

Aalto University
School of Science
Degree Programme in Computer Science and Engineering

Morley Zan-Bi

Comparative Evaluation of Glossy Surface Shading Using Object-Space Lighting and Screen-Space Shading

Master's Thesis
Espoo, June 4, 2018

Supervisor: Jaakko Lehtinen, D.Sc. (Tech.), Professor, Aalto University
School of Science
Advisor: Jaakko Lehtinen, D.Sc. (Tech.), Professor, Aalto University
School of Science

Aalto University
 School of Science
 Degree Programme in Computer Science and Engineering

ABSTRACT OF
 MASTER'S THESIS

Author:	Morley Zan-Bi		
Title:	Comparative Evaluation of Glossy Surface Shading Using Object-Space Lighting and Screen-Space Shading		
Date:	June 4, 2018	Pages:	vi + 117
Major:	Computer Science	Code:	T-111
Supervisor:	Jaakko Lehtinen, D.Sc. (Tech.), Professor, Aalto University School of Science		
Advisor:	Jaakko Lehtinen, D.Sc. (Tech.), Professor, Aalto University School of Science		
<p>The field of computer graphics places a premium on obtaining an optimal balance between the fidelity of visual of representation and the performance of rendering. The level of fidelity for traditional shading techniques that operate in screen-space is generally related to the screen resolution and thus the number of pixels that we render. Special application areas, such as stereo rendering for virtual reality head-mounted displays, demand high output update rates and screen pixel resolutions which can then lead to significant performance penalties. This means that it would be beneficial to utilize a rendering technique which could be decoupled from the output update rate and resolution, without too severely affecting the achieved rendering quality.</p> <p>One technique capable of meeting this goal is that of performing a 3D model’s surface shading in an object-specific space. In this thesis we have implemented such a shading method, with the lighting computations over a model’s surface being done on a model-specific, uniquely parameterized texture map we call a <i>light map</i>. As the shading is computed per light map texel, the costs do not depend on the output resolution or update rate. Additionally, we utilize the texture sampling hardware built into the Graphics Processing Units ubiquitous in modern computing systems to gain high quality anti-aliasing on the shading results. The end result is a surface appearance that is expected to theoretically be close to those resulting from highly supersampled screen-space shading techniques.</p> <p>In addition to the object-space lighting technique, we also implemented a traditional screen-space version of our shading algorithm. Both of these techniques were used in a user study we organized to test against the theoretical expectation. The results from the study indicated that the object-space shaded images are perceptually close to identical compared to heavily supersampled screen-space images.</p>			
Keywords:	computer graphics, real-time rendering, object-space lighting, texture-space shading, user study		
Language:	English		

Aalto-yliopisto
 Perustieteiden korkeakoulu
 Tietotekniikan koulutusohjelma

DIPLOMITYÖN
 TIIVISTELMÄ

Tekijä:	Morley Zan-Bi		
Työn nimi:	Komparatiivinen arviointi kiiltävien pintojen valaistustuloksista mallintilan valaistuksen ja ruuduntilan valaistuksen välillä		
Päiväys:	4. kesäkuuta 2018	Sivumäärä:	vi + 117
Pääaine:	Computer Science	Koodi:	T-111
Valvoja:	Jaakko Lehtinen, TkT		
Ohjaaja:	Jaakko Lehtinen, TkT		
<p>Tietokonegrafikan alalla optimaalisen tasapainon saavuttaminen kuvanlaadun ja laskentanopeuden välillä on keskeisessä asemassa. Perinteisillä, kuvaruuduntilassa toimivilla valaistusalgoritmeilla kuvanlaatu on tyypillisesti riippuvainen käytetyn piirtoikkunan erottelutarkkuudesta ja näin ollen kuvaelementtien kokonaismäärästä. Tietysti sovellusalueet, kuten stereopiirtäminen keinotodellisuussovelluksille, edellyttävät korkeata ruudunpäivitystaajuutta sekä erottelutarkkuutta, mikä taas johtaa laskentatehovaatimusten kasvuun. Näin ollen on tarkoituksenmukaista hyödyntää algoritmeja, joissa valaistuskalkulaatio saataisiin erotettua näistä ominaisuuksista ilman merkittävää kuvanlaadun heikkenemistä.</p> <p>Yksi algoritmikategoria, joka täyttää nämä asetetut vaatimukset on valaistuskalkulaatio 3D-mallikohtaisessa tilassa. Tämän diplomityön puitteissa olemme toteuttaneet tähän kategoriaan lukeutuvan valaistusalgoritmin, jossa valaistuskalkulaatio suoritetaan mallikohtaisella, yksikäsitteisesti parametrisoidulla tekstuurikartalla. Tämä tarkoittaa, että valaistuskalkulaatiosta koituvat suorituskysymykset eivät ole riippuvaisia aiemmin mainituista ruudun ominaisuuksista. Valaistuskalkulaatio yksilöllisiin tekstuurikarttoihin mahdollistaa näytönohjaimiin sisäänrakennetun teksturointilaitteiston käyttämisen korkealaatuiseen valaistustulosten suodattamiseen. Lopputuloksena saavutetaan piirretty kuva, jonka teoreettisesti oletetaan olevan laadultaan lähellä merkittävästi ylinäytteistettyä ruuduntilan valaistusalgoritmeille saavutettuja tuloksia.</p> <p>Mallikohtaisen tilan valaistusalgoritmin lisäksi toteutimme perinteisen ruuduntilan valaistusalgoritmiversion. Molempia toteutuksia käytettiin järjestämässämme käyttäjätestissä, jonka tavoitteena oli testata toteutuuko mainittu teoreettinen oletus käytännössä. Käyttäjätestin tulokset viittasivat vahvasti oletuksen pätevyyteen, käyttäjien kokonaisvaltaisesti kokien ylinäytteistetyn perinteisen valaistuskalkulaation tulokset lähes identtisiksi mallintilan valaistuskalkulaation tuloksiin.</p>			
Asiasanat:	tietokonegrafiikka, reaaliaikainen renderöinti, valaistuskalkulaatio mallintilassa, tekstuuritilan valaistuskalkulaatio, käyttäjätesti		
Kieli:	Englanti		

Acknowledgements

I would like to thank my advisor, professor Jaakko Lehtinen, for his guidance throughout the entirety of the thesis work and for being a reference on what to strive towards. I also wish to thank Jukka Häkkinen for the valuable feedback he provided on the user study's design.

Finally, but most importantly, I would like to thank my mother Eija for giving me the opportunity to get to this point in life. While I am hopeful that the future still holds even greater achievements, I have also learned the importance of living in the present and being grateful for the positive aspects of one's life.

Espoo, June 4, 2018

Morley Zan-Bi

Contents

1	Introduction	1
1.1	Background and Motivation	1
1.2	Thesis Scope	4
1.3	Structure of the Thesis	4
2	Theoretical Background	5
2.1	Visibility Determination	7
2.1.1	Object Geometry Representation	7
2.1.2	Transforming from Object-Local to View Space	8
2.1.3	Perspective Projection	11
2.1.4	Sampling	14
2.2	Shading	17
2.2.1	The RGB Model and Radiometric Units	18
2.2.2	Physically-Based Rendering	21
2.2.3	Materials	24
2.2.4	Texture Maps	27
2.2.5	Filtering Normal Maps	32
2.2.6	The Cook-Torrance BRDF	33
3	Practical 3D Rendering	41
3.1	The Programmable Graphics Pipeline	42
3.1.1	The Input Assembler and the Vertex Shader	43
3.1.2	The Pixel Shader	44
3.1.3	Forming the Final Image	45
3.1.4	The Compute Shader	45
3.2	Rendering Methods	46
3.2.1	Forward Shading	46
3.2.2	Deferred Shading	48

4	Object-Space Lighting	51
4.1	Motivation for New Rendering Techniques	51
4.2	Shading in Object-Space	53
4.3	Practical Implementation of Object-Space Lighting	54
4.3.1	The Software Used for Implementation	55
4.3.2	Object-Space Lighting Algorithm Overview	56
4.3.3	Mesh Parameterization	57
4.3.4	Rasterization	57
4.3.5	The Edge Function	60
4.3.6	Conservative Rasterization	64
4.3.7	The Structure of a G-Buffer Texel	68
4.3.8	Shading and Normal Mapping	70
4.3.9	Dilation	73
4.3.10	Screen-Space Shading Implementation	75
4.3.11	Summary Diagram of Texture-Space and Screen-Space Implementations	79
4.3.12	Comparison of Texture-Space and Screen-Space Shaded Results	79
5	The User Study	83
5.1	Test Material	84
5.1.1	Test Scenes	84
5.1.2	Video Production	85
5.1.3	Testing methods	88
5.1.4	User Study Results	91
6	Conclusions	96
6.1	Summary of Results	96
6.2	Avenues for Future Developments	97
6.3	Final Thoughts	98
A	Unity Script Pseudocode	105
B	Rasterization Pseudocode	110
C	Conservative Rasterization Pseudocode	114

Chapter 1

Introduction

1.1 Background and Motivation

The field of computer graphics is one of notable relevance to modern human society. Its contributions can be seen and appreciated in a wide variety of applications ranging from those that are purely entertainment-centric to those strictly intended for industrial or scientific developments.

It is also a field that is characterized by its rapid pace of advancement. As more powerful computing hardware has over time become both more available and more affordable, the number of practical application areas has continued to increase. One example of this is the recent drive towards consumer-targeted Virtual Reality (VR) applications, that aim to provide a more immersive user experience.

VR applications also serve to highlight an aspect of computer graphics that has remained constant through the technological evolution – the continual strive towards an optimal balance between the visual quality of rendered images and the computational requirements placed by the utilized rendering techniques. For VR, interactivity is a vital component in achieving immersion, so the latency between user input and the visual results following from it has to be kept minimal. This is important both for the level of immersion as well as for ensuring user comfort [41]. Although 15 Hz can be seen as a lower limit for the refresh rate of real-time applications [5], current VR devices, including the Oculus Rift and PlayStation VR head-mounted displays support high refresh rates of up to 90 Hz and 120 Hz, respectively, for this reason. This is notably higher than the 60 Hz or 30 Hz update rates which have historically been common for non-VR 3D applications.

VR applications frequently also make use of stereo rendering, in which a 3D scene is rendered individually for both of the user’s eyes. This can

lead to a worst case scenario, where the rendering resource usage is doubled compared to single view rendering, provided that the resolutions per view are the same across techniques.

In traditional rendering techniques, that we will collectively refer to over the course of this thesis as **screen-space shading**, object surface appearance is computed through sampling the 3D scene using a rectangular grid of picture elements (i.e., pixels). In the basic case, the pixel centers are used as sampling locations. The frequency of sampling, and accordingly the number of pixels we use, determines how high frequency visual information we can theoretically sample without information loss. As the end result for the sampling process, we have a value for the light reflected by a surface at each sample point. While the sample points themselves do not have an area, the pixels on physical output device, such as a computer monitor used for the final output, do have a non-zero area leading to a reconstruction step to be necessary. In the simplest case of reconstructing the 2D image of the scene from the sample points, we simply fill the screen pixels with their respective center sample values - a common but inaccurate method which paired with an inadequate sampling frequency can lead to visual error patterns known as *aliasing*.

Aliasing is especially visible when rendering highly reflective glossy surfaces, as their strong dependency on high frequency surface material parameters also leads to high frequency reflection patterns – which in turn require very high sampling rates to avoid information loss. Additionally, popular specular reflection models such as the Cook-Torrance BRDF [9] (covered in Chapter 2) take a statistical approach to the modelling of surface orientation at a micro scale, which is susceptible to further errors due to the way in which the distributions are generated using higher scale surface orientation information. Due to the sensitivity to the reflection parameters (which given movement in a scene can vary for a sampling location over time), glossy reflections often produce reflected light values which vary unnaturally sharply both spatially as well temporally. The temporal artefacts (referred to as *temporal aliasing*) are commonly seen as shimmering or flickering patterns in the rendered images.

Both spatial and temporal aliasing can be mitigated by increasing the sampling frequency (the number of pixels used), as well as by using filtering techniques to band-limit the highest frequency patterns in the 3D scene. As the pixel density is commonly uniform for a single rendered image, the increasing of pixel count can lead to wasteful processing in the areas of image where higher sampling rate is not needed. Additionally, as the sampled light values are tied to their specific pixel samples (they represent the surface appearance at a specific sampling location), it can be problematic if we

wanted to reuse already computed values for temporally subsequent images. The reuse of light values could potentially be used to increase rendering performance by decreasing the temporal sampling rate for surfaces or objects we deemed to have low impact on the overall image quality. In essence we would like to have the flexibility to adjust the sampling rate both based on the spatial frequency of visual details present on an object surface as the temporal rate at which we update the sampling results.

In order to decouple rendering performance from the spatial frequency (the given screen pixel resolution), it is advisable to look into methods for performing the lighting computations in spaces other than screen-space. One such alternative is **object-space lighting** [7], where lighting computations are performed in a 3D model-specific space. For this thesis we chosen and implemented an object-space lighting technique where the lighting calculations are performed per texture element (texel) onto a uniquely parametrized rectangular 2D buffer (commonly referred to as *texture maps* in rendering), leading performance to be relative to this texture map's texel count and not to that of the screen pixels'.

While this decoupling can be interesting in and of itself, true practical value from an alternative lighting technique can be derived only if the visual quality of the rendered images is not significantly decreased. A welcome side effect from using a texture map to store the reflected light of a model surface is that we can use the built-in graphics processing unit (GPU) texture filtering hardware to achieve fast and high quality anti-aliasing. Anti-aliasing is a term commonly used to denote the process of decreasing or eliminating aliasing artefacts by either limiting or removing the high-frequency components of the sampled signal before sampling takes place. In the case of texture filtering, anti-aliasing removes high-frequency patterns which would result in a distorted result for the chosen screen pixel resolution. By performing this manner of filtering only on the final lighting results we can achieve images that are more accurate to reality than images generated by traditional screen-space lighting techniques, where already the light computation input values are interpolated through the use of texture filtering methods. This also means we are able to avoid the earlier mentioned problem associated with the generation of micro-scale surface orientation distributions, further improving the lighting results.

The use of texture filtering for object-space lighting leads to the theoretical expectation that given a high enough texel count, the final image quality achieved by the technique should be close to that of a very high pixel count, screen-space technique where the final screen pixel values are computed as an average of multiple sample values and where the lighting routine utilizes only point sampled (and thus minimally filtered) input values. As part of

this thesis, we conducted a user study where the results suggest that the theoretical expectation is also valid in practice.

1.2 Thesis Scope

In this thesis we set out to implement an object-space lighting technique and evaluate its qualitative results compared to high quality renderings from a traditional screen-space technique. Performance was not a significant consideration for our implementation, and accordingly we do not provide evaluation on matters pertaining to it. This means that we also do not provide analysis or discussion on how object-space lighting should be used for optimal results in a practical scenario, where multiple factors need to be taken into account.

1.3 Structure of the Thesis

In the second chapter of the thesis we cover the aspects of the theoretical fundamentals of 3D computer graphics rendering that were required for our implementation of both the object-space and screen-space shading algorithms. During this chapter we start by covering the basic transformation pipeline, before moving onto the texture filtering operations relevant for our texture-space shading implementation. We close the chapter by describing the Cook-Torrance BRDF which we use as a shading model in the implementation.

The third chapter focuses on mapping the theoretical background to practical rendering application interfaces (APIs), with the Direct3D 12 pipeline [38] used as an example. An introduction to the differences between forward and deferred shading techniques is also given, as the former is used in our screen-space shading algorithm while a logically very similar version of the latter is used for the object-space lighting implementation.

Chapter 4 describes our object-space lighting in detail and showcases example results gained through it. In chapter 5 we explain the design for the user study we conducted and provide our findings. Chapter 6 concludes the thesis by summarizing the overall results, while additionally providing suggestions for future research relating to object-space lighting techniques.

Chapter 2

Theoretical Background



Figure 2.1: The Utah teapot, a classic example of a rendered 3D model. Image by Wikipedia user Dhatfield, distributed under a CC BY-SA 3.0 license.

Taking on any task of significant complexity necessitates the understanding of the fundamentals governing it. Without there being a solid foundation laid beforehand, effort could be wasted and the eventual end results compromised. It is for this reason that we take the time to diligently cover and describe in this chapter the theoretical fundamentals of computer graphics to the extent that is relevant for our implementation of an object-space lighting algorithm.

The rendering process of 3D computer graphics utilizes the language of

mathematics to describe the objects and information that are then used to produce the desired output image. As the world and the objects within it that are to be rendered are three dimensional (3D) entities, while the output devices that are used to view the images are two dimensional (2D) displays, it is clear that we need a way to perform a mapping from 3D to 2D (see figure 2.2).

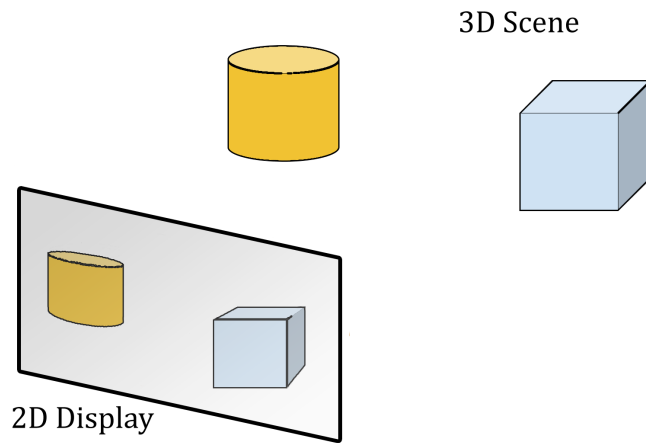


Figure 2.2: The 3D objects have to be mapped onto a 2D display.

To this end, an especially important field of mathematics for rendering is that of linear algebra due to it providing the means to handle calculations involving multi-dimensional variables in a simple form. Using the vector and matrix mathematics linear algebra provides us, we are able to take a surface point on a 3D object and through multiple matrix transformations project and finally color it on its appropriate position on the display screen. All of these steps put together are commonly referred to as the *graphics pipeline* or *rendering pipeline*. An example of a possible end result is given in figure 2.1.

