, holding that instance
  - allocate memory and build the TLAS on the GPU

4.2.2 Ray Query Loop

During the shading pass, we use ray queries to traverse the acceleration structure. The whole process is illustrated in Figure 4.3. Shader commands and data types are provided by the GLSL extension 'GL_EXT_ray_query'.

To initiate traversal, we create a rayQuery object. For the origin of the ray we use the world space position of the fragment. Next, we specify zero for the maximum and minimum distance. For the ray direction we chose any vector. The direction does not matter as our rays have the length 0. Because our rays are in fact points, all rays are coherent anyway and can be processed efficiently by the hardware. The ray is here only evaluated in its origin. To enable further optimization on the driver's side we use ray flags to disable unused functionality.

Finally, we also need to specify the acceleration structure to query against. For this we use the accelerationStructureEXT object for the top level acceleration structure we created for our decals. After the ray query is initialized, we begin ray traversal.

With the rayQueryProceedEXT command we traverse the acceleration structure until a potential hit occurs (see Figure 4.3). As we have discussed in Section 2.4.2, this command performs BVH traversal with depth first search.

When a potential hit occurs, we have intersected the AABB of a leaf node in the BVH. These intersections are calculated conservatively. Therefore they contain false positives at the edges of the AABB. In our case this does not matter, as we perform a check against the decals OBB later anyway. An AABB is visualized in Figure 4.4.

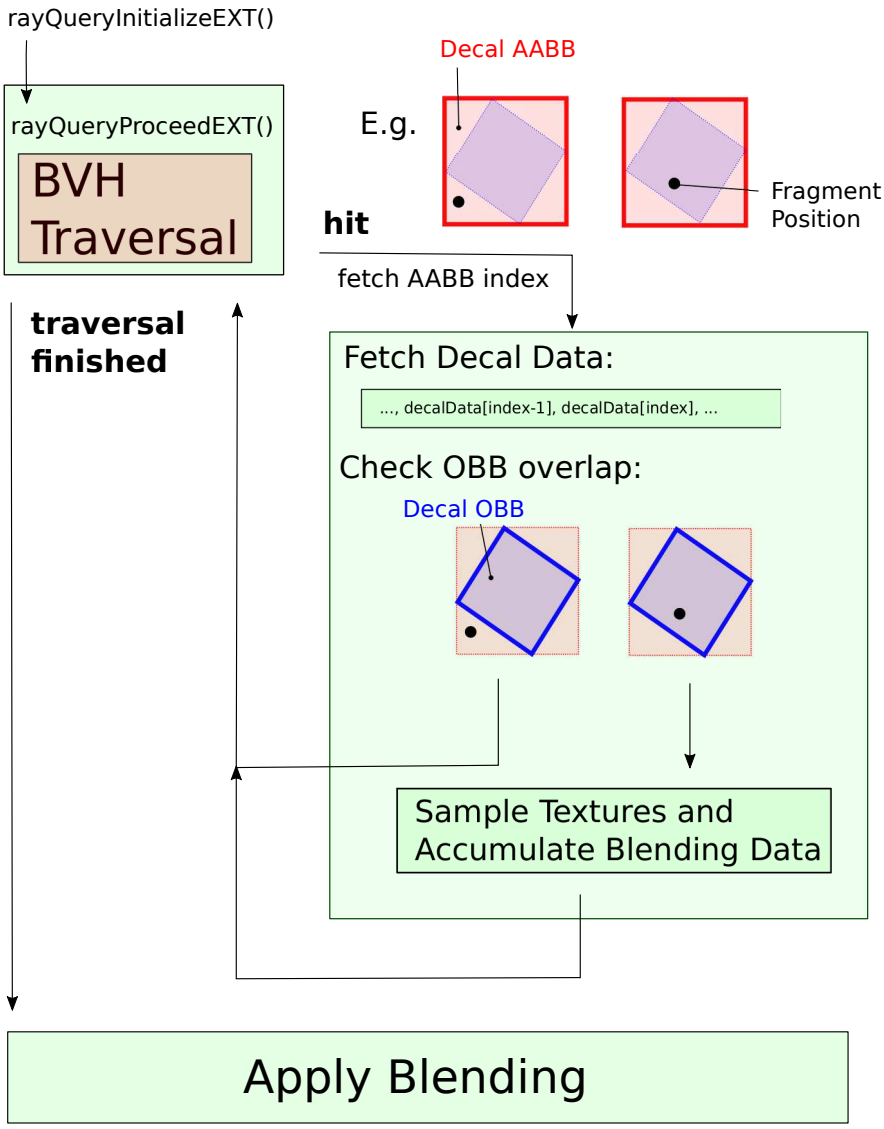


Figure 4.3: *Illustration of the ray query loop, as described in Section 4.2.2*

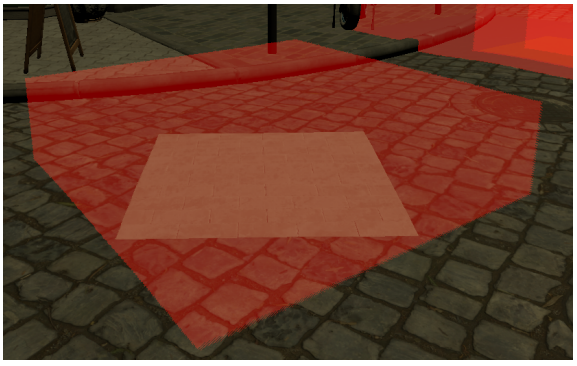


Figure 4.4: *Visualization of an AABB enclosing a decal*

For each intersection we then sample the data of the decal. To obtain the index of the intersected decal we use a built-in function. This function returns the index of the primitive in the buffer, which we used to build the BLAS. We use the same order for this buffer and the storage buffer containing all decal data. Therefore we can use this index to access the decals data. Next, we call the function to apply the decal. This part is explained in Section 4.1.3. Material parameters are therein accumulated to the blending data.