On a high level we thus have two major tasks that need to be performed. The first one is to determine the object surface points that are visible on a viewer's field of vision, represented as the *projection plane*. From this point forward we will refer to this viewer as the *virtual camera*. We will use the term *visibility determination* for this first task.

The second task is then to calculate the color the given surface point should have based on multiple parameters. These include its location, orientation and surface properties (normally referred to as *material properties*), the position of the virtual camera and the incoming *irradiance* (i.e., the total incoming radiant flux per the surface point's area, discussed in section 2.2.1).

Using parameters such as these as the input to a *reflectance function* we are able to calculate the outgoing *radiance* reflected from the surface point towards the virtual camera. This determines the appearance of the visible object surfaces. We will refer to this task by the term *shading*.

Throughout the thesis, we will use the term "3D scene" or simply "scene" to refer to the collection of all of the objects, light sources and the cameras that participate in the rendering process of a 2D image. A more detailed discussion on both of the visibility and shading tasks is given next.

2.1 Visibility Determination

The visibility of a surface point is determined by multiple factors that we divide in this thesis into three conceptual categories. The first category's factors include the position and orientation of the 3D model the surface point belongs to, so in effect its positional attributes independent of any other objects in a scene. The second category is then the possible occlusion caused by other objects in the scene. Intuitively it is clear that a part of an opaque object's surface, positioned and oriented appropriately, could partially or entirely obstruct the surface of another object from being visible to the virtual camera. The third category consists of the properties of the camera, e.g., how wide it's horizontal and vertical fields of view are, how near and far it is able to see, as well as the position and orientation of it.

2.1.1 Object Geometry Representation

As visibility is in its essence a multi-faceted problem, we need to be able to relate the locations of the 3D objects and the camera in the same context. In mathematical terms this means we have to find a way to represent the objects and the camera in the same *coordinate system*. In the common case the 3D objects are represented as *3D models*, which have been produced using a 3D modelling software, such as the 3ds Max and Maya applications from Autodesk, Inc. [6]. The models are usually composed from multiple subparts, i.e., *meshes*, which themselves are built from geometric primitives. The geometric primitives most commonly in use are triangles, but other options such as quads are also available. In addition to the geometric data, 3D models may also contain other surface property information stored in *texture maps*. An example of a 3D model is given in figure 2.3. It should be noted that although meshes are the most common 3D modeling technique, other methods such as constructive solid geometry and point cloud representation are also used.

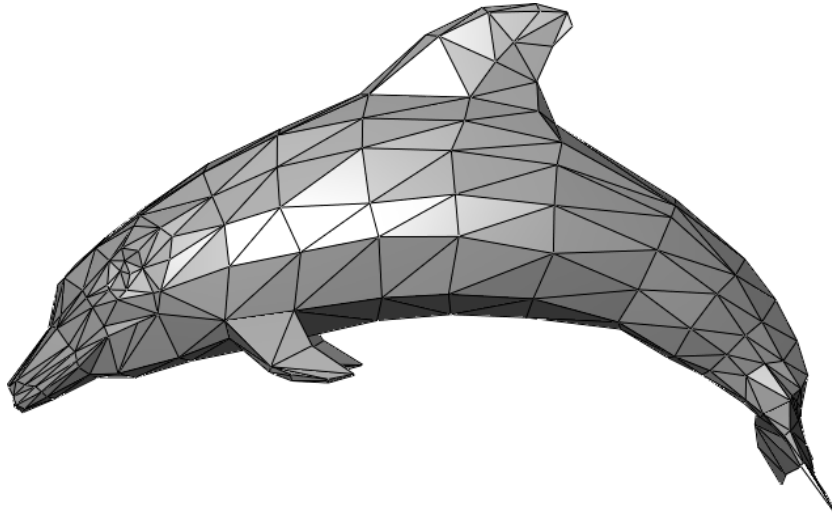


Figure 2.3: A rendering of a 3D dolphin model where the triangle edges have been rendered in black to more clearly show the primitive structure. The triangles are defined by the endpoints of their edges, i.e., the *vertices*.

In the common case where triangles are used as the primitives, each of them is formed by three vertices. While in the simple mathematical definition we would be mostly interested in the positions of these vertices, their usage in computer graphics is notably more general, with them being varied data structures in and of themselves. For photorealistic imaging, the aim of 3D modelling is to produce accurate, while also computationally efficient representations of the underlying real-world shapes and materials the models correspond to. For this reason properties such as vertex color, surface normal direction and texture map coordinates are additionally stored inside vertices. This information is then used as inputs for the shading tasks of the rendering pipeline. More detail on "per-vertex" data is given in sections 2.2.3 and 3.1.1.

2.1.2 Transforming from Object-Local to View Space

Returning back to the production of 3D models, for the ease of creation the models are each defined in their own, local, coordinate systems. Now, when the models are imported to a 3D scene for rendering they would by default all have the same origin and their coordinate axes would also be oriented the same. Depending on the possible scaling factors and positional offsets from the origin for the models, the models would be likely to be rendered

in a formation resembling a matryoshka doll with its nested structure, each centered (or very close to) about a single origin (see figure 2.4). Clearly this is rarely the desired outcome.

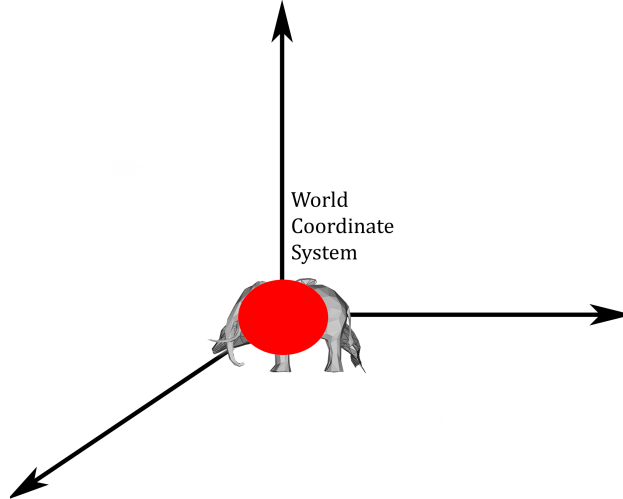


Figure 2.4: Here the objects have been positioned and oriented by incorrectly assuming their vertices to have been defined directly in the world coordinate system (depicted by the large axis arrows).

What is usually done is to specify a world coordinate system which acts as a universal coordinate system relative to which all of the objects in the scene are described. The individual objects should each have three variables defined: the translation or displacement vector, the rotation or orientation variable (typically either a 3D vector or a 4D *quaternion* [5, p. 72]) and the scaling factor vector. These represent the object's local coordinate system in the world coordinate system's context. Knowing these three variables and the origin and the coordinate axes of the world coordinate system, we can in theory place objects in the scene properly positioned, oriented and scaled. In practice, we would be well-served to utilize linear algebra and so compose matrices for translation, rotation and scaling.

While rotation and scaling are linear transformations, meaning that they can be represented as matrices operating on 3D vectors in a straightforward manner, translation is an affine transformation which requires the use of homogeneous coordinates. In practice this means the addition of a fourth

component, the w coordinate (in addition to the usual x , y and z coordinates) to our vectors.

As matrix multiplication corresponds to the composition of transformations, we will go a step further and combine all of the aforementioned transformations into a single matrix. This is usually called the *world* or *model transformation*. It is worth making the distinction clear that to transform an object's surface representation, i.e., the triangles it is formed of, we need to transform each of the triangle vertices to their world coordinates, as simply transforming an object's local origin would not help us much in rendering the object in its intended location.

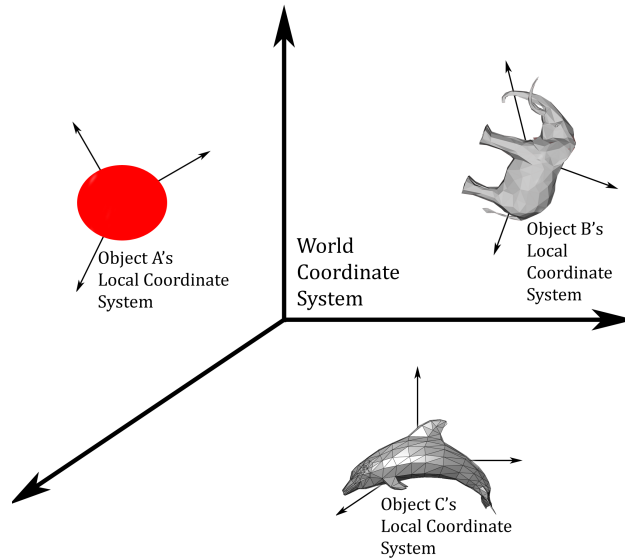


Figure 2.5: Here the objects have been positioned and oriented in the scene by transforming their vertices from the local coordinate systems (depicted by the object-specific small axis arrows) to the world coordinate system (depicted by the big axis arrows). The vertices are correctly assumed to have originally been declared relative to the local coordinate systems which are object-specific.

Although we are now able to represent all of the scene's objects' locations, orientations and scales using a single coordinate system, for visibility determination we still need to understand their relation to the camera's position and orientation. This is done by first describing the camera's position and orientation in the world coordinate system's context again using a compo-

sition matrix, resulting in the camera's world matrix. As external objects appear to move to the opposite direction that the viewer is moving, this matrix will have to be inverted to achieve the proper end result. This end result is referred to as the *view transformation*.

By using the view transformation to perform matrix-vector multiplications on world space vertex position vectors, these vertex positions are transformed into view space coordinates. Typically, in an effort to reduce the number of computations needed, we would compute a model-specific world-view transformation, which would then be used for any of the model's vertices to take them straight from the object-local space into view space, the virtual camera's coordinate system. Many times however, we do need the world space coordinate's for our subsequent shading calculations so a separate world transformation matrix is usually stored as well.

2.1.3 Perspective Projection

At this point we are able to describe a 3D model's surface in camera-relative coordinates. As previously noted, for display the primitives need to be projected from 3D to a 2D *projection plane*, also called the *projection window*. This plane can be thought of as the view space representation of the display screen. The algorithms to accomplish the desired projection fall into two main categories: **rasterization** and **ray-tracing**. In rasterization we start from a 3D model's surface point's view space location and compute its corresponding location on the projection window. In ray-tracing, on the other hand, we start from knowing the location on the projection window and finding out which surface point in the scene corresponds to it. Expressed differently, the algorithms progress conceptually in the reverse order from each other. This has several implications of which the most important are those relating to the speed of rendering as well as the complexity of implementing shading algorithms.

In **ray-tracing** the surface points for each partial element of the projection window are calculated through ray-object intersections, with the rationale being that the closest intersected surface point is the one "seen" by that ray. The ray is created by having it originate from the camera position (i.e., the origin of view space) and setting its direction so that it passes through the center of a given projection window element. These elements can be thought of as a sampling grid, where each of them is commonly referred to as a picture element or a *pixel*. As both the origin and the direction of the ray is known, a parametric representation of it can be formed and used to perform intersection with various geometric entities. Usually these are either triangles or planes, but ray-tracing enables the use of many other shapes

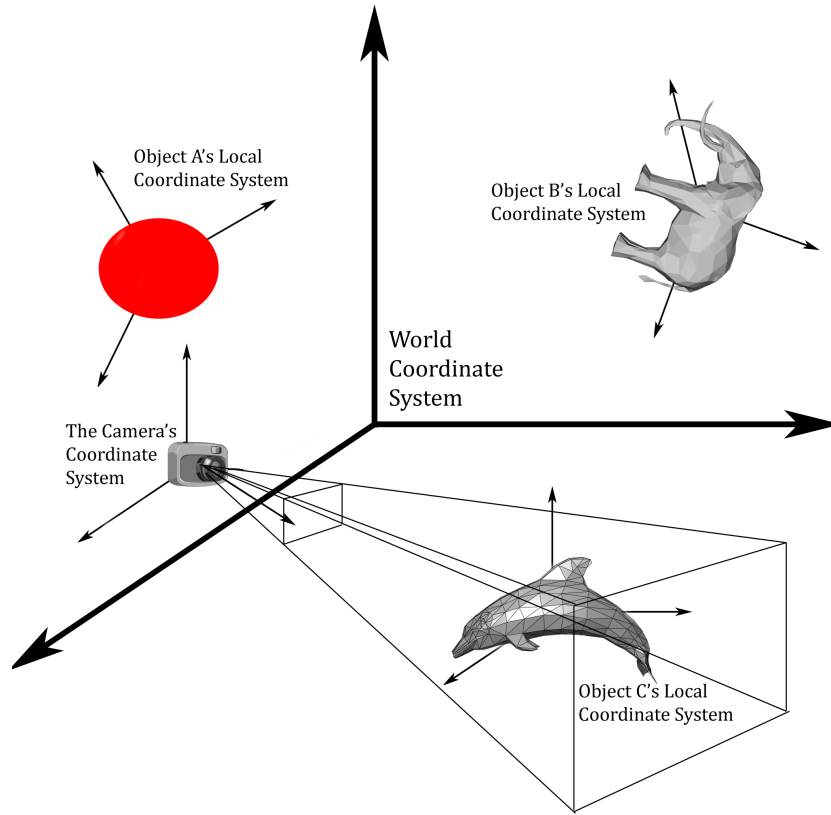


Figure 2.6: The 3D scene in world space, with the virtual camera added to the scene. The pyramid-like shape with a truncated top is the *view frustum* for *perspective projection*. Note how the virtual camera has its own coordinate system as depicted by the axis arrows originating from the camera.

such as spheres and cylinders. These cannot be used as is in rasterization without first deriving a corresponding triangle mesh, due to modern *graphics processing units* (i.e., *GPUs*) being optimized for triangle processing.

Although many acceleration structures have been devised for ray-tracing, the ray-object intersection process remains a computationally taxing one [8], making ray-tracing unsuitable for the majority of use cases necessitating real-time rendering speeds. This is also why ray-tracing will not be discussed any further in this thesis, with the focus being on rendering utilizing rasterization.

In **rasterization**, coming from view space we need a method to map the view space coordinates onto the projection window. As the projection window can be thought of as a canvas that the images are painted onto, as well as the virtual representation of the display screen, we know that it has to

be of finite size. This size is determined by the horizontal and vertical *fields of view* of the virtual camera, giving the view volume boundaries in both width and height. Additionally, to limit the number of rendered surfaces and for enabling us to perform depth comparisons while using a finite precision, the view volume is also limited in its depth. This volume is commonly referred to as the *view frustum* (see figure 2.6).

The situation calls for the use of a projection transformation – and since we use linear algebra, a *projection matrix*. There is a variety of different projections one could use to build the needed projection matrix from, but in 3D imaging the most common one is the *perspective projection*. This transformation introduces to the rendered image a foreshortening effect similar to how the human eye works with objects further away appearing to be smaller in size. In practice this effect is achieved by dividing the view space x and y coordinates by the z coordinate of a surface position. This operation is known as the *perspective divide*. As this division cannot be represented using a matrix-vector multiplication, it has to be performed in a separate step after the multiplication. The view frustum for perspective projections resembles a pyramid shape with its top cut off, due to the intersection with the near clipping plane. A 2D diagram of the view frustum for perspective projection is depicted in figure 2.7.

It is important to realize that vital properties for 2D display, like a display screen’s physical pixel resolution or the system-provided rendering window size can vary from one hardware/software configuration to another. This means that it is not possible to define a single projection transformation from 3D into the actual 2D screen pixel coordinates which would hold in the general case.

Thus, we need an intermediate coordinate system called the *NDC (Normalized Device Coordinates)* space, in which the x , y and z coordinates are mapped to a $[-1, 1]$ range (in *OpenGL* [47], that is – *DirectX* [36] uses the $[0, 1]$ range for the z coordinate). This is the coordinate system the vertices reside in after the projection transformation. Additionally, the fact that the view frustum is of finite size opens up avenues for performance optimization, as the primitives situated outside of it can be discarded entirely while those intersecting one or more of its total of six boundary planes can be clipped against them.

Keeping in mind the NDC ranges, it would be simple to clip against the range boundaries, but a further optimization step means that we perform the clipping before the perspective divide and thus in an intermediate space before the perspective transformation is complete. This space is called the *clip space* corresponding to its functional role. In clip space we effectively compare the absolute values of the x , y and z coordinates after the projection

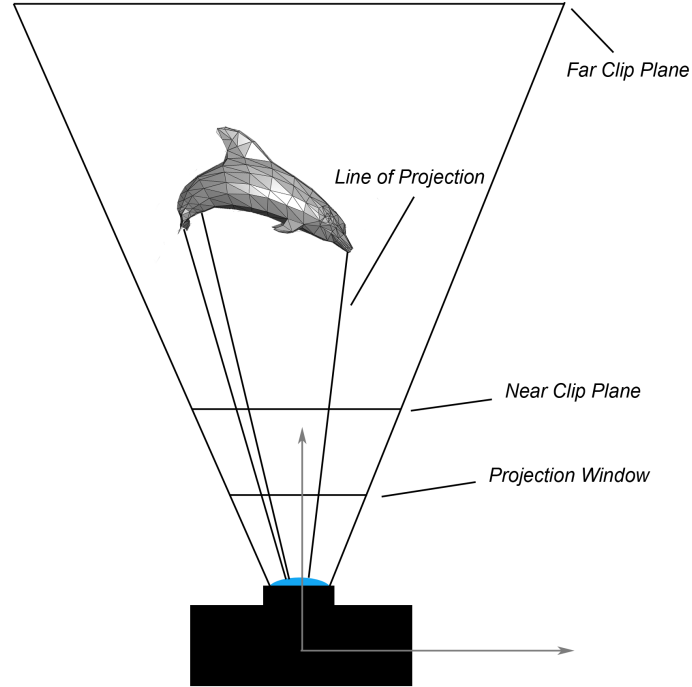


Figure 2.7: The dolphin model has been transformed into the view space, ready to be projected. The relations between the different clip planes and the projection plane/window are also depicted.

matrix multiplication against the absolute value of the vertex' depth value, stored in the homogeneous w coordinate. The comparison result is the same as if we had clipped against the NDC boundaries except slightly more performant done in this order. After the clipping has been done, the perspective projection can be completed by finally performing the perspective divide: the division of the x , y and z values by w .