The loop terminates when all intersections with decal AABBs have been found. At that point we have accumulated blending data for all decals overlaping the fragment. In the next step blending is performed. As this the ray query loop is the key component of our technique, we hence show the code used to initialize the ray query and perform the traversal.

```
// Ray Query Loop
  vec3 origin = data.worldPos;
  vec3 direction = vec3(1.0, 0.0, 0.0);
  float tMin = 0.0;
  float tMax = 0.0;
  rayQueryEXT rayQuery;
  rayQueryInitializeEXT(rayQuery, topLevelAS,
      gl_RayFlagsSkipClosestHitShaderEXT
      | gl_RayFlagsOpaqueEXT
      | gl_RayFlagsCullBackFacingTrianglesEXT
      | gl_RayFlagsCullFrontFacingTrianglesEXT,
      0xFF, origin, tMin, direction, tMax);
  while(rayQueryProceedEXT(rayQuery))
  {
    applySingleDecal(blendingData,
      rayQueryGetIntersectionPrimitiveIndexEXT(rayQuery, false),
      data, dx_vtc, dy_vtc);
  }
```

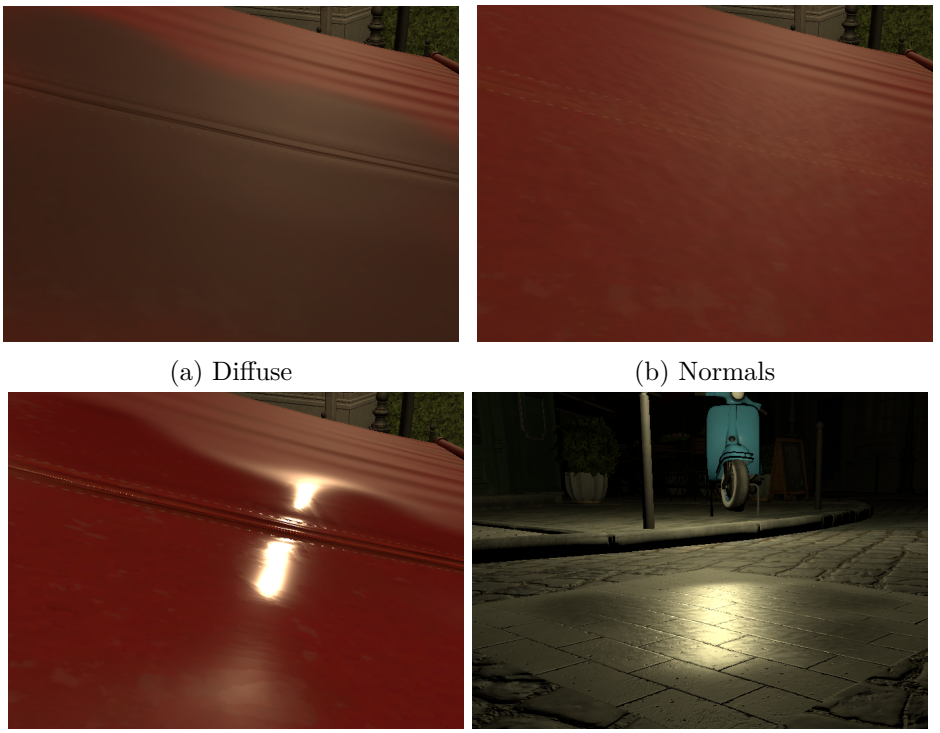
Listing 4.1: *The function 'applySingleDecal(...)' checks if the the fragments position is contained in the OBB of the decal. In that case we accumulate the decals blending data. This process is descibed in Section 4.1.3*

4.3 Blending

As we only use one material definition, decals can always be blended with the underlying surface. Blending materials with different material definitions is therefore a problem we do not regard here. Our material definition holds four parameters: diffuse albedo, surface normal, roughness and metallicity. Blending is handled the same way for all material parameters. We do not treat the normal in a special way, as to be mathematically correct[40]. This could be implemented for alpha blending between channels. To control blending between decals we provide a weight and a channel parameter, that must be set per decal. Those parameters do not affect how the decal is blended to the background.

Blending for deferred decals is normally done with alpha blending. This works provided they share the material definition of the underlying surface and with other decals. As discussed in Section 3.2 deferred decals are applied in a separate render pass, by rendering their box mesh. By sorting the box mesh we can control the order in which they are blended. This allows for (order dependent) alpha blending.

In our case we obtain the decals in the order which they are intersected by the ray query operation. This order is implementation defined and must be assumed to be arbitrary. To apply alpha blending, we would have to store all decals indices into an array, before sorting them after their weight. While sorting is slow, dealing with an array of dynamic size is even worse for our shader program. Therefore our blending method cannot solely



(a) Diffuse (b) Normals

(c) Roughness & Metalicity (d) 2 weighted blended decals

Figure 4.5: Blended Decals

rely on alpha blending. The alternatives are to use weighted OIT, or to use a fixed amount of blending channels. We use a combination by applying weighted OIT per channel and blending channels together with alpha blending. The combination offers us a compromise between being able to handle an arbitrary amount of decals and having fine control over blending.

4.3.1 Per Channel OIT Blending

Our blending method uses the equation described is Section 2.2. Weight is accumulated for all material attributes individually. Only metallicity and roughness share the same weight. Hence, the struct to accumulate blending data for one channel holds three 4-component vectors and three weights. For each vector we use the first three components to accumulate the blending data (diffuse albedo, normals, roughness and metallicity). We use the fourth component to store the current coverage of the background. Before we start the ray query loop, we initialize our blending data for each channel.

During the ray query loop, we then accumulate the decals blending data to its respective blending channel. The channel is stored for each decal. Next, we apply the decal to the blending data of the channel. After the ray query loop is terminated, we use the blending data, to blend all attributes with the current values stored in the shading data. Lighting computations are applied thereafter, using the modified shading data.

```
1    // Diffuse Blending
2    blendingData.color.rgb       += color.rgb * color.a * weight;
3    blendingData.color.a        *= 1.0-color.a;
4    blendingData.weight_color    += color.a * weight;
5    // Normal Blending
6    blendingData.normal.rgb      += normal.rgb * weight * normal.a;
7    blendingData.normal.a       *= 1.0-normal.a;
8    blendingData.weight_normal     += weight * normal.a;
9    // Roughness & Metalicity
```

```
10      blendingData.specularData.rgb  += specularData.rgb * weight *
        specularData.a;
11      blendingData.specularData.a    *= 1.0-specularData.a;
12      blendingData.weight_specularData += specularData.a * weight;
```

Listing 4.2: *Blending data is accumulated for diffuse color, normals roughness and metalicity*

For each attribute we accumulate data for 3 terms of the blending equation described in Section 2.2. The final blending equation has the form:

$$\text{newBackground.rgb} = \frac{\text{blendingData.rgb}}{\text{blendingData.weight}}(1-\text{blendingData.a})+\text{background.rgb}\cdot\text{blendingData.a}$$

The equation is evaluated per attribute. The blending data struct is used to accumulate the values of the terms. Alpha blending between channels is applied by successively evaluating the equation for each channel, using the new background value. For a channel c we accumulate the following terms of our blending equation of all overlapping decals d:

The rgb components accumulate the weighted attributes in the numerator:

$$\text{blendingData.rgb} = \sum_{\text{d}\in Decals_c} \text{d.rgb} \cdot \text{d.a} \cdot \text{d.weight}$$

The a component accumulates the visibility of the background:

$$\text{blendingData.a} = \prod_{d\in Decals_c} (1 - \text{d.a})$$

The accumulated weight forms the denominator:

$$\text{blendingData.weight} = \sum_{\text{d}\in Decals_c} \text{d.a} \cdot \text{d.weight}$$

5. Results

In this chapter we evaluate our technique with different renderers. The renderers are listed here:

- Forward shading with depth prepass,
- Deferred shading,
- Deferred texturing with visibility buffer.