2.1.4 Sampling

The only step left now is the *viewport transformation* which takes the NDC coordinates into *screen space*, i.e. the actual screen pixel coordinates. This is done by first transforming the coordinates into the range $[0, 1]$ and then scaling them by the viewport's pixel dimensions. As we now have the screen

positions of the vertices we could proceed to drawing (or *filling* as its known in computer graphics) the insides of the triangle they form. This however would leave an important part of visibility determination – *occlusion*, unaddressed.

For this we need to compare the screen space z coordinate values (in the $[0, 1]$ range) to the possibly already filled pixel's z values. In other words we are comparing the depths which is why the buffer storing these values for each drawn pixel is called the *depth buffer* (also commonly referred to as the z -buffer). Following from the fact that the values are stored per screen space pixel, the dimensions of the depth buffer have to match those of the *back buffer* (where the drawn pixel colors are stored). In a typical usage scenario, the depth values are at first initialized to the maximum possible value of 1.0 (the far clip plane). After this we only update those back buffer pixels whose corresponding depth buffer value is greater than the currently drawn *triangle fragment's* depth value. This in effect means that only the surfaces nearest to the camera per back buffer pixel are visible, as we desired. An example of a triangle filled into a pixel grid can be seen in figure 2.8.

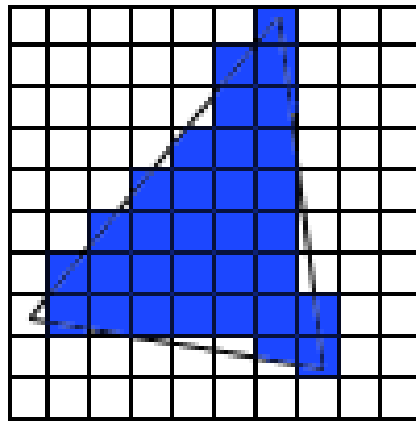


Figure 2.8: A solid color triangle rasterized onto a screen pixel grid. The sampling location for rasterization here is the center of the pixel cell. Note how this form of sampling results in *aliasing* artefacts on the triangle's edges.

From the figure it can clearly be seen that the reproduced triangle exhibits jagged edges unlike the mathematical definition of a triangle. This is due to the sampling process in which the triangle has to be presented as a finite collection of discrete pixels on the back buffer. The error in reproduction makes it seem that (solely based on it) the original shape could have plausibly been something else than a triangle. This confusion between the original and reproduced images, or more generally *signals*, is in the field of signal processing referred to as *aliasing*, as the identity of the original signal

can be confused with another separate signal. The possible presence and magnitude of aliasing therefore depends on our ability to reproduce the original signal without information loss, so that no misidentification can occur. This depends on two aspects: the frequency of the original signal and the frequency of the reproduction.

Before explaining this further, it is worth underlining that one of the key aims of computer graphics is to achieve *photo realism*, so as to be able to render virtual worlds that would appear as real to us as the real world viewed through our eyes does. Now, while the real world is continuous (at least to the extent relevant to this thesis) computers operate on discrete information. It follows that even if we used a continuous mathematical model of a real world phenomena a discretizing (or "sampling") step would still need to be performed to quantize the derived values. While sampling can be thought of as a simplifying reproduction of the original signal, it does not necessarily introduce information loss as long as the sampling frequency is high enough compared to the frequency of the original signal. The answer as to what "*high enough*" means is given by the *Nyquist frequency*. The Nyquist frequency is defined as a sampling frequency that is twice that of the highest frequency present in the original signal. Provided that our sampling frequency is equal to or greater than that of the Nyquist frequency, it is theoretically possible to reconstruct the original signal from the sampled values [5, p. 119].

While a solid theoretical basis, the Nyquist frequency does not help in many practical cases in computer graphics, as no upper limit exists for, e.g., the frequency of change for the color values a surface assumes between adjacent points. The end result is that we sample a surface (and the material properties needed for shading) which, while correct for that particular sampled point without an area, is incorrect when used as is for the projection window pixel which does have a non-zero footprint on the surface.

The situation can perhaps more easily be understood by relating the way in which our virtual camera functions compared to real-life photography. In real photography there exists a class of devices very similar to our virtual camera called the *pinhole camera*. Without delving too deep in to the particularities of the pinhole camera, the key similarity to our virtual camera is that they both project visible surface points in the external world onto unique positions on a projection plane. To be precise, while the virtual camera maps exactly one external point onto its corresponding projection plane point, the real pinhole camera collects light from a small angle around the to-surface-direction vector, introducing a degree of blur into the produced image. In effect this serves as a low-pass filter, removing high frequency detail from the image leading to decreased aliasing. The cause is that no real-world pinhole camera actually has an infinitely small aperture size (i.e.,

the pinhole).

A more important difference however is that a real-world pinhole camera's image sensors are physical entities with a non-zero area, meaning that they can sample light for the entire area they represent. This leads to an image sensor forming a final sample value for the incoming light that is an average of all the light entering the sensor. This is in contrast with the virtual camera's "sensors" which are infinitely small, singular points. When the final output image is reconstructed from these point samples, projection window's pixel cells are filled in by extending these sparsely laid out sample points' values to each cover the entirety of their corresponding pixel. This causes aliasing partly due to the sampling method which can be inadequate depending on the sampling and source signal frequencies, but also due to the chosen reconstruction method.

The problem can be made less apparent by increasing sampling frequency - in this case the number of pixel samples taken. If the number of pixel samples we take is higher than the number of pixels in the final output image, we can form the final pixel values as averages from the pixel samples. This is referred to as *supersampling*. The higher sampling frequency ensures that less information is lost in the sampling process, while the averaging process (*downsampling*) limits the highest frequencies present in the samples used for the final pixel values. However, as noted the highest frequency present in the original signal in the general case is unbounded, so supersampling and the accompanying downsampling do not solve the problem completely.

In practice, low-pass filtering methods are usually employed to limit the maximum signal frequencies present in the input data we use for coloring the screen pixels. As the storing of this data is commonly performed using *texture maps*, modern graphics hardware support a variety of texture filtering methods (discussed in section 2.2.4) to minimize aliasing in a performance efficient manner.

Other, common anti-aliasing methods in popular use today include *multisample anti-aliasing* (MSAA) [33], a form of geometry anti-aliasing which performs supersampling only for pixels inhabited by fragments from multiple triangles, and various *post-process* techniques such as SMAA [23] and FXAA [30] which operate only on the back buffer's pixels after all of the shading computations have already been performed.

2.2 Shading

After performing the *world*, *view*, *projection* and *viewport transformations* to take a triangle's vertices from *object-local space* coordinates through *clip*

space coordinates into *window* coordinates, we are able to fill the triangle shape into the screen pixel grid. While it is possible to perform this filling-in simply by using a single uniform color value for the entire triangle, this would hardly result in a realistic appearance. Thus, there is a need for a more sophisticated method for coloring, or *shading*, the pixels. From our experience of the real world we know that the surface appearance of objects depends on many different factors, including the direction and intensity of *light sources* emitting light rays to the environment as well as the properties of the objects themselves.

As touched upon already in section 2.1.1, we store some of these object properties on a per-vertex level. The most obvious one is of course the vector for the positional coordinates of a vertex, but for shading many other details such as the facing direction of a surface are also important to know. These per-vertex values are *linearly interpolated* using the *barycentric coordinates* of a given point inside a triangle to then gain approximations for the values as they would be for that coordinate location. This interpolation is discussed further in chapter 4 where we delve into to the details of our object-space lighting implementation. We will now move on to the theoretical basis of surface appearance.

2.2.1 The RGB Model and Radiometric Units

The matter of determining the color of a surface point is philosophically speaking a case two sides. On one hand, the visual appearance of an object is related to the light that is reflected towards a viewer. Light can be described as consisting of particles referred to as photons. Each of these photons holds a certain amount of energy, and so it is also meaningful to measure the amount of energy of the light rays an emissive surface emits. If an object emits photons, it is a light source. In addition to emitting photons, objects can also reflect photons and thus light, which is how humans are able to perceive surface appearance. We can also consider the flow of photons and the energy associated with them in relation to the passage of time. This flow of energy per a duration of time (i.e., a second) – in other words the "power", is called the radiant flux. [5]

While the object surfaces and the photons reflected from them can be thought of as what physically exists, the other side of the coin is what humans are able to perceive of them. For this end it is useful to examine light from another perspective. In addition to its particle nature, light can also be described as having wave-like properties. Evolution has resulted in humans possessing a complex formation of the eye, the optic nerve pathways and the brain's visual cortex to enable visual perception. All of these components put

together are commonly referred to as the *human visual system* (HVS). In the context of the HVS, the eye serves as an optical sensor through the use of its rear surface called the *retina*, which is formed by several neuron layers. The central part of the retina, where the eye's lens focuses the incoming light rays is called the fovea. For our purposes the most important of these neurons are the light sensitive rod and cone cells. The rod cells vastly outnumber the cone cells (by 100 million to 6.5 million) and are mostly situated in the regions away from the fovea (leading them to be vital for peripheral vision). They are unable to sense differences in the wavelengths of the light rays, and their visual resolution is low, but they are very sensitive to even low intensity light – resulting in rod cells being mostly responsible for vision in low-light conditions.

Unlike the rod cells, the cone cells are mostly situated in the foveal region of the retina and have a higher visual resolution due to manner in which neural signals are collected from them. The different wavelengths sensed by the rod cells are processed by the *human visual system* (HVS) resulting in the perception of different color hues. It is worth noting that only a limited range of wavelengths – from approximately 380nm to 780nm, i.e., the *visible spectrum* can be seen by the eye [5]. Neighboring the visible spectrum on the low end side is ultraviolet radiation while on the high side the rays are categorized as infrared, or heat radiation. For computer graphics we usually are mainly interested in what a human eye can see, so we limit ourselves to only the visible spectrum of light rays.

The topic of color perception is a complex one, and it is important to understand that color perception does not solely depend on the light's wavelength but also the amplitude of the wave. This gives colors their perceived brightness, which can result in colors of identical hues being perceived differently. Although it would be possible to compute the colors based on an emitted (or reflected) radiant flux when the associated *spectral distribution* for the flux is known, this degree of physical accuracy is rarely sought. Instead, a simplified *RGB color model* is popularly used in computer graphics. In the RGB model colors are represented as a mix of the three primary colors of the additive color theory: red, green and blue. The effect of brightness on perceived colors has been incorporated into the RGB values so that we can define any of the entire color gamuts's individual colors as just as the ratios of the three primary colors.

The coverage of the RGB model's gamut in relation to the HVS's gamut is decided by the choice of which colors the red, green and blue primaries exactly map to, while the resolution or the ability to discern between the values inside the gamut is governed by the precision (the number of bits) we use to represent the RGB components.

We now know how to quantify the traveling light rays, in relation to human perception and computer graphics, in a meaningful and efficient way so the next step is to examine how light interacts with objects. Provided that the object in question's surface is a non-self emitting one, the only way for humans to see it is through the surface reflecting visible light from external light sources to the viewer's eye. Self-emitting surfaces can of course also be seen by humans, but we will not discuss the rendering techniques related to those cases.

Another source of complications is how in the real world the incoming light to a surface could arrive after potentially multiple bounces on other surfaces in its environment. This category of surface illumination is traditionally referred to in computer graphics terms as global illumination. However, we will also limit the discussion so that we only consider light arriving directly from a light source to a surface, i.e., the local illumination. Following this, if we were to have only one light source, with no area, the theoretical maximum for the radiant flux reflected by a surface area would be determined by the following equation:

$$L_o = E_i * \max(\mathbf{n} \cdot \mathbf{l}, 0) \quad (2.1)$$

where L_o and E_i are measures of *radiance* and *irradiance*, respectively. In computer graphics it is common to express both as RGB vector quantities. The \mathbf{n} , the normal vector, signifies facing direction of the surface while \mathbf{l} is the to-light source vector. Both of these vectors are unit length. The dot product between them is clamped to result in 0 at the minimum, as negative values would indicate that the surface is being lit from behind its normal's direction.

Irradiance is the density of the radiant flux originating from all incoming directions passing through a unit surface area. In essence it is the area density of the light power and it governs how much a surface is being illuminated by a light source. For a given amount of light power, the irradiance decreases as the area onto which the light spreads increases and vice versa. If instead of examining radiant flux passing a unit area we think of the flux arriving onto a surface from a given originating direction, i.e., the radiant flux per unit solid angle we are describing what is called the flux's radiant intensity.

Combining these two concepts of irradiance and radiant intensity we get the density of the radiant flux with regard to both a single incoming direction and the surface area the flux passes through: the area density of the flux per unit solid angle. This measure is the **radiance** to a surface point.

Equation 2.1 captures the fact that as the difference in direction between the surface normal \mathbf{n} and the to-light vector \mathbf{l} increases, the amount of energy

received by the surface area decreases. This is simply due to the density of the photons hitting a surface area, i.e. irradiance from the radiant flux, decreasing as the elevation angle between \mathbf{n} and \mathbf{l} increases. It has to be noted that irradiance from radially emitting light sources is inversely proportional to the square of the distance between the light source and the illuminated surface. This means that while a light source (such as a *directional light*) emitting light uniformly in a single direction does not experience any distance based *attenuation* on the irradiance it causes, other lights source types like *point* and *spot lights* do experience it. This is why an attenuation term is normally used with these types to more accurately mimic the real world. An example of a distance based attenuation function is given by the following equation:

$$f_{att}(d) = \frac{1}{k_q d^2 + k_l d + k_c} \quad (2.2)$$

where d is the distance between the light source and the surface, and k_q , k_l and k_c are the coefficients for the quadratic, linear and constant terms, respectively. These coefficients can be used to subtly alter the lighting of a scene according to artistic goals. Distance based attenuation only affects the irradiance to a surface but not the incoming or outgoing radiance, explaining also why humans do not perceive the brightness of objects to diminish or change the further away they are.

2.2.2 Physically-Based Rendering

We proceed by noting that a light ray hitting a surface results in one of two possible outcomes: the light ray being reflected away from the surface (*surface reflection*) or it being transmitted through refraction into the surface. The first case can be seen as purely an alteration to the direction the light ray travels, while the second case involves both a change to the direction as well as to the energy of the light ray. The change in a light ray's energy is due to the distance traveled inside a medium, where interactions with its particles lead to a portion of the light's energy being absorbed into the medium [21].

It is possible for a material to completely absorb the energy of a light ray through this internal travel, leading to no light re-emerging outside. Conversely, if a light ray is able to re-emerge it may do so as multiple "sub-rays" scattered into differing directions with a fraction of the original ray's energy. The changes in direction are caused by a composite material consisting of several material with differing indices of refraction [21]. This form of reflection is often categorized as *body reflectance* or *subsurface scattering*. In the real world the exit locations are generally not the same as the location

where the light first made contact with the surface, but this detail is often avoided in rendering unless it is deemed to have a high impact on the surface appearance – as is the case when viewing materials exhibiting sub-surface scattering, like the human skin, from a close distance [21].

A fact that is not avoided on the other hand is the effect of absorption on the appearance, as it not only affects the brightness of the re-emerging light but can also do so on a per-wavelength basis. This is the phenomenon that causes the majority of the materials we encounter in the real world, categorized as *dielectrics* or *insulators*, to have a characteristic color to them. It is worth underlining that as this color is the result of body or **diffuse reflection**, materials that do not exhibit diffuse reflection (i.e., *conductors*, metals) do not have the same kind of characteristic color to them. For conductors the appearance of a color can only derive from light source color or through the possible bounces light rays are subjected through the impurities immediately on top of the conductor’s surface.

During recent years, a collection of rendering techniques commonly grouped under the name of *physically-based rendering* (PBR) have gained popularity among real-time rendering implementations, due to the increases in computing power and the high visual quality PBR is able to provide [39]. In simple terms PBR stands for techniques in which the shading of a surface is done utilizing lighting models and surface descriptions that aim to closely approximate the physical reality. Previously used, popular shading models such as the *Phong model* contained simplifications that in places broke the laws of physics. The most glaring of these was going against one of the key principles in physics – that of energy conservation, where no energy can simply appear from or disappear into nothing. Following the principle of energy conservation, we then know that the amount of outgoing light reflected from a surface can never exceed the amount of incoming light, as could be deduced from equation 2.1 earlier.

As previously noted, we also know that the outgoing light can be divided into reflected and transmitted components. Transmission is a complex phenomenon to model in rasterization-based graphics pipelines (due to the scarcity of information during shading from other objects in the scene), and has the most effect on the appearance of highly transparent objects. This is why a simplification is commonly done to forego the modelling of true transparency and instead to just focus on the body reflection aspect of transmission. Body reflection is depicted in figure 2.9 as the rays bouncing inside the surface before emerging in (seemingly) random directions. This is what we mean by the term *diffuse reflection*.