While we try to archive decent visuals, the overall visual quality is not a concern of our implementation. The only quality difference that can arise from using ray query decals over deferred decals, lies in blending. Blending we therefore evaluate explicitely. Hence, we assess our technique in 2 aspects. The main aspect is performance, and scalability. The second aspect is blending. For comparison we also implement deferred decals as described in Section 3.2.

5.1 Implementation

Here we describe our implementation of the rendering systems we use to evaluate ray query decals. We also briefly describe our implementation of deferred decals. The basis of these techniques is described in the background section.

All render techniques are integrated in the same render application. The application is built on the Vulkan SDK 1.2.141.2 ([41]). For window management we use GLFW 3.3.2 ([42]). To load .obj files we use the tinyobjloader, provided on GitHub under the MIT License [43]. To load images, we use stb image, provided under public domain [44].

Systems for scene loading and acceleration structure construction are shared between the renderers. The 3 renderers manage their respective render pipelines and render passes, as well as their specific resources. We change between renderers at runtime without duplicating or reloading scene data. All shader functionality such as decals and lights can be toggled on an off at runtime. This invokes recompiling the shaders without given features. To compile shaders, we use the glslang validator provided with the Vulkan SDK. Optimizations are turned on by default. We use a swapchain with 3 swapchain images and use the mailbox present mode (no V-Sync). Synchronization of GPU and CPU is handled with one fence per swapchain image.

The general implementation for forward rendering and image loading largely follows basic Vulkan Tutorials [45, 46]. The acceleration structure construction is based on the nvpro

samples, by NVIDIA. [39] The deferred renderer and several small sections are inspired by the Vulkan example provided on GitHub by Sascha Willems [47]. Different concepts and solutions for shader programing are inspired by the renderer provided by Dr. Christoph Peters.

For all renderers lighting is performed in the shading pass. Light is applied additively for every light source. All light sources are point lights. For local light sources we offer two settings. The default settings are to treat all lights globally, therefore computing illumination for every light source per fragment. We use this setting to evaluate our first three test scenes. The experimental setting uses the same method we use for ray query decals to apply lights. This setting is used for the fourth test scene. We further discuss this topic for future work in Section 6.1.

Finally, we offer some other options, such as: ray query shadows, ray query reflections, visualizing AABBs, ambient term, emissive term. Not all these features are implemented properly for every renderer. For evaluation, these features are turned off. Next, we describe the implementation of our renderers.

5.1.1 Forward Shading with Depth Prepass

Our first renderer is a forward renderer with a depth prepass. The framebuffer therefore holds two render targets: Color and depth.

The depth pass only stores the depth value to the depth buffer. We still need to execute a fragment shader to perform alpha testing. This allows us to render cutout material, without increasing overdraw during the shading pass. To reduce texture access, we only perform alpha testing on materials that use an alpha map. As mip mapping blurs the edges on cut out geometry, we perform the alpha test against the value of 0.5.

The shading pass uses the depth buffer to perform an early z test. Here we limit depth buffer access to read only. We then obtain shading data from the interpolated vertex data. When reading normal maps, we transform normals to world space, by using the tangent frame. Next, we apply decals. Finally, we use the shading data to perform shading in world space. Hence, the output is written to the color buffer.

5.1.2 Deferred Shading

We implement deferred shading by using multiple subpasses. For the g-buffer we use the following layout:

- Diffuse albedo ($3 \cdot 8$ bit diffuse albedo + 8 bit unused alpha),
- Surface normals ($3 \cdot 10$ bit normal + 2 bit unused alpha),
- Roughness metalicity (8 bit roughness, 8 bit metalicity).

To reconstruct the world space position during shading, we also store the depth buffer.

Vulkan introduces the concept of subpasses, which offers optimizations primarily on tile based hardware (normally used for mobile devices). Subpasses may be used instead of multiple render passes, if for a given pixel, we only access pixel data of the same pixel from earlier passes. For further information Sascha Willems offers a comprehensive description of subpasses in Vulkan [48]. As we only need local pixel data for shading, we use subpasses.

The geometry pass works as described in 2.6.1. As with the depth pass, we perform an alpha test to handle cut out geometry. Next, we store diffuse albedo, normal, roughness and metallicity to the g-buffer. For the shading we render a screen filling triangle [47]. In the fragment shader we fetch all attributes from the g-buffer and continue as with forward shading.

5.1.2.1 Deferred Decals

We apply deferred decals in an optional subpass between geometry pass and shading pass.

To render decals, we use a vertex buffer holding the box geometry per decal. Prior to rendering, we create this buffer from a sorted list of all decals. These we sort after their weight and channel in ascending order. This allows us to use alpha blending for deferred decals.

In the subpass the box meshes are rasterized. Then, for a fragment we reconstruct the position from the depth buffer. As described in Section 3.2 we check if the fragment is contained inside the decals OBB. If this is the case, we sample the decal data, and blend it to the g-buffer.

We need to treat the special case that the camera is located inside a decals OBB. In this case we check every pixel for containment inside the decal. We do this by transforming the first triangle of the box mesh into a screen filling triangle. Other triangles of the same decal are discarded due to backface culling.

5.1.3 Deferred Texturing with Visibility Buffer

Our last renderer uses a visibility buffer. Overall, the renderer works similarly to the deferred rendered. Only in this case we store a 32-bit buffer holding the primitive index. We do not need an instance index.

Again, we use multiple subpasses. The visibility (or geometry) pass only processes the vertex positions. As we do not have access to uv coordinates and the material id, we cannot perform an alpha test. Also, we do not want to perform any texture access in the visibility pass. Therefore we do not properly handle cut out geometry with this renderer.

As in deferred shading, we perform the shading pass by rendering a screen filling triangle. In the fragment shader we fetch the triangle index for each pixel. We then use this index to access the buffers holding the vertex attributes. As described in Section 3.4.1 we calculate the barycentric and then interpolate the vertex data. To access textures, we use implicit derivatives. These are wrong for 2x2 sets of fragments that do not share the same primitive. This is a known problem, but we have not fixed it yet for our rendered. As a solution we could calculate the analytic derivatives to obtain texture gradients. Lastly, we apply decals and computer lighting as with the other renderers.