From figure 2.9 we can also see the surface reflection part of light-material interaction. While the diffuse reflection can be thought of as a soft reflec-

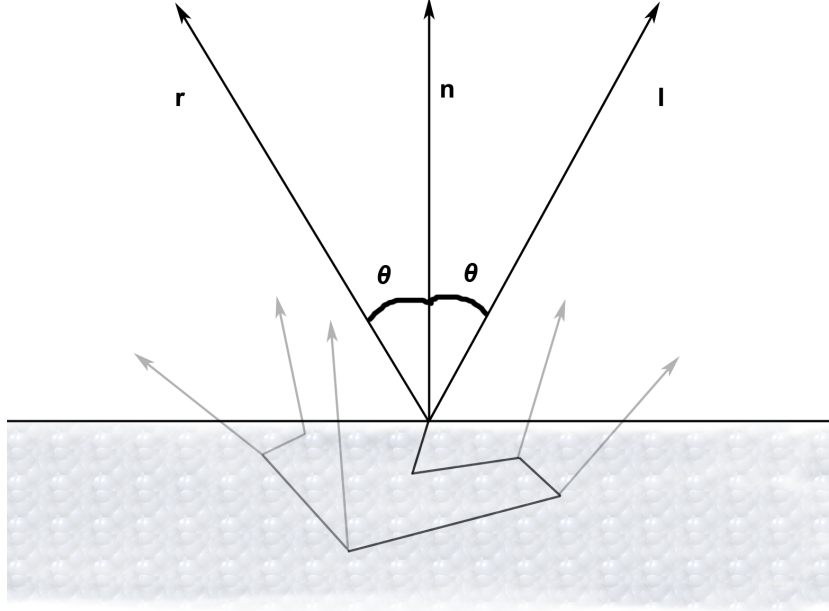


Figure 2.9: The surface and body reflections are illustrated in this image. The vectors \mathbf{n} , \mathbf{l} and \mathbf{r} denote the *surface normal*, *to-light* and *reflection* vectors respectively. While the surface reflection occurs always "on the spot" to the reflection vector \mathbf{r} 's direction, the body reflection can have multiple bounces inside the surface and thus multiple emergence points and directions. We will not cover lighting models that take into account the distance travelled inside the surface.

tion with view-direction independence, the surface reflections is highly view-dependent. Expressed in another way, the surface reflections correspond to mirror-like or **specular reflections**. According to the law of reflection, the angle a reflected light ray makes with the surface normal (depicting the orientation of the surface) is equal to the angle between the incoming light ray and the surface normal. For ease of use, a reversed version of the physical light travelling direction is usually used, as depicted in the figure.

The modelling of specular reflections leads to two interesting characteristics. The first one is that the contribution of an incoming ray can not be seen by the virtual camera, unless the view direction \mathbf{v} from it to the surface is equal to that of the reversed reflection direction $-\mathbf{r}$: $\mathbf{v} = -\mathbf{r}$. The second one is that as the specular reflection occurs exactly on the material surface

(as opposed to inside it), there should be no absorption involved in theory. Thus, the amount of energy and – so the brightness of the light should be unaffected, with the color of the reflected light being solely dependent on the incoming light’s color.

On the other hand, diffuse reflection reflects incoming light out to seemingly random directions with a (possibly) perceivable effect on the light color. To model the randomized emergence directions, an idealized concept of a perfectly diffuse surface can be used, where the outgoing, reflected light is evenly distributed over the half-sphere (*hemisphere*) above a surface point. This distribution ensures that the surface appearance is the same for all unobstructed view points, while simultaneously modelling the change to the outgoing light’s color (the distributed rays have been attenuated to keep in line with the conservation of energy). [5, p. 110]

The change to the diffusely reflected light’s color is modelled by an RGB vector representing the amount of absorption an incoming light ray undergoes depending on its spectral power distribution (in RGB model terms, the fraction amount of each primary color the light ray consists of). This RGB vector is traditionally called the *diffuse albedo*, as it expresses the fraction of outgoing light divided by the incoming light for each primary color [5, p. 239]. This is a characteristic property for each material type.

2.2.3 Materials

In reality, object surfaces are not entirely specular or diffuse but instead exhibit a combination of both reflection types. This is why in computer graphics we usually think of materials as having a certain amount of roughness or glossiness to them, describing how diffusely or specularly they reflect light.

Previously in our discussion we have also come to understand a number of other material properties affecting how light interacts with surfaces, aiding us in modelling realistic lighting in rendering. Let us summarize these:

- **Diffuse albedo**
- **Specular reflectivity**
- **Normal**
- **Roughness**

These are among the most common surface properties one will come across in modern computer graphics. Going through the list, the **diffuse**

albedo represents the amount of light a surface reflects per each of the RGB model's primary colors. In practice this RGB vector is commonly augmented into a four component *RGBA* vector, where the *A* component is used to store an additional surface property. Common use cases include storing a value for representing the opacity of a surface (for transparency using blending effects) or for conveying the amount of *ambient occlusion* for a surface (a technique used to simulate *global illumination*).

While the diffuse albedo governs the body reflections of a surface, the **specular reflectivity** parameter serves as one of the inputs we use to determine surface reflections. Every material has a characteristic amount of surface reflection for light striking the surface head-on, i.e., when $\mathbf{l} = -\mathbf{n}$, and this is what the specular reflectivity captures. For dielectric materials this is an RGB vector, where each of the components is equal to the others (so it could effectively be described by just a gray scale value), but for conductors the component values vary slightly. This is an approximation to model how metallic object do have a tinted reflection to them, although theoretically specularly reflected light's color should not be altered as a consequence of the reflection. In the case of dielectric materials the specular reflectivity value is equal to the *F0* or the *Fresnel reflectance at 0 degrees*. We will be examine the Fresnel effect in section 2.2.6.

The **normal** is a 3D vector which denotes the general direction the surface is oriented towards. It is perpendicular with regard to the surface tangent and normalized to unit length. The normal can be defined at different scales of observation depending on intended usage and thus the needed accuracy. Common normal vector scales used in 3D rendering are primitive (e.g., a triangle), vertex and *texel* (from the term *texture element*) scales. These correspond to defining normals in respectively, as *primitive uniforms*, per-vertex data or as the texels of a *normal texture map*. As surface appearance is heavily affected by the surface normal's orientation, the use of high resolution normal data can in the general case lead to more photorealistic results. The drawback of course with the higher data resolution are the potentially adverse effects on memory consumption and computational load.

As a concrete example, the lowest resolution of normal data one would use is the primitive scale, where the surface orientation can be described only once per primitive leading to uniform lighting results over each primitive. This results in stark discontinuities being visible at boundaries between adjacent primitives. Utilizing per-vertex normals enables the calculation of more precise surface orientation by interpolating the vertex normals for each primitive to gain surface normal values for points lying inside the primitives. With normal texture maps we store a large number of normal samples over a primitive's area and then use the texture coordinates for a particular

point we are interested in to index into and interpolate between these values. As the resolution we store the normal samples greatly exceeds that of only storing normals per vertex, normal maps allow us to gain significantly more precise approximation for surface orientation. Texturing will be discussed in more detail in section 2.2.4 and implementation details relating to the use of normal map will be examined in section 4.3.8.

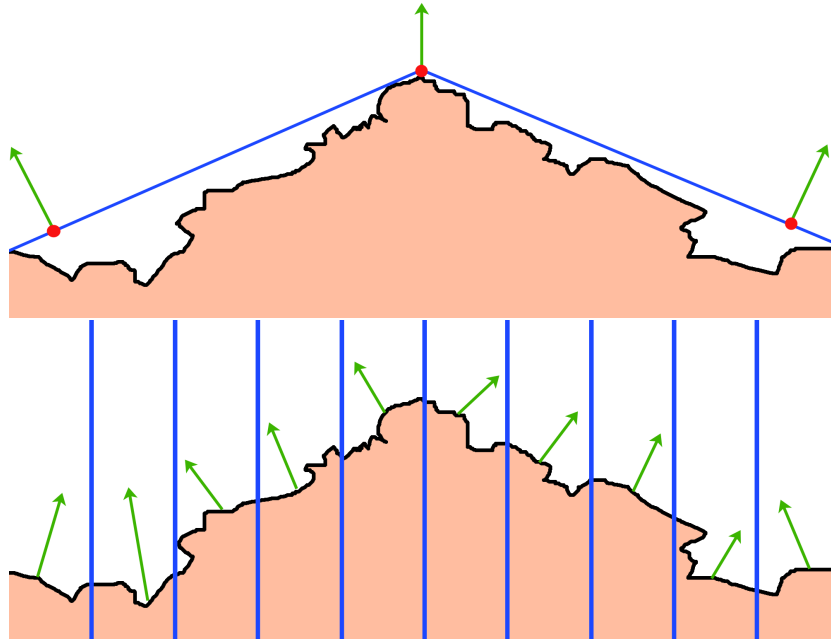


Figure 2.10: Illustration of vertex normals and a normal texture map.

In figure 2.10, we have illustrations for two of the different scales of surface normal representation, the per-vertex normals (above) and the normal maps (below). The rough uneven surfaces depicted by the black outline represent the real-world target we are modelling. In the upper image the blue lines depict the primitive surfaces, with the red dots being the triangle vertices where we store the surface orientation information. Orientation for points on the blue lines between the red dots have to be derived from these through interpolation. In the lower image, the blue lines depict normal map texel boundaries. The resolution of information in this case is notably higher when compared to the per-vertex equivalent. While the use of normal texture maps allows us to greatly enhance the accuracy of geometry detail modelling without prohibitively high resource cost, we can notice that even then we can fail to capture very miniscule details present in the real-world surface. To model these micro-level details we take a statistical approach and utilize BRDF models with **roughness** values as inputs to approximate

these variations. An example BRDF model, the *Cook-Torrance BRDF* will be examined in section 2.2.6.

2.2.4 Texture Maps

Material properties like normal and diffuse albedos are usually stored in 2D arrays which in rendering we call **texture maps**. As previously said, the primitives we render are comprised of vertex structures which among many other properties include texture coordinates, also known as the **UV coordinates**. While the dimensions, expressed in texture elements (**texels**), of a texture map may vary from one map to another, the UV coordinates use normalized values in the $[0, 1]$ range making them resolution independent.

It is still possible for an application to specify UV coordinates outside the $[0, 1]$ range in which case the appropriate behaviour is decided by what is called the **texture addressing mode**. The possible modes are **wrap**, **mirror**, **clamp** and *border color* in the terminology used by Direct3D [34] – the exact names vary between different graphics rendering interfaces. *Wrap* drops the integer part of any UV coordinates outside the normalized range, while the *mirror* mode mirrors the UV coordinates at integer boundaries. *Clamp* limits the UVs to the $[0, 1]$ range with any lower or higher coordinate values mapping to 0 or 1, respectively. In effect this means that outer values are mapped to the texture map boundaries. The *border color* mode allows the developer to simply specify a value that should be used in cases where the UVs fall outside of boundaries. We will continue the discussion with assuming that the UV coordinates we are using are in the normalized range.

During the shading part of the rendering pipeline, when we desire to use property values stored in a texture map, a given texture map texel can be indexed into and its value retrieved by multiplying the U and V coordinates by the texture map width and height dimensions, respectively. In the simplest form of sampling, the **nearest neighbor filtering** (i.e., **point sampling**) the texture indices (which need to be integers) are calculated simply by truncating the floating point indices resulting from the multiplication. As a concrete example of texture sampling, to sample the diffuse albedo value during shading from a diffuse map one would use the interpolated UV values from the rasterization stage for a given inside point of a primitive to create the truncated diffuse map indices. After this we simply would index into the texture map using these indices.

Depending on the projected size of a primitive, the UV coordinates of its vertices and the texture map's texel resolution, there are often cases where a texel cell's projection onto the screen does not exactly match a screen pixel cell in size or alignment. The texel cell might cover a larger or smaller

area of the projection window compared to a screen pixel cell's coverage and the texel cell's center might be offset in relation to a pixel cell's center. Indeed, the discrepancies in cell sizes are so common that they have their own categorization.

In the case of a texel cell being larger than a screen pixel cell, we speak of **magnification**. It is easy to picture cases where, for example, resulting from the projection transformation, a primitive has been projected to so as to cover a very large screen area size. In such a case the magnification can be so significant that a large number screen pixel cells are covered by a single texel cell. In these cases simply using point sampling would lead to a discontinuous, "blocky", result due to the changes between values retrieved from adjacent texels exhibiting more visible step changes because each texel cell covers a larger area on the screen. This is a form of aliasing. What is done instead is interpolation between sample values from neighboring texels both horizontally and vertically. This process of performing the interpolation in two dimensions is called **bilinear filtering**. [5, p. 159]

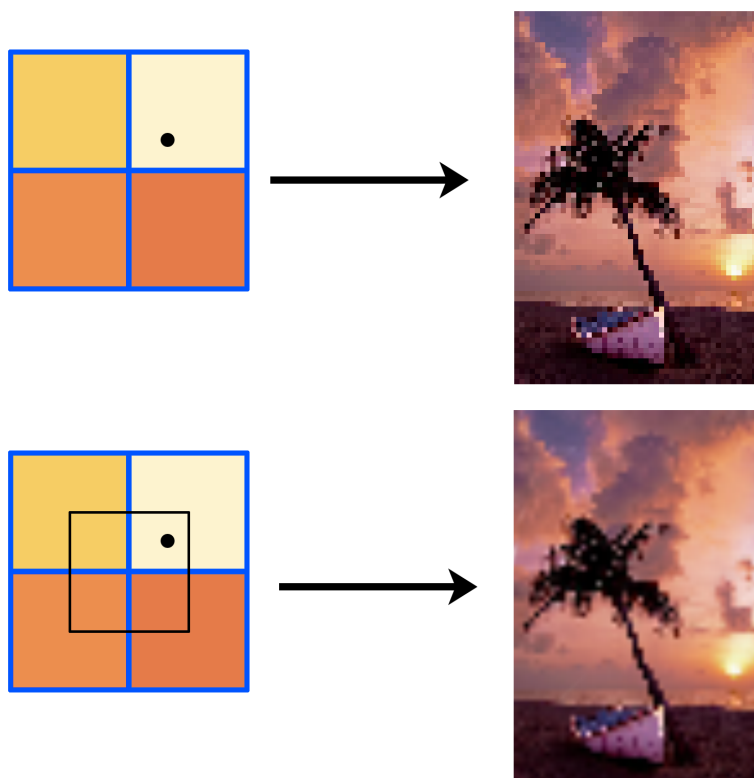


Figure 2.11: Nearest neighbour filtering (upper image) and bilinear filtering (lower image).

Examples for both nearest-neighbour and bilinear filtering are shown in figure 2.11. In the illustration, the blue edges denote texel cell boundaries, the black dot is the UV coordinate location for the rasterized primitive fragment and the black edges are the interpolation line segments for bilinear filtering. In nearest neighbor filtering we take a single sample from the texel center the UV coordinate is nearest to, which is why it is also known as "point sampling". In bilinear filtering we consider the four nearest texels and interpolate between them to construct the final sample value. Performing a given filtering method over the whole image produces results displayed on the right-hand side of the figure.

Forming the inverse of the magnification problem are the situations falling under the **minification** category, where the rasterized primitive fragments have their UV coordinates index sparsely into the texture map. Observed from another perspective, the texture map's texel cells are small enough in area compared to screen pixels so that a single screen pixel can cover many texels. Simply taking a single sample from one of these covered texels would in many cases lead to a value unrepresentative of all of the texel values as a whole, while the utilization of bilinear filtering would only slightly improve the result. While unrepresentative values cause aliasing artefacts even in completely static images, the problem gains gravity in temporal cases where there is motion present. In the described case a given texture moves over the screen, leading the inadequate filtering to rapidly change resulting values even for corresponding surface points between sequential rendered images. This can cause visible flickering, a form of *temporal aliasing*.

Flickering areas in images have been shown to cause a human user's visual attention to shift towards them and can be perceived as annoying [53] [14]. The underlying cause for the flickering in this case is that the screen resolution is not high enough to sample the texture map with the methods covered so far without aliasing being introduced.

For this situation it is useful to remember the Nyquist frequency mentioned in section 2.1.4. The Nyquist frequency and the theory behind it tells us that the sampling frequency (e.g., the screen resolution) has to be at least double that of the highest frequency in the original signal (e.g., the texture map). We then have two alternatives: either to increase the screen resolution or decrease the texture map resolution. Increasing the screen resolution to solve the aliasing problem in the general case is not a practical solution due to the additional resources an even perceptually passable result would require, so we choose the latter option. Following from the Nyquist frequency, the need for double the sampling frequency (resolution) for both dimensions gives us a clue on the resolution we should downscale our texture map to. Simultaneously, we keep in mind that the projected size of a single texture

can vary from situation to another, and resultingly that cases where little or none downscaling is desired are possible. In other words, it would be useful to have different resolution versions of a texture map that we could choose from on a case-by-case basis.

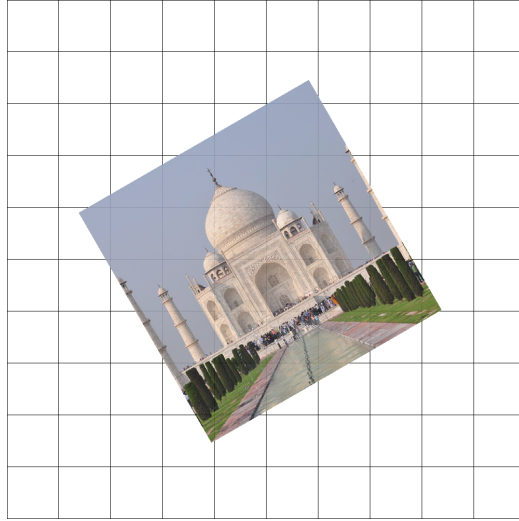


Figure 2.12: An illustration of a case where a texture has undergone minification to a degree where one screen pixel cell can cover even hundreds of texel cells.

The appropriate remedy is the method called **mipmapping**, in which progressively lower-resolution versions of the original texture map are precomputed and stored. During precomputing the texel values for lower resolutions need to be determined as ideally a collection of all of the texels values contributing to a "lower resolution" texel. This is performed by using one of several available filters, such as *Gaussian* or *Lanczos* filters, until the texture versions with dimensions of down to 1×1 have been created. Each of the texture versions in the resulting *mipmap pyramid* is called a *level* with the lowest of those corresponding to the highest resolution (original) texture and the highest level corresponding to the 1×1 resolution version of the texture. [5, p. 163]

With the mipmap pyramid complete, in cases of texture minification, we use the appropriate mipmap level to gain the closest pixel-to-texel area coverage match possible and thus a less aliased end result. The approximation can either be done by taking the longest edge length of the pixel cell's projection onto the texture or by using gradients describing the rate of change in UV coordinates compared to the change in screen pixels. As the mipmap levels

rarely result in an exact match between pixel and texel cell sizes, to further lessen the amount of aliasing we sample two adjacent mipmap levels and linearly interpolate between those sample values. This technique is called **trilinear filtering** [5, p. 166] and it not only can improve results for static images but also eliminate visible transition "seams" that could otherwise be visible on screen areas where sampling changes between different mipmap levels.