5.2 Evaluation

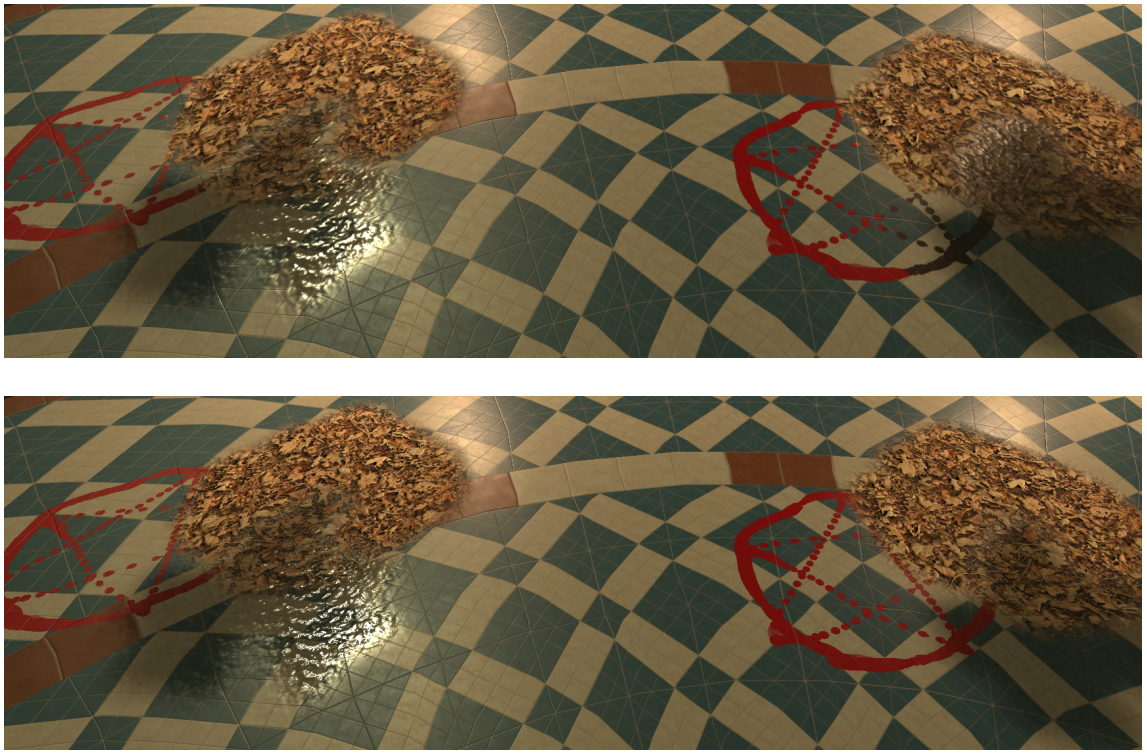
In the following section we evaluate our technique for each renderer with 4 test scenes. For each renderer we measure the average frame time across 200 frames. We measure the frame time with decals and the frame time when branching over the entire decal step (No Decals). We do not remove the decal step at compile time, as compiling the shaders without any ray query functionality, causes optimizations on the compilers side, that introduces additional complications. We also measure the frame time for deferred decals in deferred rendering. The first scene only contains a few decals. Here we focus on evaluating blending. The second scene evaluates performance for a more realistic scenario. The third scene stands for the extreme case and tests the scalability of our technique. The fourth scene uses the same technique we use for decals to render many local light sources. All scenes are common test scenes from the ORCA library [33].

5.2.1 System Specifications

All scenarios were tested on the same hardware:

- GPU: NVIDIA RTX 2070 SUPER (VRAM: 8GB GDDR6),
- CPU: Intel Core i7-7700 (3.60 GHz),
- RAM: 16 GB DDR4,
- OS: WINDOWS 10 (64 Bit),
- Resolution:1920x1080 (Full HD),
- Graphics Driver Version: 451.74 .

5.2.2 Few Blended Decals



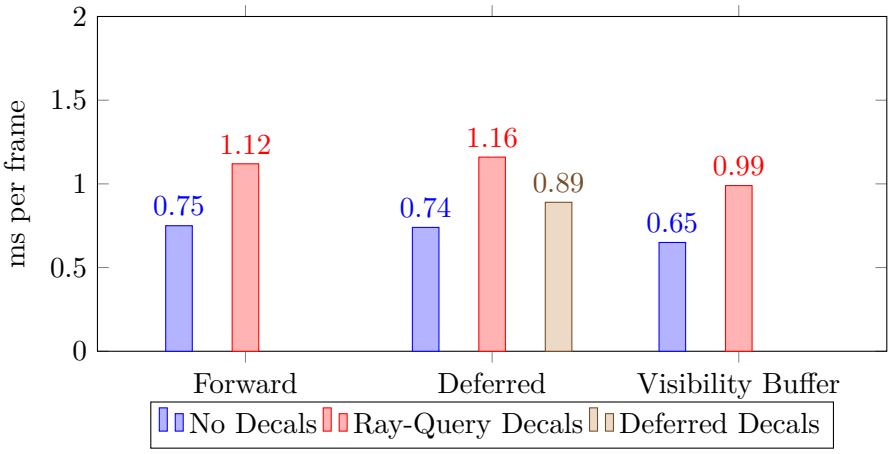


Figure 5.1: *Test scenario demonstrating decal blending. The scene show the ground of the UE4 Sun Temple [49] with 6 decals [50]. (Top) Ray query decals in deferred rendering (Middle) Deferred decals. (Bottom) Bar chart, showing frame times*

For our first scenario we place a small amount of decals in the Sun Temple Scene [49]. We use a few light sources for illumination. All decals are visible and cover a significant portion of the screen. The characteristics of this test are:

- Vertex count: 1,641,711,
- Decal count: 6,
- Light sources: 3.

First, we compare our blending strategy to pure weighted OIT. Figure 5.1 (Top) shows to groups of 3 ray query decals. In the right group we blend all decals in one channel with weighted OIT. We model the decals to be layered on top of each other. We chose the

weights accordingly. For the pentagram, the leaves and the puddle we specify the following weights: 1, 100, 1000. While the leaves blend properly with the water and the pentagram, the watercolor almost completely overwrites the pentagram color. This is caused by the large difference in weight between water and pentagram decal. On the left side we use the same weights, but we assign the puddle to the blending channel 1. This causes the puddle to be blended over the other decals with alpha blending. We see that the usage of a different channel allow for it blend properly with the other decals.

Figure 5.1 (Middle) shows the same decals, rendered as deferred decals. We order the decals after their weight and channel (see section 5.1.2.1) and blend with alpha blending.

In the left group our blending technique archieves comparable results to pure alpha blending while in the right group we see a notable difference where the pentagram is underwater.

For the frame time we observe a similar increase between 0.34ms and 0.42ms for all render techniques when we activate ray query decals. For deferred decal, the gap is only 0.15ms. Therefore, in this scene deferred decals prove significantly faster than ray query decals. The camera view is selected so that the decals cover large areas of the screen, while not much other geometry is visible. Therefore the substantial proportion of frame time dedicated to rendering decals is to be expected.

5.2.3 Many Decals with many Lights

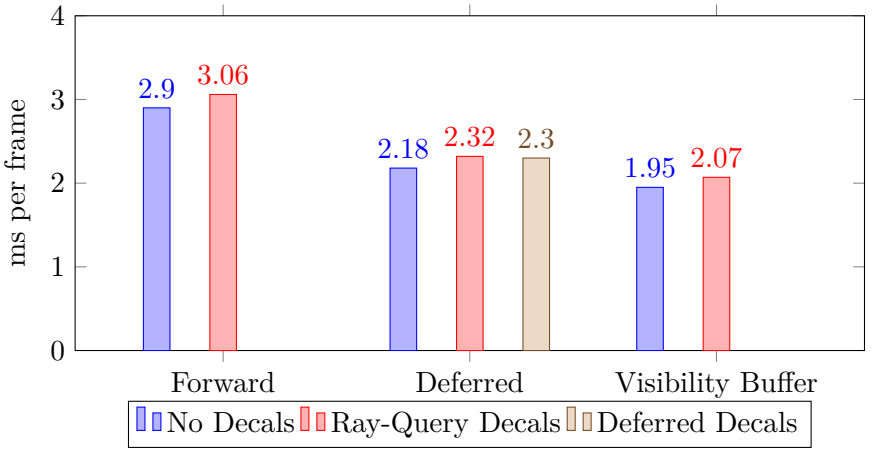


Figure 5.2: *Test scenario with 100 decals. (Top) The scene shows a view of the Amazon Lumberyard Bistro covered in puddle decals, rendered with deferred texturing (visibility buffer) [17]. (Bottom) Bar chart, showing frame times*

For a more realistic scenario we place 100 puddle decals in the Amazon Lumberyard Bistro Scene [17]. We use 28 light sources, placed a the lanterns. Many of the decals are occluded or cover only a small screen area. Almost all are contained in the view frustum. The characteristics of this test are:

- Triangle count: 2,832,120,
- Decal count: 100,
- Light sources: 28.

The impact on the frame time is again similar for all rendering techniques. Here it ranges from 0.12ms to 0.16 ms. Here we note that ray query decals in deferred shading are almost equally fast to deferred decals. Shifting the camera view lower to the ground further reduces the difference between deferred and ray query decals. Moving the camera up higher increases it. At very flat angles ray query decals are even faster than deferred decals. To explain this behavior, we analyze the proportion of false positives, that are processed per

fragment. For ray query decals this equals the proportion of visible fragments, which are contained in the decals AABB but not in its OBB. Deferred decals however are processed for all fragments covered behind the rasterized bounding box of the decal. For flat angle, the boxes cover up many fragments that are not contained within the decals. For flat surfaces the number of false positives can be minimized by using thin slabs for the decals bounding boxes. Uneven surfaces require thicker decal boxes and therefore produces more false positives. In our example we use cubical bounding boxes, hence these are not optimized for deferred decals. Still this example shows an overall satisfactory performance of ray query decals in a more realistic setting with a sparse coverage of decals.