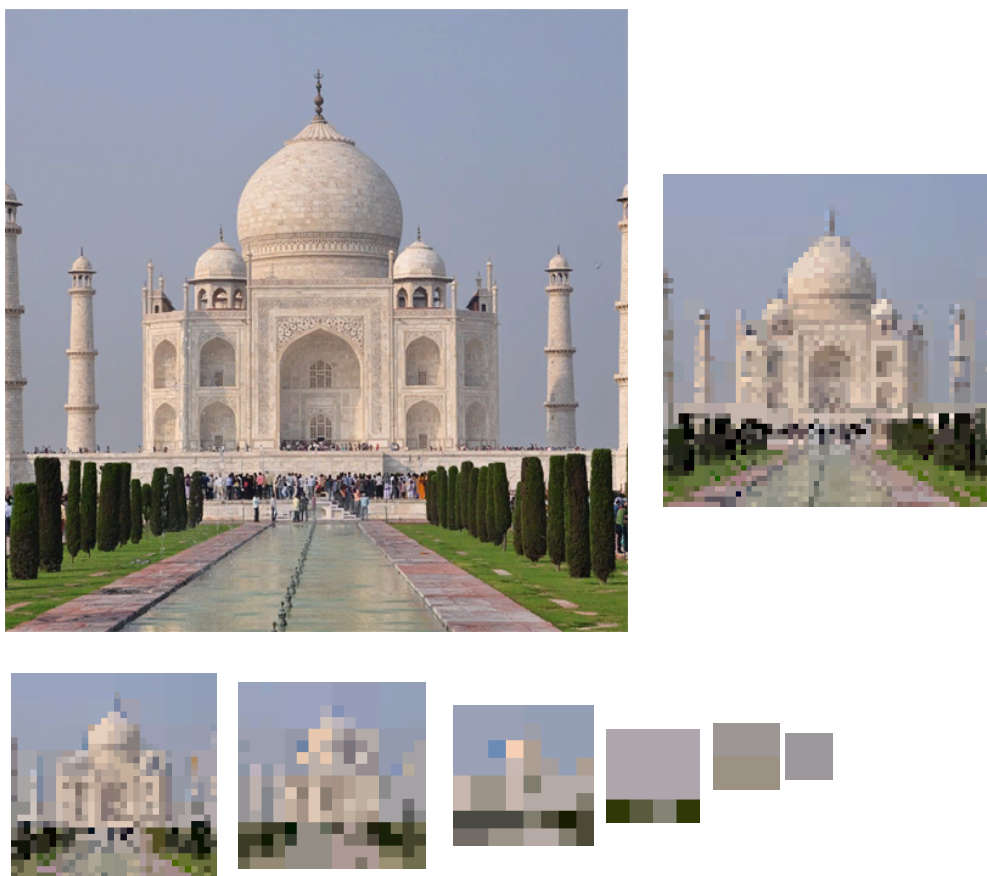


Figure 2.13: An image texture's mipmap chain, generated using the Kaiser filter. Mip levels 1 and 2 have been omitted to save space. Note that the mip level sizes here are only illustrative, and not according to actual resolutions.

To end the section on the fundamentals of texture mapping, we note that the textures in a 3D scene are very rarely observed strictly from a head-on direction. This leads to the texel cell's projection onto a screen pixel cell potentially not being a rectangle, but more commonly a trapezoidal or elliptical shape [17]. In practice this means that the density of texels covered by a pixel can vary inside the pixel cell. The result is that a single sample from

the mipmap chain is not enough in these cases as the appropriate mipmap level can vary inside a single screen pixel cell. One mipmap level, while appropriate for a portion of the pixel cell, might be too low resolution for another, leading to a blurry appearance. This is solved with anisotropic filtering methods, which perform multiple mipmap level estimations per screen pixel cell and resultingly sample the appropriate mipmap levels before forming a final filtered value for the screen pixel.

2.2.5 Filtering Normal Maps

The texture filtering methods discussed in section 2.2.4 only work correctly when the values held by the texture maps have a linear correspondence to the final shaded colors output by a BRDF, like in the case of the diffuse albedos, as an example [5, p. 271]. This linear correspondence does not hold for normal maps, as the stored normals' influence on the BRDFs themselves is not linear, and thus the simple method of creating a mipmap chain described earlier does not produce correct end results when applied on normal maps.

The more precise reason for this is due to modern reflectance equations used in shading taking a statistical approach, where a surfaces microscale orientation is described as a normal distribution. With this in mind, performing the mipmapping process would not only need to average the surface normals (from normal map texels) but also preserve the overall shape and size of the distribution lobe. See figure 2.14 for an illustration of this.

The incorrect results from foregoing this are especially visible on glossy surfaces with high specular reflectivity as sharp changes in specular highlights which can shimmer in and out of the view even with slight changes in viewer or surface position. In reality, specular reflections should exhibit slight blurring as the distance between the viewer and the surface increases [42].

Several approaches to address this problem have been proposed, such as LEAN mapping [42] and Toksvig's method [48] which both aim to better represent the variance in directions of the normals – and as a result the effect mipmapping has on the normal distribution function (discussed in the next section) which are computed based on normal map normals. Although these approaches enable results which are closer to the ground truth, they also introduce requirements on the storage format or precision of the normal maps, as well as being closely tied to specific BRDFs [39].

The object-space lighting method implemented as part of this thesis performs mip-mapping only on the final reflected radiance values, and accordingly is not susceptible to any of the aliasing problems arising from the improper averaging of BRDF input data such as the normals of a normal map.

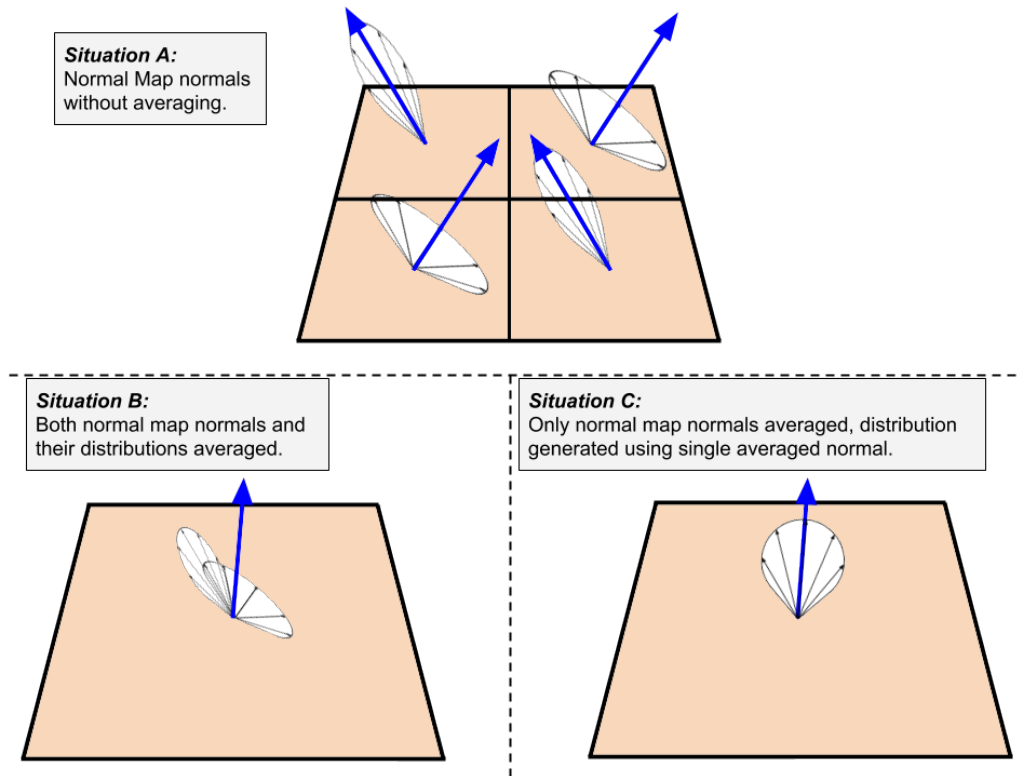


Figure 2.14: Illustration of the error caused by mip-mapping normal maps without considering the microscale distributions the normal vectors control. Situation B represents the case where we have correctly averaged also the distributions, leading to a wider and less sharp lobe. In situation C the lobe the distribution has been generated by simply considering a single averaged normal map normal vector, resulting in a sharp lobe that incorrectly represents the microscale normal distribution.

2.2.6 The Cook-Torrance BRDF

While in section 2.2 we introduced the equation for calculating the maximum reflected radiance to a viewer, it was only able to describe surface appearance for an ideally diffuse material. Real world materials are normally more accurately modeled as a combination of diffuse and specular properties. Similarly, the reflected radiance depends also on the relation between the to-view and to-light directions. In general, the fraction value of radiance reflected by a surface can be formalized for idealized, non-area light sources as the following equation:

$$f(\mathbf{l}, \mathbf{v}) = \frac{L_o(\mathbf{v})}{E_L * (\mathbf{l} \cdot \mathbf{n})} \quad (2.3)$$

where \mathbf{l} and \mathbf{n} are the to-light and surface normal vectors respectively, $\mathbf{l} \cdot \mathbf{v} \geq 0$ and L_o and E_L are the outgoing (reflected) radiance from and the incoming irradiance to a surface, measured perpendicularly with regard to \mathbf{l} . The function f in equation 2.3 is what is referred to as the *bi-directional reflectance distribution function* (**BRDF**) as it simply describes surface reflectivity with regard to the two directions, \mathbf{l} and \mathbf{v} . As we have learned in section 2.2.3, material properties including the surface orientation and material-specific reflectance also affect the final values given by the BRDF. These values vary depending on the wavelength distribution of the incoming irradiance, so it follows that the irradiance, radiance and also the BRDF result values in equation 2.3 are represented as RGB vectors.

So far we have been talking about the incoming light as a measure of irradiance, which means that we are interested in the sum of all of the radiant fluxes passing through a surface point. In the real world light could arrive from every possible direction in the hemisphere over a surface point, meaning that this sum could be mathematically thought of as an integral operating on differential irradiances. This differential irradiance can be expressed for a surface point \mathbf{p} and incoming light direction ω_i in terms of radiance as follows:

$$dE(\mathbf{p}) = L_i(\omega_i) * (\omega_i \cdot \mathbf{n}) d\omega_i \quad (2.4)$$

Knowing this we are now ready to present the equation for the total radiance reflected towards a viewer. Reordering the terms in equation 2.3, substituting incoming irradiance with incoming incoming radiance (using equation 2.4) and adding the integral summation we get:

$$L_o(\mathbf{p}, \omega_o) = \int_{\Omega} f(\mathbf{p}, \omega_i, \omega_o) \otimes L_i(\mathbf{p}, \omega_i) * \mathbf{n} \cdot \omega_i d\omega_i \quad (2.5)$$

This equation means that the outgoing radiance L_o to a view direction ω_o is the sum of all the incoming incident radiance given by $L_i(p, \omega_i) * (\mathbf{n} \cdot \omega_i)$ and scaled by the BRDF f . As the incoming radiance is used to represent the differential irradiance in this equation, we need to remember to take into consideration the distance-based attenuation from equation 2.2 when we are dealing with spherically radiating light sources. The summation is done over the hemisphere denoted by Ω , with the BRDF f controlling the fraction of light reflected per each solid angle ω_i , with the reflectivity being able to be varied according to the material properties of the precise surface point \mathbf{p}

we are examining. Note that we use the component-wise multiplication sign when multiplying with the BRDF value as the reflectance can vary per RGB primary color. Since we will not be examining the use of area-lights in this thesis, equation 2.5 can be further simplified to the following form:

$$L_o(\mathbf{p}, \mathbf{v}) = \sum_{k=1}^N f(l_k, \mathbf{v}) \otimes L_k(\mathbf{p})(\mathbf{n} \cdot l_k) \quad (2.6)$$

Rather than summing up the incident radiance for each solid angle over the hemisphere we model the incoming radiance as originating from a set number of non-area light sources in the surface point's environment. $L_k(p)$ signifies the radiance from the k th light source, while taking into consideration the distance-based attenuation for the radiant flux. In place of the previous (solid) angles ω_i and ω_o for the incident and outgoing directions, we can now use the direction vectors l_k and v analogously to denote the to-light and to-viewer directions.

There have been numerous different BRDFs devised for physically based rendering with each having their own material types they are best suited for. The BRDFs can also vary on how strictly they adhere to the energy conservation and Helmholtz reciprocity principles. The one we have selected for closer examination in this chapter and for use in the practical part of the thesis work is the **Cook-Torrance BRDF** [9] following the description by Karis [25]. It is a physically plausible specular reflection model utilizing the *microfacet theory*. Microfacet theory will be discussed later on during this section.

As previously noted, real world materials are most often partly specular and partly diffuse in how they reflect light. Accordingly, for realistic results we need an additional model to capture the diffuse reflection characteristics. For this we employ the **Lambertian reflection model** which, as can be surmised from its name, is closely related to *Lambert's cosine law*, that we introduced in equation 2.1 in section 2.2.1. Putting everything we have learned together, we can present the following high-level equation:

$$f_r = k_d * f_{Lambert} + k_s * f_{Cook-Torrance} \quad (2.7)$$

The equation tells us that the total fraction f_r of the incoming light that reflected to the viewer is the sum of the diffusely reflected fraction $f_{Lambert}$ and the specularly reflected fraction $f_{Cook-Torrance}$. Both the diffuse and specular parts need to be multiplied with additional factors k_d and k_s respectively, to ensure that the energy principle is not violated. As light that gets refracted can not (surface) reflect, it follows that $k_d + k_s = 1$. The manner in which we decide which values to use for k_d and k_s is discussed in

the section on the *Fresnel effect*. Moving forward, $f_{Lambert}$ is defined simply as:

$$f_{Lambert} = \frac{c}{\pi} \quad (2.8)$$

where c is the diffuse albedo characteristic to a material and the $\frac{1}{\pi}$ is a normalizing coefficient [21]. It ensures that values given by the $f_{Lambert}$ cannot result in cases where the amount of outgoing (diffusely) reflected light is greater than that of the incoming light. As with other BRDFs, the Lambertian BRDF naturally outputs RGB values. The specular portion of the reflectance function f_r is defined as follows:

$$f_{Cook-Torrance}(\mathbf{l}, \mathbf{v}) = \frac{D(\mathbf{h})F(\mathbf{v}, \mathbf{h})G(\mathbf{l}, \mathbf{v}, \mathbf{h})}{4(\mathbf{n} \cdot \mathbf{l})(\mathbf{n} \cdot \mathbf{v})} \quad (2.9)$$

This is a very complex function so it is best understood by examining its components in isolation. More specifically, we have the new terms D , F and G , which represent functions for modelling different aspects contributing to specular reflection. Put another way, the values given by these functions can be thought of as each scaling downwards the specular reflection amount. We will now describe each of these functions.

To understand the **normal distribution function** D , one has to first be aware of **microfacet theory**. Microfacet theory is a model which describes object surfaces as consisting of microscopic mirrors (or facets) with each of them having their own individual orientation. In computer graphics we would say that these *microfacets* have individual normal vectors associated with them. From the previously described law of reflection in 2.2.2 we know that this normal along with an incoming light ray's incident direction determines the reflected light's direction. This reflection direction has to be the same as the to-viewer direction for the viewer to be able to see it. These two facts are combined into the **half vector** which is the \mathbf{h} seen in equation 2.9. The half vector represents the orientation with which a surface's normal (\mathbf{n} in the equation) has to be aligned with to produce a reflection visible to the viewer. The situation is illustrated in figure 2.15.

The half vector can be computed simply by taking the sum of the vectors \mathbf{v} and \mathbf{l} and normalizing the resulting vector:

$$\mathbf{h} = \frac{\mathbf{l} + \mathbf{v}}{\|\mathbf{l} + \mathbf{v}\|} \quad (2.10)$$

Another detail we also need to know is that microfacets model those physical structures of a material that are of such a miniscule scale that they would not cover anything close to the area of a single screen pixel cell in

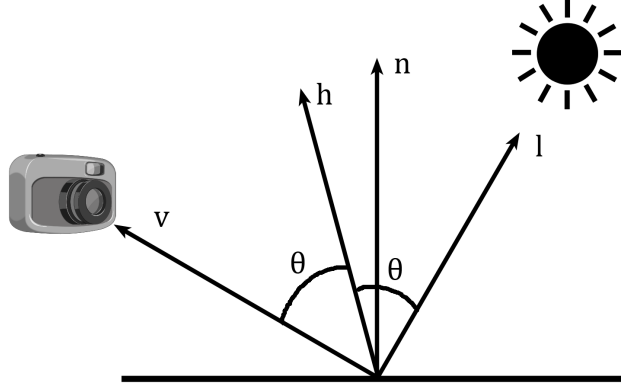


Figure 2.15: The half vector describes the orientation a surface would need to have in order for light from light source direction \mathbf{l} to be reflected towards the virtual camera direction \mathbf{v} .

any practical scene. This leads to the pixel's color to be affected by a large number of microfacets.

To reconcile this with the law of reflection, a statistical view of the microfacets is taken with the variance in orientations being parametrized as the material attribute *roughness* (or *glossiness* depending on the workflow, these are just the inverses of each other). Using roughness, the macroscale surface normal \mathbf{n} (e.g., the normal retrieved from a normal map) and the half vector \mathbf{h} , the normal distribution function D estimates the percentage fraction of microfacets aligned with \mathbf{h} . Like the name suggest, these fraction values are normalized to the range of $[0, 1]$.

As was the case with the overall BRDFs, there also exist various alternatives available for the selection of the normal distribution function. The one chosen for this thesis is the **Trowbridge-Reitz GGX**, which is defined as follows:

$$D_{GGX}(\mathbf{h}) = \frac{\alpha^2}{\pi((\mathbf{n} \cdot \mathbf{h})^2 * (\alpha^2 - 1) + 1)^2} \quad (2.11)$$

where the vectors \mathbf{n} and \mathbf{h} have the same definitions as we have previously given them. The α denotes the roughness parameter. As the roughness increases, the values given by the D_{GGX} function decrease. Physically this makes sense as the greater variance in surface orientation should result in smaller fraction of mirror-like reflection to any one direction. Similarly, the

greater the difference in orientation between \mathbf{n} and \mathbf{h} , the less statistical chance there is for mirror-like reflections.

The next part of Cook-Torrance BRDF (equation 2.9) is the F , or **Fresnel function**. Earlier in section 2.2.2 we noted that a light ray hitting a surface can be modelled as being divided into a reflected and refracted part, but we did not yet explain how we would know the fraction amounts for each these parts.

From the real world we can observe that the amount of light reflected by for example the surface of a lake increases as the elevation angle between the to-viewer direction and the lake surface's normal increases. Conversely, the lake surface appears more transparent the closer the to-viewer and normal directions are aligned together. When the elevation angle reaches 0° , the material reflects light the least and refracts (i.e., transmits) it the most. The Fresnel equations developed by Augustin-Jean Fresnel captures exactly this phenomenon, but their original formulation containing the division of light into polarized components would be computationally very expensive to perform for real-time rendering.

For this reason, a simplified approximation formula called **Schlick's approximation** [45] is used instead. In this formula we start by acknowledging that the minimum fraction of light being reflected for a given material occurs at the incident angle of 0° . This fraction value is commonly denoted as F_0 , the base reflectivity. The F_0 value can be calculated providing we know the indices of refraction (*IOR*) for both of the participating mediums, and which one of these mediums is the one being entered and which one we are exiting from.

However, the F_0 s of common mediums (e.g., air, water, wood) are readily available for use from various sources both offline and online so this calculation can usually be avoided. Understanding that we can take the F_0 value as a base reflectivity for a medium and that the reflectivity decreases as the incident angle of a light ray increases, Schlick was able to formulate his approximation as follows:

$$F_{Schlick}(\mathbf{h}, \mathbf{v}) = F_0 + (1 - F_0)(1 - (\mathbf{h} \cdot \mathbf{v}))^5 \quad (2.12)$$

where the \mathbf{h} and \mathbf{v} are again the half-vector and the to-viewer vector respectively, with F_0 being the base reflectivity for the material we are interested in. The original formulation used the surface normal vector \mathbf{n} instead of the half-vector \mathbf{h} we use here, but this is just a modification to aid the usability of the formula without changing its meaning. The aim of the Cook-Torrance BRDF (which the $F_{Schlick}$ function is a part of) is to model mirror-like reflections, and as those reach the viewer only when $\mathbf{n} = \mathbf{h}$, the

change is justified.

The Fresnel effect changes depending on the wavelength of light ray's so F_0 and the reflectance values given by $F_{Schlick}$ are both represented as RGB vectors. The values for each component is in the range $[0, 1]$ with the meaning being that 1 is the case where all incident light for a given component is reflected, while 0 means that all of the light for that component is transmitted. This is how we arrive at the values to use for the coefficients k_S and k_D in equation 2.7, with $k_D = 1 - k_S$ for each component.

The final part of the $f_{Cook-Torrance}$ BRDF (equation 2.9) is the **geometry function** G modeling the geometric occlusion caused by the roughness of a surface – a form of microscale self-shadowing. Again many alternatives exist for the geometry function, but the one selected for examination here is the one known as **Schlick-GGX** using Smith's method. The Schlick-GGX is defined as:

$$G_{Schlick}(\mathbf{x}) = \frac{\mathbf{n} \cdot \mathbf{x}}{(\mathbf{n} \cdot \mathbf{x})(1 - k) + k} \quad (2.13)$$

where \mathbf{n} is the surface (macro) normal, \mathbf{x} is the direction in relation to which we want to know the occlusion and k is a remapping of the roughness value α seen earlier. According to Karis [25] this is intended to correct an error in the original formulation of $G_{Schlick}$. The remapping is defined as:

$$k = \frac{\alpha}{2} \quad (2.14)$$

Returning to equation 2.13, the intention is to calculate the geometric occlusion based on the direction \mathbf{x} . When using Smith's method, geometric occlusion is modeled as comprising of two different factors. The first one is the amount of incident light occluded (or shadowed) due to the surface roughness, decreasing the illumination to a particular surface point. In this case the direction \mathbf{x} would take the value of the to-light vector \mathbf{l} . The second one is the occlusion amount for the reflected light caused by surface roughness. Here we think of the obstruction depending on the to-viewer direction \mathbf{v} . In practice we simply compute the normalized fraction values from $G_{Schlick}$ using \mathbf{v} and \mathbf{l} as the arguments for the function and multiply the results together:

$$G(\mathbf{v}, \mathbf{l}) = G_{SchlickGGX}(\mathbf{v}) * G_{SchlickGGX}(\mathbf{l}) \quad (2.15)$$

The results given by $G(\mathbf{v}, \mathbf{l})$ are normalized values in the range $[0, 1]$, so they can also be thought of as the percentage of being able to escape any geometric obstruction by surface roughness. Indeed, given an amount of incident light to a surface point the three functions D , F and G can be seen as giving the percentage values of light being reflected due to on

the distribution on microfacet orientations, the amount of reflection versus transmission when interfacing between particular medium or the surface's self-shadowing. Putting everything together and to sum up the discussion, we can now present the completed form of the reflectance equation 2.7 we introduced earlier:

$$L_o(\mathbf{p}, \mathbf{v}) = \sum_{k=1}^N \left(k_d * \frac{c}{\pi} + k_s * \frac{DFG}{4(\mathbf{n} \cdot \mathbf{l}_k)(\mathbf{n} \cdot \mathbf{v})} \right) \otimes L_k(p)(\mathbf{n} \cdot \mathbf{l}_k) \quad (2.16)$$

We have implemented this equation for use in the shading part of our object-space lighting algorithm. In practice, it is wise to augment the $4(\mathbf{n} \cdot \mathbf{l}_k)(\mathbf{n} \cdot \mathbf{v})$ term in equation 2.16 with a small ϵ constant to avoid division by zero. Another practical consideration is the fact that the normal \mathbf{n} in the equation is the normal sampled from a normal map, and thus can lead to cases where backfacing surfaces (from the light source's point of view) can be lit even though this does not make sense geometrically. To prevent this, we perform a test to see whether a surface is backfacing by mandating that the dot product of the geometric (primitive) normal of a surface and the light direction vector \mathbf{l} has to be non-zero.

Chapter 3

Practical 3D Rendering

Up to this point we have put the majority of our focus on explaining the theoretical basis for the computer graphics pipeline. While it certainly is possible to implement the theory covered by building everything from the ground up using the basic libraries languages like Java, C++ and many other similar programming languages provide, this is rarely done. The reason for this is that any practical implementation of the graphics pipeline using the rasterization method for visibility determination is bound to perform better when executed on a **GPU** (*Graphics Processing Unit*). The GPU is a computer hardware component designed for efficient rendering operations, including fast matrix multiplication, texture sampling and triangle rasterization.

To utilize the GPU's power, rendering applications have to be programmed through the use of a graphics programming API (*Application Programming Interface*), of which several alternatives may exist depending which platform we are using (e.g., a Windows PC, a videogame console, a mobile device etc.). Rather than giving completely free reigns to the programmer, the graphics APIs typically expose only a carefully structured subset of the graphics pipeline. When the first major APIs (such as Microsoft's **Direct3D** [36] and the Khronos Group's **OpenGL** [47]) were originally released during the 1990s, programmers were restricted to such a degree that they could not explicitly control the surface shading algorithms used in their 3D applications [5, p. 33].

As years have progressed, the primitive transformation and the fragment shading portions of the pipeline have progressively been made more programmable, yet there several areas have still remained off-limits. The most glaring of them in our opinion has to have been the inability to implement a fully hardware-accelerated (in effect disregarding the compute shader capabilities discussed in the following section) visibility determination based on ray-tracing, but this has likely to be due to the scarcity of hardware sup-

port for it rather than strictly an API omission. Recent developments and the announcement of DirectX Raytracing [35] indicate, however, that this limitation may soon be a thing of the past.

3.1 The Programmable Graphics Pipeline

To gain an improved appreciation for how the theoretical graphics pipeline maps to the one present in a modern graphics API, we will give a high-level walkthrough of the Direct3D 12 rendering pipeline [38]. The rendering pipeline can be conceptualized as the functional chain of stages that is commenced by a **draw call** being passed to the GPU. A draw call is an instruction which will at the least include information on which vertices to render as well which rasterization and general rendering settings to use. The following diagram serves as an abstracted overview of the pipeline:

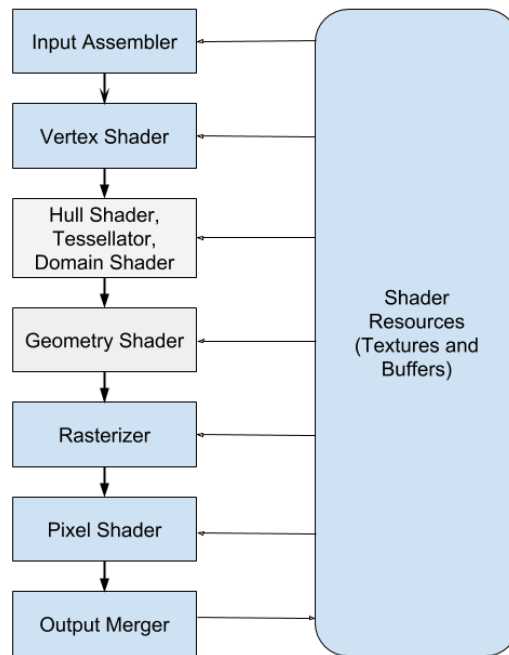


Figure 3.1: The Direct3D 12 rendering pipeline.

The stages containing the term *shader* in their name, are stages the graphics programmer can control by writing small programs called **shader**

programs. These are written in a C-like programming language, which in Direct3D 12's case is the **HLSL** (*High-level Shading Language*) [37]. In the diagram it is also illustrated how all the different stages comprising the pipeline link together in sequence, with the arrows between them signifying data being passed from one stage as the output to another stage as the input. Do note also the use of additional resources such as texture maps and camera and light source information which are passed to the pipeline as *uniform* or constant values. This simply means that they do not vary depending on individual vertices being processed by the pipeline, but are instead uniformly defined for all of the vertices of a given draw call.

The reason graphics APIs use a sequential pipeline such as the one in the diagram owes to the fact that the rendering pipeline itself can be parallelized. APIs like Direct3D utilize the **stream processing paradigm** [32] to do exactly this with the input data being streamed through both programmable and non-programmable stages. In stream processing terms these stages would be called kernel functions [18]. The benefit gained from this is that the rendering performance can be increased by increasing the number of computational units and moving a part of the workload to the new units – providing that we are not limited by memory resources or other ancillary hardware in size or throughput. In the case of normal graphics pipeline programming the developer needs to only be mindful of providing sequential stream input data (i.e., vertices and uniform resources) and the programmable kernel functions (i.e., shader programs) for the GPU. The GPU drivers will perform the mapping of the streams to the available streaming processors [26].

3.1.1 The Input Assembler and the Vertex Shader

We now move on to describing the pipeline stages. Starting from the top of figure 3.1 we have the **input assembler** stage in which the vertices we wish to use for rendering are assembled into primitives, e.g., triangles for use in other stages. The vertices are provided in a contiguous block of data named the **vertex buffer**. The vertex data is declared on the CPU side by the application (which can be written in e.g., C++) and then copied from *system memory* into *GPU memory* from where it can be used by the HLSL shader programs.

A special **index buffer** is also used in this stage, if available, to map the vertices to be used for each triangle. The justification for using an index buffer is that commonly the polygons we render share vertices with adjacent polygons, meaning that we would have to duplicate some of the same vertex data for each individual polygon. With an index buffer we can store each

vertex only once and then reference it through the indices, leading to memory storage and bandwidth benefits. [31]

The next stage is the **vertex shader**, which is a fully programmable one. This stage has the responsibility of transforming the positions of the input vertices, originally declared in object local space, into clip space coordinates as described in section 2.1.1. This is done so that the GPU can properly rasterize the polygons. The rasterization process itself is not exposed as programmable in the Direct3D 12 pipeline but is instead a fixed function stage.

Additionally, in the vertex shader we can pass texture coordinates and perform other per-vertex transformations, for example on vertex normal and tangent vectors, for use in the later stages. Do note that these transformations need not result in clip space coordinates, but can be defined in relation to an arbitrary coordinate system as is purposeful. A typical example of this is to pass the world-space position coordinates as output from the vertex shader in addition to the clip space equivalents, to perform lighting computations in world space.

The vertex shader stage is followed by the geometry and tessellation shader stages. These aim to enable primitive-level manipulation of geometry and the runtime derivation of higher-detail meshes using control-point based subdivision, respectively. The geometry and tessellation stages are not relevant for the practical work part of this thesis, so they will not be discussed further.

3.1.2 The Pixel Shader

Following the vertex shader, the next programmable stage that is of interest to us is the **pixel shader**. The pixel shader is executed for each individual back buffer pixel covered by a primitive, enabling per-pixel computations. Another way to look at this is to say that we perform computations per primitive fragment, which is where the OpenGL equivalent's name, *fragment shader*, comes from.

The inputs for this stage depend on values output from the vertex shader, with the difference between them being that the pixel shader inputs have undergone interpolation to be correct for each fragment. This means we can have pixel-precise texture coordinates, normal and tangent vectors and various other useful information. The texture coordinates can be used to retrieve texture map values through HLSL provided **texture samplers**, with the numerous addressing and filtering options described in section 2.2.4 being available. In line with the stream processing paradigm, any texture maps used will need to have been copied from system memory into GPU memory

before the stream processing has started.

The level of precision afforded by the pixel shader makes it suitable for operations demanding high-frequency sampling such as the BRDF shading calculations discussed in section 2.2.6. This is why the bulk of lighting computations are typically performed in the pixel shader as opposed to the vertex shader. It is still useful to remember that moving computation to the vertex shader can be beneficial in cases where the added precision is either unnecessary or less important compared to the performance benefits gained through it.

3.1.3 Forming the Final Image

The final stage of the Direct3D 12 pipeline is the **output-merger stage** where the final pixel color values are decided. It is a fixed function stage over which the programmer has control only through specifying state settings. These states include the *blend*, *depth* and *stencil* states.

The blend state controls how a new pixel color value fresh from the pixel shader should be blended with a potentially already present pixel color value in the back buffer. This is useful if a programmer would be interested in implementing a transparency effect, as an example.

The depth state determines when or if back buffer color values should be overwritten depending on the depth value for the fresh pixel compared to the extant pixel in the back buffer. The extant pixel depth values are held in the depth buffer, a buffer with the same dimensions as those of the back buffer.

Finally, the stencil state controls how the stencil buffer is used and how back buffer writes depend on it. The stencil buffer is again a buffer with equal dimensions to the back buffer and is probably best described as a scratch pad for keeping track of occurrences a developer might be interested in. For example, during development it can be useful to track the number of pixel shader invocations by setting the values in the stencil buffer to be incremented after a pixel shader has completed. [31]

3.1.4 The Compute Shader

A separate but powerful stage outside of the sequential graphics pipeline is the **compute shader**. The compute shader is intended to be used for the more general computational tasks which do not lend themselves well for graphics processing pipeline covered so far, but can still make use of the stream processing paradigm. This form of computation on the GPU is commonly referred to **GPGPU** (*General-purpose computing on graphics processing units*).

As GPUs are designed to operate on contiguous blocks of data, the input resources for GPGPU tasks are buffers and textures. The distinction between buffers and textures is made here, as the texture resources can be sampled using the hardware texture units of a GPU for texture filtering. This is a useful feature particularly for image-processing related workloads. The compute shader API also provides different buffer types to choose from based on the type of read or write access (or a combination of them) the developer desires to use. [31]

While the graphics pipeline operated on primitive vertices which are further down the stream transformed into the polygon fragments covering a pixel, compute shader programs operate on threads. A thread can be regarded as the smallest part a work task can both logically and practically be divided into. If, for example, we desired to write a compute shader to alter a digital photograph, we would have a thread correspond to one texel of the texture map representing the photograph.

As the compute shaders themselves are kernel functions that have to be generically applied to the threads, the GPU automatically generates various *system ID* values for use as input in the shaders. Through these ID values we are then able to control execution down to a specific thread's level. [18]

The results of a compute shader can be written to a write-enabled buffer provided the developer has declared it. As this buffer resides in the GPU memory, it has to be copied to the CPU memory if further processing on the CPU side is desired or the results need to be stored on a storage device such as an SSD. There is a performance penalty associated with moving data between the different memory pools so it is to be avoided if possible. An example opportunity for this would be when we utilize the compute shader to compliment the graphics pipeline by performing texture map alterations. The resulting altered texture map would then be sampled in the pixel shader when calculating a back buffer pixel color. As the pixel shader operates on resources in the GPU memory, there would be no need for a copy back to CPU memory in this use case. [31]

3.2 Rendering Methods

3.2.1 Forward Shading

The described stream processing method of taking geometrical primitives through the graphics pipeline by transforming them in the vertex shader and finally performing lighting operations on the primitive fragments in the pixel shader is called **forward shading** [28]. While this method has the benefit of

simplicity of implementation for scenes with highly homogeneous light source and surface material types, it begins to reveal its flaws as the heterogeneity increases [7].

In section 2.2.6 we described a version of the Cook-Torrance BRDF, which while popular in modern applications, is only one of various possible light-material interaction models one could use. To model light's interaction on human skin for example, one would be well advised to explore **BSSRDF** (*Bidirectional scattering-surface reflectance distribution function*) techniques [11]. These provide ways to more accurately describe body reflections compared to the simplified Lambertian diffuse reflectance model, but also require a different set of material parameters compared to the Cook-Torrance BRDF.