5.2.4 Very many Decals

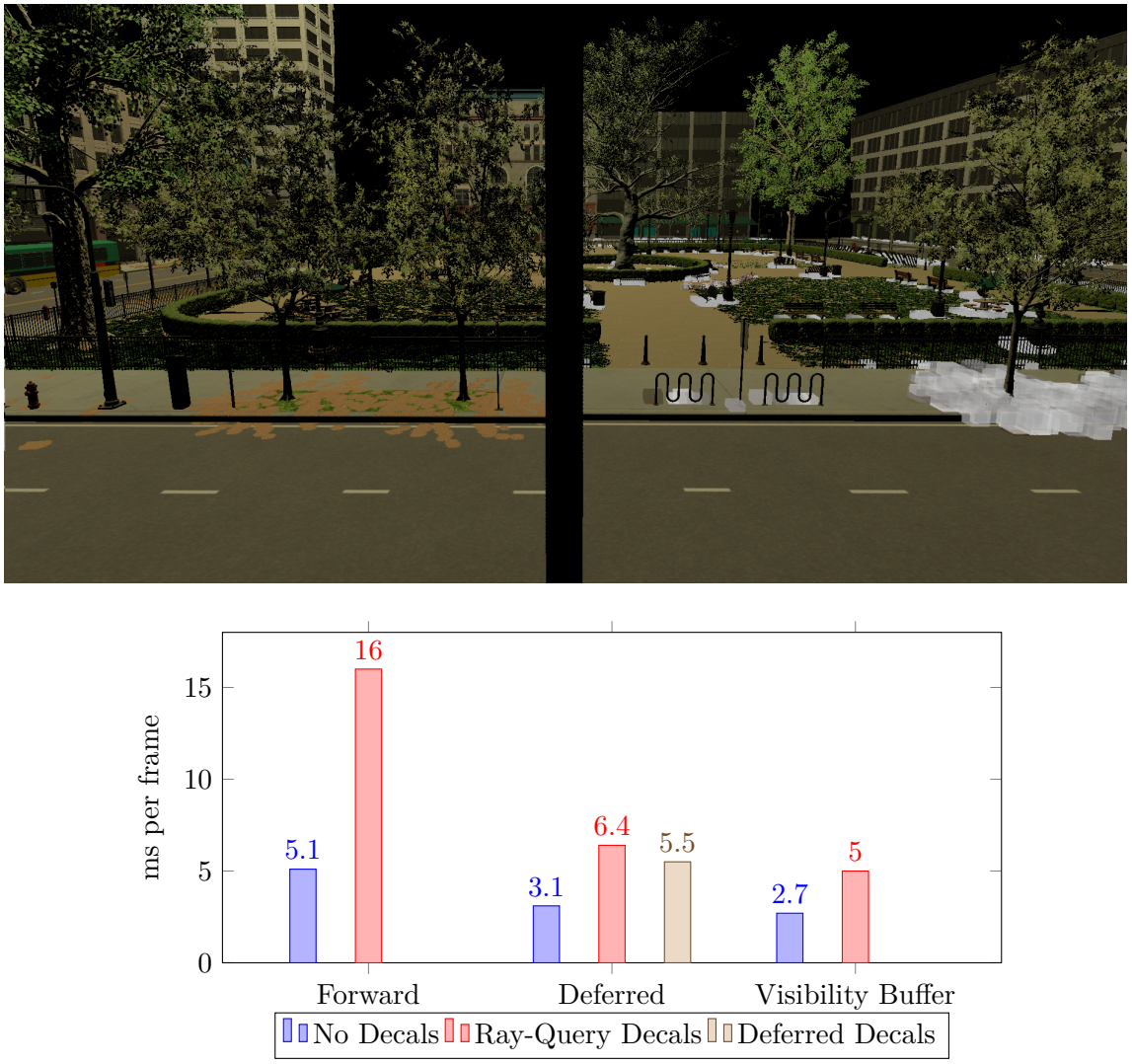


Figure 5.3: *Test scenario with 100k decals. (Top Left) The scene shows a view of the NVIDIA Emerald Square covered with leaf decals, rendered with forward shading [51]. (Top Right) Visualization of the decal AABBs. (Bottom) Bar chart, showing frame times*

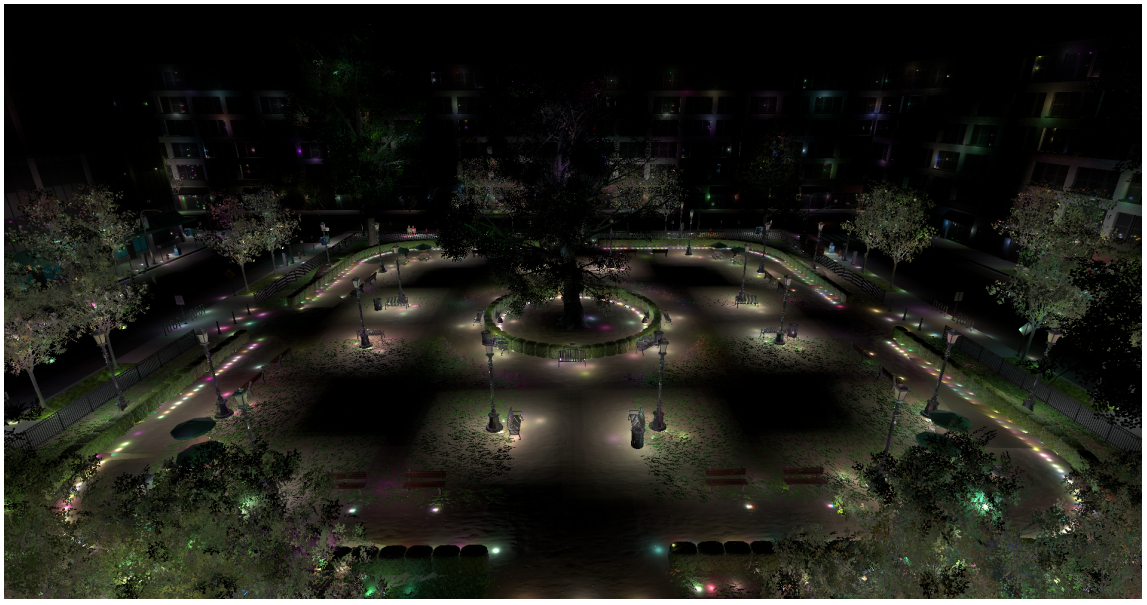
As a stress test for our technique, we place 100.000 leaf decals in the NVIDIA Emerald Square Scene [51]. We use 1 light sources to eluminate the entire scene. While the view contains many decals, that is only a portion of the total decal count. The characteristics of this test are:

- Triangle count: 10,046,405,
- Decal count: 100.000,
- Light sources: 1,
- Half size textures.

We use this test scenario to evaluate the stability of ray queries in the extreme case. The 100.000 decals are generated and applied to the ground of the test scene. The results are mixed. For forward rendering decals have a substantial impact on the framerate, over 10 ms. For deferred shading and deferred texturing, the increase in frame time is moderate,

regarding the number of decal. For deferred shading, ray query decals are only 0.9 ms slower than deferred decals. For the latter 2 render techniques, this example proves the stability for high amounts of ray query decals applied to a large scene.

5.2.5 Very many Lights



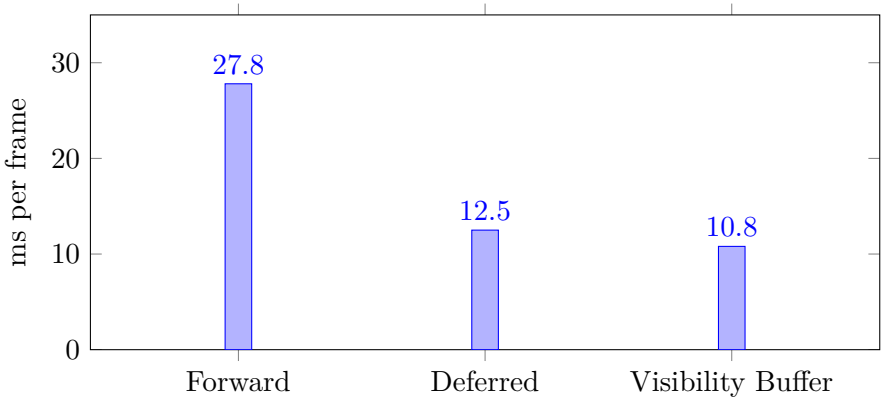


Figure 5.4: *Test scenario with 28k local point light sources. (Top) The scene shows a view of the NVIDIA Emerald Square covered with small light sources [51]. At many places we see the illumination abruptly cut of by the lights AABB. (Bottom) Bar chart, showing frame times*

In this scene we test our method with many local ligts on NVIDIA Emerald Square [51]. The characteristics of this test are:

- Triangle count: 10,046,405,
- Decal count: 0,
- Local light sources: 28587,
- Half size textures.

We here use the method described in section 4.2 to render local lights. Rendering lights is not part of this work, but we regarded it for further work. As with cluster shading and deferred shading, the concept used to render decals can be used to manage any sort

of local influence, such as local lights. In this scenario we generate local pointe lights for 28587 locations. Again, the results are mixed. For forward rendering we leave the domain of real time rendering, while for deferred shading and deferred texturing we reach between 80 and 90 fps. Also, there are different variables that can still be adjusted. For instance, we notice box-shaped artifacts around all light sources. Lights are only evaluated inside their AABBs, but in contrast to decals, they do not have a fixed influence area. The AABB size must be balanced to prevent artifacts, while keeping the average per fragment light count low. We further explore this idea for future work.