Let us imagine a scene where we had a human model with exposed skin and a model with a surface exhibiting very limited diffuse reflection, such as a metal sword, we would desire to have different shading routines for both of these surface material types. This would mean authoring separate vertex shader (to preparing shading parameters for interpolation during rasterization) and pixel shader (to implement the BRDF or BSSRDF) programs for each case. The number of different vertex and pixel shaders naturally increase as the scene's complexity for shading types increases. This inconvenience turns into a problem when we consider that for reflectance computations we also need to be able to compute the irradiance contribution from potentially numerous different types of light sources (e.g, directional light, point light, spot light, etc.) as each of them requires a type specific method. Using programming constructs such as loops and branches is possible but is associated with a performance penalty (although [7] argues "it is no longer prohibitive"), while authoring different shader programs for each case might be impractical even if it could be automated. [5]

Another issue with forward shading is that during the rendering of a single frame fragments from many different primitives may be rasterized onto the same back buffer pixel. Due to how the depth buffer works (with fragments further away from the virtual camera being overwritten), this can lead to the pixel shader being executed for a given pixel only for its result (the pixel color) to be overdrawn later by another invocation of a pixel shader. This clearly leads to a waste of compute and memory throughput resources.

It is possible to mitigate the problem to an extent by ensuring through a *depth-sorting algorithm* that the primitives closest to the camera are rendered first, and then utilizing the hardware and API provided *early Z pass* method to avoid the execution of the pixel shader for non-visible fragments. In practice, depth sorting algorithms cost additional performance and for the general scene are not entirely robust, while the early Z pass technique may

not be compatible with the specific shading methods a developer wishes to use as it needs the depth information for a fragment to be unchanged through the pixel shader. [5]

3.2.2 Deferred Shading

A presently popular solution to the problems posed by forward shading is to use a method called **deferred shading**. In deferred shading, as the naming suggests, the shading or lighting computations are postponed to be executed only after the visible fragments and their related material properties have been decided. Essentially we decouple the material property evaluation from the irradiance contribution and surface reflectance evaluation. [5]

In practice this is achieved by initially passing the scene geometry through the graphics pipeline, but instead of writing the results to a back buffer with the (almost) final colors to be displayed on a screen, we write material properties to an intermediary geometry buffer, commonly referred to as the **G-buffer**. The G-buffer can be implemented either as separate buffers for each material property or as one aggregate buffer. [5]

Whichever solution we choose, the relevant detail to remember is that just as was the case with the depth and stencil buffers, the G-buffer is also to have the same dimensions as the back buffer. This is because in effect we are rasterizing the material properties to be used later on in the shading of each back buffer pixel. Similarly, as the material properties are to be used as parameters for the BRDF or BSSRDF shading implementations, we need to ensure that we store all of the needed properties such as the diffuse albedo, the surface normal vector and the roughness value. An example visualization of a possible G-buffer's contents is given in figure 3.2.

The final computation of reflected light values for each back buffer pixel is achieved through performing rendering passes using light volumes. The light volumes in this context mean simple geometric shapes such as spheres or cones representing the area of effect for point or spot lights, as an example. The light volumes provide an elegant way to avoid lighting computations for cases where the distance-based attenuation on surface irradiance would have caused the effective contribution to be close to zero – be it actually or only perceptually. [5] In practice, using meshes such as cones or spheres may performance-wise be a sub-optimal solution, so simple axis-aligned bounding boxes are also used. [28]

As the light volumes are rasterized we not only gain their coordinates on the back buffer, but also the coordinates for indexing into the G-buffer. This is how we can use the material values in conjunction with the light source properties to compute the irradiance to a surface fragment and finally the

amount of reflected outgoing radiance to the virtual camera. Contributions from multiple light sources can be computed by using additive blending (in the output-merger stage) to sum up the total reflected radiance.

On the shader management side, the decoupling of *material evaluation* and *light computations* from each other means that we are no longer faced with a potentially impractical number of material and light source type combinations. With regards to performance, an evident benefit compared to forward shading is that with deferred shading we now only compute reflected radiance values for visible fragments.

At the same time this also presents a drawback from sampling perspective as we only have information from one fragment to base our computations on. This means that traditional hardware supported MSAA can not be used (as we have no sub-pixel information for fragment coverage), meaning that post-process methods are at present the only practical anti-aliasing solution for most applications using deferred shading. [7]

Deferred shading introduces also some negative performance impacts of its own. Due to having to store a possibly large number of material parameters, the G-buffer can reserve a considerable amount of GPU memory as well as cause a major increase in GPU memory bandwidth consumption during its construction phase in the geometry pass. As was the case with forward shading, the early Z pass can be utilized in the geometry pass to minimize overdraw and thus save on memory bandwidth costs. [5]

When discussing deferred shading, it is important to realize that it is not a method that is exclusively useful for shading directly to screen pixels (as is done in screen-space shading). Indeed, during our thesis work we found the decoupling of surface material evaluation from the shading computations to be very useful in our implementation of object-space lighting. In the implementation, the G-buffer is an object-specific buffer into which we evaluate material properties such as diffuse albedo, surface normal and the roughness parameter right at the 3D scene initialization phase. This can be done, as the G-buffer contents do not change during runtime (in contrast to screen-space shading, where contents vary based on visible surfaces) and therefore removes the need for any further material computations after program start-up.

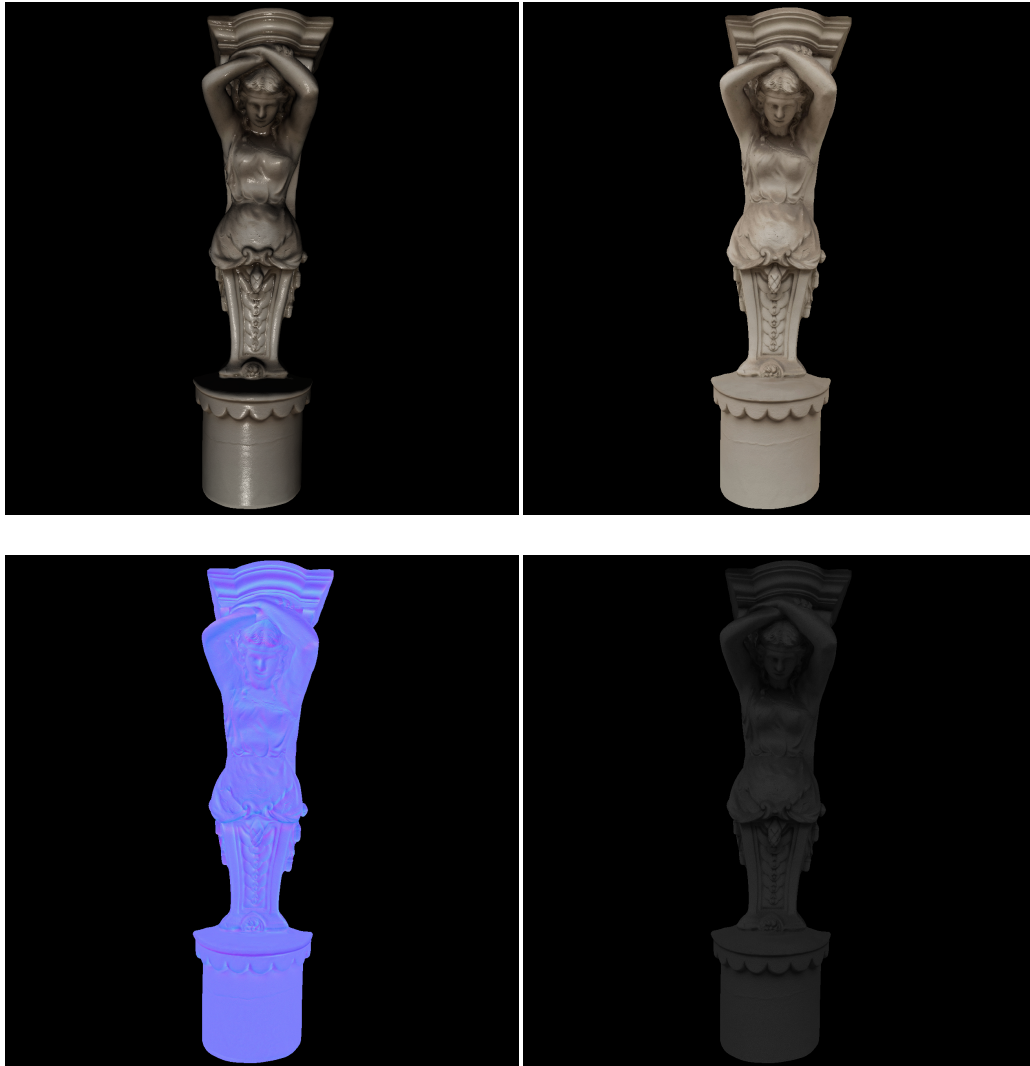


Figure 3.2: A visualization of the contents of a G-buffer and the final shading result (top left). The material parameters visualized here are the diffuse albedo (top right), the surface normal (bottom left) and the surface roughness values (bottom right).

Chapter 4

Object-Space Lighting

4.1 Motivation for New Rendering Techniques

In the field of computer graphics a great emphasis is often placed on achieving an optimal balance between the quality of visual presentation and the rendering performance. In practice this means that programmers will generally pursue the development and implementation of rendering techniques that lead to a decrease in the system resource usage – whether it falls on the CPU or GPU side, or on the memory subsystems, while aiming to keep presentation quality at a level similar to the original.

This work is often necessitated by competitive market pressures. An example case of this has been the recent drive towards **Virtual Reality** (*VR*) enabled applications and computer games [15], where an assortment of additional burden is placed on resource usage. Compared to traditional desktop monitors and television and mobile screens, virtual reality headsets present the user with a separate screen for each eye, enabling the applications to make use of the human visual system’s (*HVS*) stereo vision properties for an improved sense of presence [29].

The two screens and thus two views necessitate modifications to the portion in rendering pipeline we termed as *visibility determination* in section 2.1. As the views need to be represented by two slightly offset virtual cameras (to simulate two human eyes) we need to store and perform separate view transformations per camera. Further along the pipeline this can lead to the geometric primitives being rasterized onto slightly different positions on the screen (or possibly even being culled or occluded) with different evaluated surface properties for each camera.

The differing properties then lead to different view and light directions and also different sampled material parameters, meaning that separate shad-

ing executions are needed for both of the stereoscopic views. The end result is that, for a straightforward implementation of stereo rendering, we would need to double the number of draw calls and graphics pipeline passes in addition to needing extra memory space for storing the separate transformation matrices and frame, depth and stencil buffers. On the GPU side the pixel shader with its lighting computations is commonly the most expensive on resources, so the doubling of resource usage in stereo rendering has the highest *computational latency* cost here.

Another requirement for VR is that of the applications operating on a relatively high framerate. While traditionally many 3D applications and computer games have targeted a frame buffer update rate of 60 Hz, for VR the recommended rate is notably higher. This is in part because modern VR headsets incorporate a form of headtracking, where the user's physical position and orientation directly affect the positions and orientations of the virtual cameras, and thus the final rendered images. Another factor is the desire to minimize screen flickering, as VR headsets commonly use displays with low pixel persistence. [1]

A low frame buffer update rate (measured as frequency, $\frac{1}{s} = 1 \text{ Hz}$) means higher computational latency (measured in time, 1ms) between the commencing of rendering and the output of the final images to the screens. If no optimization counter-measures are taken, this high latency can be sensed by the user as a mismatch between her own position/orientation and that suggested by the displayed images. This can lead to motion sickness, which is why the frame buffer update rate should be kept consistently high. [1]

Current guidelines from major VR headset manufacturers recommend update rates of up to 90 Hz (in the case of Oculus VR, HTC and Valve Corporation) or 120 Hz (from Sony Corporation), although they also propose the possibility to use update rates as low as 60 Hz or 90 Hz provided that a motion interpolation technique such as that described by van Waveren [52] is used.

The contrasting pressures to both target high frame buffer update rates, while having to face the increased resource usage load posed by VR rendering form a challenge in their own right. However, it truly actualizes when we simultaneously need to strive towards providing the end users with a level of visual fidelity similar to which they have been accustomed to with previous non-VR applications. This then forms a clear motivation for devising novel techniques which not only are able to offer performance benefits but ideally also allow us to retain a visual quality close to that of the more expensive previous techniques.

4.2 Shading in Object-Space

As noted, the shading part of the rendering pipeline is the one incurring the highest cost from stereo rendering needed for VR. This is due to shading being performed in the pixel shader once for each geometric primitive fragment that has been rasterized onto the frame buffer. This is the same for both forward and deferred shading as they both perform their lighting calculations in screen space on a per-screen pixel basis. Lowering the rendering resolution is certainly a possible mitigation measure, but it can lead to unacceptable losses in fidelity and increases in aliasing artefacts. A perhaps more intriguing path to explore would be to attempt to decouple the shading performance from frame buffer resolution. This way the shading costs of stereo rendering could be lessened while still maintaining a high frame buffer resolution for the rasterization of scene geometry.

One category of such alternative shading techniques is traditionally called **texture space shading** or **object-space lighting** [7]. We hold the view that both of these terms have their place, as they each highlight different important aspects of the category. On a base level we indeed desire to perform the lighting on a per-object basis, where we can separately define the sample rate both *spatially* – as well as *temporally* if needed. While decoupling shading from the uniform frame buffer resolution can be beneficial, object-space lighting allows us to change the shading update rates separate from the frame buffer update rate [19].

A potential usage scenario would be to update the shading for far away objects (possibly covering a small number of frame buffer pixels) at half the frame buffer update rate as this would free up rendering resources and the difference in visual quality might not be impactful for the user. Remembering that high frame buffer update rates are heavily recommended in VR rendering, this can be significant avenue for performance improvements.

While the term *object-space lighting* harkens more to the high-level concept of the category, the term *texture space shading* illuminates the practical implementation side. To achieve per-object shading results we can use an additional texture map in which we store the reflected radiance for each texel. This texture map covers the object’s entire surface area, so through texture filtering we will be able store and reconstruct the reflected radiance for the given object. We will use the term **light map** for this additional texture map henceforth in this thesis. The use of the light map as a shading space means that in addition to the back buffer properties such as resolution and refresh rate, shading can also be decoupled from the final geometry pass of the scene geometry. As Hillesland [20] points out, a further pursue of this

avenue can lead to novel anti-aliasing methods.

The light map texels need to have a correspondence to a 3D mesh's surface through normalized UV texture coordinates as described in section 2.2.4. Once each light map texel has its appropriate radiance value, we can sample the light map on a subsequent render pass to gain the correct reflected light values for the screen-visible parts of an object's surface. Put in simpler terms, the technique described here can be thought of as *painting* the reflected light on to the light map. The key distinction to other common texture maps such as the diffuse albedo maps is that the light map contains the final radiance values as output by e.g., a BRDF, rather than being a map used in providing input values for a reflectance function. This enables us to avoid the problems arising from the mipmapping of normal maps in particular (discussed in section 2.2.5), allowing for more realistic rendering.

It is important to note that even though it is not presently popular in modern rendering systems, object-space lighting in itself is not a new concept. As an example, it was used in the Reyes rendering algorithm most famous from its inclusion in Pixar Animation Studio's Renderman rendering software until 2015 [20].

4.3 Practical Implementation of Object-Space Lighting

We set out to implement an object-space lighting technique utilizing texture maps as the storage for the reflected light values. An implementation of a traditional forward shading (in screen-space) pipeline was also created for verification and qualitative comparison purposes. This was to serve both as a form of *ground truth* for assessing the correctness of the object-space lighting implementation, but also due to the thesis work including a user study portion. Through the user study we had the goal of evaluating the users' ability to perceive differences between the ground truth version and the object-space implementation (to provide further verification) and to uncover possible preferences between scenes rendered using the different lighting techniques.

The user study will be described in detail in chapter 5, while in this chapter we will focus on giving a walkthrough of the different parts of the object-space lighting implementation.

4.3.1 The Software Used for Implementation

A vital consideration at the beginning of any software implementation is the decision on which programming technology platforms to use. For our rendering purposes the possible alternatives were narrowed down to technologies in three categories:

- **Low-level rendering APIs** (such as *Direct3D* and *OpenGL*)
- **Mid-level frameworks** (e.g., NVIDIA's *Falcor* [40])
- **Production-ready game engines** (e.g., *Unreal Engine 4* [12] and *Unity* [51])

All of these alternatives utilize the GPU for accelerated rendering tasks, which is what we desired. This desire also meant that CPU-based software rendering was never a real option for the final implementation, although it was used in early stages for proof of concept work.

The low-level APIs generally provide the developer with the greatest degree of control over the entire graphics pipeline, while higher level frameworks and engines hide some functionality behind abstraction layers. This can lead to complications in specific cases where precise control is needed but not easily accessible to the developer. The major benefit from using the frameworks and game engines is the increase in productivity they offer in the general use case. These considerations were taken into account and because of them the idea of using low-level rendering APIs was quickly abandoned.

Work initially commenced on the Falcor platform but we quickly found that the unfinished nature of the framework (this is only in our own estimation, and we do want to note that Falcor was at the time only at a *beta* stage) coupled with the scarcity of documentation posed serious challenges for productivity.

A decision was taken to move the work to Unity, a game engine tested through years of both commercial as well as hobbyist 3D software production. The wealth of readily available documentation and support online was also a key factor behind the decision. While the workflow and main programming language used differed between Falcor and Unity, the porting of existing code was relatively swift so as not to pose a problem.

In Unity the workflow's main areas of interest for a programmer are the **C#** (or alternatively *Javascript*) **script files** and the **shader and compute files**. The shader files correspond to programmable stages of the graphics pipeline described in section 3.1, whereas the compute files contain compute shader programs (section 3.1.4). The authoring of both of them is done in a

variation of Direct3D's and DirectCompute's HLSL. The script and shader files are assigned to their appropriate objects in a particular 3D scene through Unity's *editor interface*, while the compute files can be assigned as properties for the script files. This is how the compute shader programs can be executed from the scripts. The execution of assigned scripts, shader and compute files is by default performed automatically during each frame update by Unity.

The screen-space shading implementation was done by authoring shader files containing vertex and pixel shader programs, while the object-space lighting implementation called for the utilization of shader, compute and script files. We will now proceed by giving an examination of the latter technique's implementation.