6. Conclusion

We conclude that ray query decals are applicable for all evaluated render techniques under normal conditions. Deferred shading performs slightly better with deferred decals. We note that ray query decals are especially interesting for deferred texturing systems, such as the visibility buffer renderer. The results we obtain are promising, even for large numbers of decals. As these renderers cannot rely on deferred decals we herein see a viable alternative for rendering decals with deferred texturing.

As it stands, we have proven the viability and feasibility, of our approach to rendering decals. Similar options to render decals, such as decals in cluster shading remain to be compared with ray query decals. Also, it would make sense to consider implementing ray query decals into other render techniques, such as real-time ray tracing.

Over the course of this work we have discovered that the method used for ray query decals can be modified to solve similar problems, such as rendering local light sources. In the next section we discuss the application of this concept for local point light sources.

6.1 Future Work

Figure 6.1: (Left) View of the tower in Emerald square with 1.3 Million small local point lights distributed throughout the entire scene. (Right) Visualized light AABBs

Approaches for rendering decals and rendering local lights must solve the same underlying problem. Local lights and decals are entities that only apply to fragments inside their bounding volume. We require an efficient way to apply entities only to those fragments. In deferred shading this is done by rasterizing a proxy-mesh representing that bounding volume. In tiled and cluster shading we calculate a list of entities that must be evaluated per tile/cluster. With ray query decals we traverse an acceleration structure to enumerate all decals per fragment. Naturally, this method can be used for any entity with a local influence area.

In Section 5.2.5 we test this possibility with many local point lights. In Figure 6.1 we take the light count even higher, while reducing the lights intensity and resulting AABB size. In all test cases we still encounter visible artifacts, where lights are cut of by their AABBs. Also, we only support this technique with point lights at the time. Another problem are specular highlights on smooth materials. These can be cause by light sources far away. Therefore, capturing the influence area of a light in a fixed sized AABB for all materials is very inaccurate. For future work we consider testing and optimizing this method for point lights and exploring possibilities for other light sources.

7. Acknowledgements

I would like to pay my special regards to my supervisor Dr. Christoph Peters, for the expertise and aid he provided over the course of this thesis. The numerous explanations helped me a lot in implementing the different render techniques in Vulkan. I thank him a lot for his dedication to help me uncover many bugs and oddities I planted into my application. I am also am very thankful for all the motivation and feedback I received while writing my thesis.

Besides, I would like to acknowledge the Vulkan tutorials, especially the one of Brotcrunsher, which got me through my first weeks of learning Vulkan. From there on the extensive collection of Vulkan examples and forum answers of Sascha Willems were very helpful.

Finally, I am grateful for all the professionally created assets provided in the ORCA library. These allowed me to demonstrate and test my implementation with realistic and visually appealing assets. All those assets are linked in the bibliography.

Bibliography

[1] L. Williams, "Pyramidal parametrics," *SIGGRAPH Comput. Graph.*, vol. 17, no. 3, p. 1–11, Jul. 1983. [Online]. Available: https://doi.org/10.1145/964967.801126

[2] K. Bjorke, "High-quality filtering," in *GPU Gems*, R. Fernando, Ed. Boston: Addison-Wesley, 2004. [Online]. Available: https://developer.nvidia.com/gpugems/GPUGems/gpugems_ch24.html

[3] T. Porter and T. Duff, "Compositing digital images," *SIGGRAPH Comput. Graph.*, vol. 18, no. 3, p. 253–259, Jan. 1984. [Online]. Available: https://doi.org/10.1145/964965.808606

[4] M. McGuire and L. Bavoil, "Weighted blended order-independent transparency," *Journal of Computer Graphics Techniques (JCGT)*, vol. 2, no. 2, pp. 122–141, December 2013. [Online]. Available: http://jcgt.org/published/0002/02/09/

[5] C. Münstermann, S. Krumpen, R. Klein, and C. Peters, "Moment-based order-independent transparency," *Proc. ACM Comput. Graph. Interact. Tech.*, vol. 1, no. 1, Jul. 2018. [Online]. Available: https://doi.org/10.1145/3203206

[6] S. Lagarde and C. De Rousiers, "Moving frostbite to pbr," *Proc. Physically Based Shading Theory Practice*, 2014. [Online]. Available: https://seblagarde.files.wordpress.com/2015/07/course_notes_moving_frostbite_to_pbr_v32.pdf

[7] T. Möller, E. Haines, and N. Hoffman, *Real-Time Rendering, Fourth Edition.* A K Peters/CRC Press, 2018, vol. Fourth edition. [Online]. Available: http://www.redi-bw.de/db/ebsco.php/search.ebscohost.com/login.aspx%3fdirect%3dtrue%26db%3dnlebk%26AN%3d1855548%26site%3dehost-live

[8] P. Beckmann and A. Spizzichino, "The scattering of electromagnetic waves from rough surfaces," *ah*, 1987.

[9] B. Walter, S. R. Marschner, H. Li, and K. E. Torrance, "Microfacet models for refraction through rough surfaces." *Rendering techniques*, vol. 2007, p. 18th, 2007.

[10] B. Smith, "Geometrical shadowing of a random rough surface," *IEEE Transactions on Antennas and Propagation*, vol. 15, no. 5, pp. 668–671, 1967.

[11] E. Heitz, "Understanding the masking-shadowing function in microfacet-based brdfs," *Journal of Computer Graphics Techniques (JCGT)*, vol. 3, no. 2, pp. 48–107, June 2014. [Online]. Available: http://jcgt.org/published/0003/02/03/

[12] C. Schlick, "An inexpensive bdrf model for physically based rendering," *Computer Graphics Forum*, vol. 13, no. 3, pp. 149–162, September 1994.

[13] B. Burley and W. D. A. Studios, "Physically-based shading at disney," in *ACM SIGGRAPH*, vol. 2012, 2012, pp. 1–7.

[14] Ray tracing. Accessed: 2020-09-19. [Online]. Available: https://developer.nvidia.com/discover/ray-tracing

[15] T. Saito and T. Takahashi, "Comprehensible rendering of 3-d shapes," in *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '90. New York, NY, USA: Association for Computing Machinery, 1990, p. 197–206. [Online]. Available: https://doi.org/10.1145/97879.97901

[16] The devil is in the details. Accessed: 2020-09-15. [Online]. Available: https://advances.realtimerendering.com/s2016/Siggraph2016_idTech6.pdf

[17] A. Lumberyard, "Amazon lumberyard bistro, open research content archive (orca)," July 2017, `http://developer.nvidia.com/orca/amazon-lumberyard-bistro`. [Online]. Available: http://developer.nvidia.com/orca/amazon-lumberyard-bistro

[18] Doom eternal - the digital foundry tech review. Accessed: 2020-09-19. [Online]. Available: https://www.youtube.com/watch?v=UsmqWSZpgJY&ab_channel=DigitalFoundry

[19] M. Evans. (2015, February) Drawing stuff on other stuff with deferred screenspace decals. Accessed: 2020-09-19. [Online]. Available: https://martindevans.me/game-development/2015/02/27/Drawing-Stuff-On-Other-Stuff-With-Deferred-Screenspace-Decals/

[20] D. Rosen. (2009, June) How to project decals. Accessed: 2020-09-19. [Online]. Available: http://blog.wolfire.com/2009/06/how-to-project-decals/

[21] P. Kim, "Screen space decals in warhammer 40,000: Space marine," in *ACM SIGGRAPH 2012 Talks*, ser. SIGGRAPH '12. New York, NY, USA: Association for Computing Machinery, 2012. [Online]. Available: https://doi.org/10.1145/2343045.2343053

[22] R. Zioma, "Better geometry batching using light buffers," *SHADERX4 Advanced Rendering Techniques*, pp. 5–16, 2006.

[23] D. Trebilco, "Light-indexed deferred rendering," Dec 2007. [Online]. Available: https://github.com/dtrebilco/lightindexed-deferredrender/blob/master/LightIndexedDeferredLighting1.1.pdf

[24] C. Balestra, "The technology of uncharted drake's fortune," in *Game Developers Conference 2008*, 2008.

[25] G. Thomas, "Advancements in tiled-based compute rendering," in *Game Developers Conference*, 2015, pp. 803–894.

[26] T. Harada, "A 2.5d culling for forward+," in *SIGGRAPH Asia 2012 Technical Briefs*, ser. SA '12. New York, NY, USA: Association for Computing Machinery, 2012. [Online]. Available: https://doi.org/10.1145/2407746.2407764

[27] O. Olsson, M. Billeter, and U. Assarsson, "Clustered deferred and forward shading," in *Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics*. Citeseer, 2012, pp. 87–96.

[28] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha, "Fast bvh construction on gpus," in *Computer Graphics Forum*, vol. 28, no. 2. Wiley Online Library, 2009, pp. 375–384.

[29] E. Persson, "Practical clustered shading," *SIGGRAPH Course: Advances in Real-Time Rendering in Games*, 2013.

[30] BINDLESS TEXTURING FOR DEFERRED RENDERING AND DECALS. Accessed: 2020-09-15. [Online]. Available: https://mynameismjp.wordpress.com/2016/03/25/bindless-texturing-for-deferred-rendering-and-decals/

[31] Gpu-driven rendering pipelines. Accessed: 2020-09-15. [Online]. Available: http://advances.realtimerendering.com/s2015/aaltonenhaar_siggraph2015_combined_final_footer_220dpi.pptx

[32] C. A. Burns and W. A. Hunt, "The visibility buffer: a cache-friendly approach to deferred shading," *Journal of Computer Graphics Techniques (JCGT)*, vol. 2, no. 2, pp. 55–69, 2013.

[33] Orca library. Accessed: 2020-09-19. [Online]. Available: https://developer.nvidia.com/orca

[34] Forward+ decal rendering. Accessed: 2020-09-17. [Online]. Available: https://wickedengine.net/2017/10/12/forward-decal-rendering/

[35] Fixing screen-space deferred decals. Accessed: 2020-09-15. [Online]. Available: https://bartwronski.com/2015/03/12/fixing-screen-space-deferred-decals/

[36] Normal vector transformation. Accessed: 2020-09-15. [Online]. Available: https://www.cs.upc.edu/~robert/teaching/idi/normalsOpenGL.pdf

[37] The bitsquid low level animation system. Accessed: 2020-09-15. [Online]. Available: http://bitsquid.blogspot.com/2009/11/bitsquid-low-level-animation-system.html

[38] Ray tracing in vulkan. Accessed: 2020-09-15. [Online]. Available: https://www.khronos.org/blog/ray-tracing-in-vulkan

[39] Nvidia vulkan ray tracing tutorial. Accessed: 2020-09-15. [Online]. Available: https://nvpro-samples.github.io/vk_raytracing_tutorial_KHR/

[40] Blending in detail. Accessed: 2020-09-15. [Online]. Available: https://blog.selfshadow.com/publications/blending-in-detail/

[41] Lunargvulkan. Accessed: 2020-09-19. [Online]. Available: https://vulkan.lunarg.com/

[42] Glfw. Accessed: 2020-09-19. [Online]. Available: https://www.glfw.org/

[43] tinyobjloader. Accessed: 2020-09-19. [Online]. Available: https://github.com/tinyobjloader/tinyobjloader

[44] stb image. Accessed: 2020-09-19. [Online]. Available: https://github.com/nothings/stb/blob/master/stb_image.h

[45] Vulkan tutorial. Accessed: 2020-09-19. [Online]. Available: https://vulkan-tutorial.com/

[46] Brotcrunsher vulkan tutorial. Accessed: 2020-09-19. [Online]. Available: https://www.youtube.com/watch?v=mzVFHEmnRLg&ab_channel=Brotcrunsher

[47] Vulkan c++ examples and demos. Accessed: 2020-09-19. [Online]. Available: https://github.com/SaschaWillems/Vulkan

[48] Vulkan input attachments and sub passes. Accessed: 2020-09-19. [Online]. Available: https://www.saschawillems.de/blog/2018/07/19/vulkan-input-attachments-and-sub-passes/

[49] E. Games, "Unreal engine sun temple, open research content archive (orca)," October 2017, `http://developer.nvidia.com/orca/epic-games-sun-temple`. [Online]. Available: http://developer.nvidia.com/orca/epic-games-sun-temple

[50] Leaf texture. Accessed: 2020-09-19. [Online]. Available: https://3dtextures.me/2016/05/16/dead-leaves-001/

[51] K. A. Nicholas Hull and N. Benty, "Nvidia emerald square, open research content archive (orca)," July 2017, `http://developer.nvidia.com/orca/nvidia-emerald-square`. [Online]. Available: http://developer.nvidia.com/orca/nvidia-emerald-square

Erklärung

Ich versichere, dass ich die Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe. Die Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt und von dieser als Teil einer Prüfungsleistung angenommen.

Karlsruhe, den September 20, 2020

(Sidney Hansen)