4.3.2 Object-Space Lighting Algorithm Overview

Before moving on to detailed explanations of the object-space lighting algorithm steps in the following sections, we present a general overview for the rendering of a single 3D model in a scene:

- **Pre-program:**
 - Uniquely parameterize the 3D model's surface, so that its triangles can be mapped into a helper texture (the G-buffer).
- **On 3D scene initialization:**
 - Fill the G-buffer with surface information.
 - * Includes the surface normal, diffuse albedo, roughness etc.
- **For every lighting update for this model:**
 - Perform shading using information in the G-buffer and store radiance values into the Light Map.
 - We use the **same reflectance functions** as with our Screen-Space Shading implementation.
- **For every screen-update:**
 - As with Screen-Space Shading, project the 3D model to 2D, but instead of doing lighting calculations just **sample the Light Map** to retrieve correct radiance values.

4.3.3 Mesh Parameterization

The first problem that arises when designing a shading method that operates on texels is the question on how to map these texels to a 3D mesh's surface. The naive solution would be to simply use the mesh vertices' UV texture coordinates, but this does not work well in the general case. The artists creating meshes and texture maps often use authoring methods, such as the mirroring of UV coordinates around a reflection plane so that – as an example, both the left and the right hand side of a human face's mesh are textured by a shared area of a texture map. In other words, the texels are not uniquely mapped for the mesh's surface.

As the light map is to store the correct reflected light values for the entire mesh surface, we need to have unique UV coordinates. In 3D modelling and rendering these are called **unwrapped UV coordinates**, and the process of generating them is known as *mesh unwrapping*, a particular form of mesh parameterization [5, p. 153]. The creation of an mesh unwrapping algorithm or even the implementation of an existing one was outside of the scope for this thesis, so the remaining two alternatives were to either procure meshes with artist-made, ready unwrapped UV coordinates or to utilize existing unwrap software.

Unity includes such functionality for its own light baking functionalities, but our own tests revealed it to produce discontinuities in the generated UVs leading to the appearance of "cracks" in final render results. We hypothesize this to be related to the conservative rasterization algorithm we implemented (described in section 4.3.6), but this realization came too late in the project to be properly tested for validity. Irregardless of the exact cause, the artefacting meant that the only viable option left was to utilize meshes with pre-made unwrapped UVs, which coupled with budget limitations for the thesis greatly limited the number of available and suitable meshes.

4.3.4 Rasterization

As we now have a way parameterize the mesh, the next step is to devise a method for rasterizing the mesh polygons into a texture – such as the light map. For the thesis work we support only meshes which consist of triangle primitives. The rasterization is needed so that we are able to fill the texels covered by a mesh triangle with shading related information. This information in the case of the light map is the collection of reflected light values, but there are other useful pieces of information that we would like to store also.

Although rendering performance is not a key consideration of the thesis

work implementation, we decided to take a conceptually simple optimization step. In a basic light map implementation one could perform the mesh triangle rasterization and for each filled texel calculate through vertex attribute interpolation the appropriate position coordinates, vertex normals and other similar properties. The mesh's associated normal and diffuse albedo maps would be sampled via interpolated UV coordinates and all of the gained material properties could be passed in conjunction with uniform attributes (including virtual camera and light source position values) as inputs to a BRDF, which would then calculate the reflected radiance. This value would then be stored in the lightmap per each texel.

If we are to assume that the models have a rigid surface structure – in the sense that the UV coordinates of the mesh triangles' vertices do not change after program initialization, it is clear that there is no need to perform the surface material evaluation more than once. This gives us the option to either precompute and store the material values, sacrificing memory space but saving on computation and memory bandwidth costs, or to perform the material evaluations (and thus waste computation resources) each time we want to update the reflected radiance values. With the latter option we would of course avoid incurring the memory space cost, which in itself can be high depending on the number of stored properties and scene complexity.

As memory space was no issue in the thesis work's context, we chose to take the first option. Due to its apparent similarity in concept to the **deferred shading** description given in section 3.2.2, we also chose to call the intermediate buffer in which we store the material properties the **G-Buffer**. This is the naming we will use for that buffer from now on. The G-Buffer in our implementation consists of a singular buffer with the same texel dimensions (width and height) to that of the light map.

At this point we are ready to summarize the overall structure of the object-space lighting algorithm in the following pseudocode:

Rasterization :

```
foreach triangle  $\mathbf{T} \in \text{Mesh}$ 
  foreach texel  $\mathbf{G} \in G\text{-Buffer}$ 
    decide if texel  $\mathbf{G}$  is covered by  $\mathbf{T}$ 
    if  $\mathbf{G}$  is covered:
      store material properties to  $\mathbf{G}$ 
```

Shading :

```
foreach texel  $\mathbf{L} \in \text{LightMap}$ 
  if corresponding texel  $\mathbf{G} \in G\text{-Buffer}$  is filled
    perform shading for  $\mathbf{L}$  using  $\mathbf{G}$ 
```

As a practical consideration, a given mesh's associated vertex and index data need to be stored into separate vertex and index buffers in a Unity script. This is because Unity is primarily set up towards screen space rasterization-based 3D rendering using the graphics pipeline. Thus, alternative rendering methods need to implement some of the basic rendering utilities including mesh geometry handling by themselves. A pseudocode version of the Unity script handling the execution of the different compute shader programs is given in Appendix A.

After the mesh data has been declared and any utilized texture maps (e.g., diffuse and normal maps) have been assigned for the script, the G-Buffer is created as a *read-writeable buffer* with the desired resolution for a rasterization compute shader program. As previously noted the resolution has to be the same as the one used for the lightmap.

When deciding on the suitable resolution it has to be kept in mind that after shading in texture-space to a light map we have the opportunity to use the **hardware supported anti-aliasing** functionality intended **for texture sampling** covered in 2.2.4. This means that we get the benefit of bilinear filtering for magnification, and anisotropic (and linear) filtering of mipmaps to handle minification. The resolution choice thus does not so much affect the visibility of aliasing artefacts (especially *temporal aliasing* based ones as those are minimized to a large extent by the texture filtering), but rather the sharpness at which we can represent fine surface detail.

On the performance side increased resolution leads to increases in memory and computation costs. In this thesis' context these were not significant limitations (the user study was to be done using pre-rendered videos), so

the G-Buffer and light map resolutions could be set to the **4096*4096** texel limit, which Unity can still reliably handle.

After the material properties have been rasterized in to the G-Buffer, another read-writeable buffer is created for the light map. This is used by the compute shader program for shading. The shading program has to be executed every time we desire to have updated reflected radiance values for an object. In the thesis' implementation the update is performed for every frame buffer update (60 Hz for the user study videos), but in the general case it could be set to rates much lower than that. It could also be set to be dynamically based on scene properties such as the object-to-camera distance or the rate of change in lighting conditions or camera position. A simple *passthrough* shader program incorporating a vertex and a pixel shader was written to perform the basic object-to-clip space transformation for the mesh vertices and the sampling of the lightmap for the mesh surface, respectively.

The rasterization part of the algorithm is of notable complexity so it will be delved into in more detail in the next section.

4.3.5 The Edge Function

In order to fill the G-buffer texels with their appropriate material data, we need a way to identify the triangle a given texel is overlapped by. Additionally, as we desire to use material parameters that vary over the surface of a triangle, we also need a method to determine the exact location a texel's center lies on the triangle. Both of these goals can be achieved through the use of a **rasterization algorithm**, of which there exist a variety of.

For the thesis work we have chosen to implement the **edge function** method. As performance is not a key consideration, the choice was made based on the expected ease of implementation. The edge function method was originally presented by Pineda [44] and it is based on deciding on which of the two half-spaces formed by a triangle edge a test point (the texel center in this case) lies in. The properties of the vector cross product are used in the method as follows.

Let us have a triangle defined by the three 2D vertices **a**, **b** and **c**. Let us also name the edges defined by these vertices as **A** = **b** - **a**, **B** = **c** - **b** and **C** = **a** - **c**. The texel center point that we want to test is named as the 2D point **p**. Auxillary edges from the triangle vertices to **p** are denoted as **P_a** = **p** - **a**, **P_b** = **p** - **b** and **P_c** = **p** - **c**. Using the definition of the cross product for 2D vectors on the vectors **P_a** and **A** we have:

$$\mathbf{P}_a \times \mathbf{A} = (p.x - a.x)(b.y - a.y) - (p.y - a.y)(b.x - a.x) \quad (4.1)$$

If we only examine the magnitude of the cross product, we can express it in another useful form:

$$|\mathbf{P}_a \times \mathbf{A}| = |\mathbf{P}_a||\mathbf{A}|\sin\theta, \text{ where } \theta \in [0, \pi] \quad (4.2)$$

where θ is the angle between vectors \mathbf{P}_a and \mathbf{A} . The result of equation 4.2 is the same as the area for a parallelogram formed by \mathbf{P}_a and \mathbf{A} . Thus we can interpret equation 4.1 as giving us the signed area of the same parallelogram.

Knowing that the lengths of the sides of must always be non-negative, the only way we could retrieve negative values from the right side of 4.2 is by relaxing the angle θ to take on any real value.

If we additionally define θ to be a rotation angle defined by the right-hand rule (where positive rotation angles are counter-clockwise), by looking at the sign of the result from 4.1 we now have an idea on how to decide which side of the edge \mathbf{A} point \mathbf{p} is situated in. In the cross product the angle is interpreted as starting from the first operand vector to the second one, so with the right-hand rule the sine values for angles in the range $[0, \pi]$ result in positive values, while angles in the range $[\pi, 2\pi]$ result in negative values. By computing the signs for the cross product for $\mathbf{P}_a \times \mathbf{A}$, $\mathbf{P}_b \times \mathbf{B}$ and $\mathbf{P}_c \times \mathbf{B}$, we can decide on whether the point \mathbf{p} lies inside the triangle.

The order of the cross product operands is important from a geometric perspective, as is also reflected by the fact that cross product is an anti-commutative operation. This means that both an interpretation for the sign of its result as well as the operand order has to be decided and then strictly adhered to. For the thesis work the first operand has been decided to be the auxillary vector from the edge vector starting vertex to the sample point p , with the current triangle edge vector being the second operand. **Positive signs from equation 4.1 are taken to mean that the sample point lies on the "inside" half-space defined by a edge, while negative sign tells us that the point is in the "outside" half-space.**

An additional, practical consideration for the operand ordering is that the order can be reversed due to the triangle's **winding order**. The winding order denotes the order in which the triangle vertices are defined, with the options being either clockwise or counter-clockwise ordering. **In our implementation we expect the vertices to be defined in clockwise winding order.**

We can now present the final edge test function. Let us have an edge defined by its end points \mathbf{a} and \mathbf{b} , and let us set $\mathbf{D}_y = \mathbf{b} \cdot \mathbf{y} - \mathbf{a} \cdot \mathbf{y}$, and $\mathbf{D}_x = \mathbf{b} \cdot \mathbf{x} - \mathbf{a} \cdot \mathbf{x}$. Then we can express the edge function for a sample point (\mathbf{x}, \mathbf{y}) as:

$$EdgeTest(x, y) = (x - a.x)D_y - (y - a.y)D_x \quad (4.3)$$

Changing the end points appropriately allows us to test the sample point against all of the triangle edges.

We now have a method to test if a sample point (i.e., the texel center) is inside a triangle. In the common case the any given triangle defined in the UV space will only cover a small portion of a texture map such as the G-buffer, so it would be wasteful to perform the test for against each and every texel center. We perform the simple optimization of determining the **AABB** (*Axis Aligned Bounding Box*) of a given triangle first by computing the minimum and maximum texel coordinates in both the width and height dimensions. Thus we arrive at a rectangle which tends only a portion of the entire texture map, leading to a decreased number of texels needing to be tested. The AABB as well as the edge function are illustrated in figure 4.1. An example of a triangle rasterized by testing texel centers within its AABB against the edge function for each of the triangle's edges is depicted in figure 4.2.

As mentioned before, the cross product's (and thus also the edge test function's) result can be interpreted as the signed area of the parallelogram defined by the operands. When this value is divided by two, we get the area of the corresponding triangle. This means that if we use the triangle vertices **a**, **b** and **c** we have the following equation:

$$TriangleArea(\mathbf{a}, \mathbf{b}, \mathbf{c}) = \frac{EdgeTest(c.x, c.y)}{2} \quad (4.4)$$

Note that in our implementation we only use this function when EdgeTest returns a non-negative value, so this relation holds. The triangle areas for the sub-triangles defined by the texel sample point and the two triangle edge vertices (that we use in EdgeTest to test against) can also be calculated. **These sub-areas when divided by the complete triangle's area result in values which represent for a given sample point the "closeness" or weighting towards a corresponding triangle vertex.** This corresponding triangle vertex is always the vertex which was not part of the EdgeTest. As an example we give the calculation for a sample point **p**'s weight towards the triangle vertex **c**:

$$\omega(\mathbf{p}) = \frac{TriangleArea(\mathbf{a}, \mathbf{b}, \mathbf{p})}{TriangleArea(\mathbf{a}, \mathbf{b}, \mathbf{c})} \quad (4.5)$$

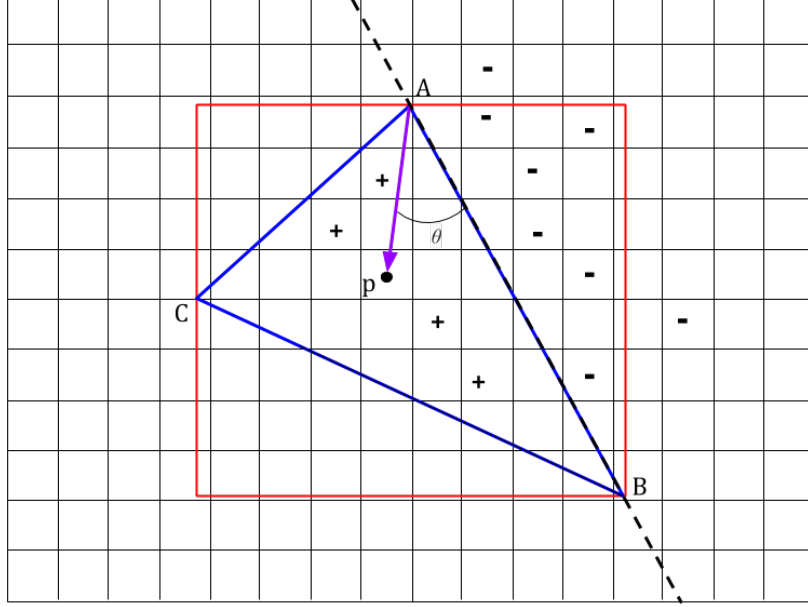


Figure 4.1: The edge function return positive values only when \mathbf{p} is located in the positive half-space formed by the edge we are testing against. By performing this test for each of the triangle edges, we can decide if \mathbf{p} lies inside the triangle. Using the AABB (drawn in red) we can greatly reduce the number of texels we perform this test for.

We can gain sample point \mathbf{p} 's weights towards the triangle vertices \mathbf{a} and \mathbf{b} in the similar way. In mathematics these weights are commonly referred to as **barycentric coordinates**. These weights are important for gaining interpolated vertex attribute values for points inside the triangle. These can be any of the vertex attribute values including vertex normal or position. By using the weights we in effect know how much an inner triangle point's attribute should be effected by the corresponding attribute values in each of the triangle's vertices. The interpolation for a sample point \mathbf{p} is performed by the following function:

$$X_{interpolated}(\mathbf{p}) = \omega_a X_a + \omega_b X_b + \omega_c X_c \quad (4.6)$$

where the weights have been calculated for \mathbf{p} according to equation 4.5. All

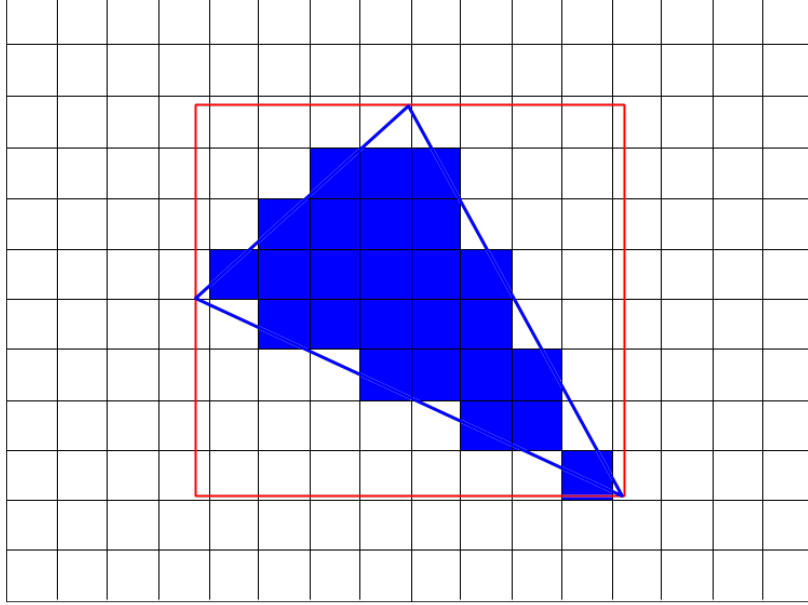


Figure 4.2: An example triangle rasterized into a texel grid by testing the texel center positions against the edge function.

of these weights sum up to 1, accordingly to the triangle area definition:

$$\omega_a + \omega_b + \omega_c = 1 \quad (4.7)$$

An example of the results from the rasterization method described here can be seen in figure 4.3. A more thorough pseudocode version of the rasterization implementation is given in Appendix B.

4.3.6 Conservative Rasterization

Unfortunately, the way in which we test whether a triangle covers a texel by only testing against the texel center is susceptible to underestimation, as could be seen in figure 4.2. This means that the method fails to detect cases where other parts of a texel might be covered but the texel center itself is not. For our texture space shading purposes this leads to the end result exhibiting notable dark "cracks" or "seams" in places where the mesh's triangles have not been properly rasterized into the G-buffer.



Figure 4.3: A model of a statue, where the reflected radiance has been rasterized onto the light map – a texture covering the entire model surface. Note the dark “seam” artefacts especially visible around the neck and armpit areas of the model (but present throughout) due to the non-conservative rasterization approach.

In order to avoid this, we perform an additional rasterization pass after the one described previously, but this time we use an overestimating conservative rasterization approach. The algorithm we use follows the one outlined by Hasselgren et al in [16] for use in the conservative rasterization of triangles from three dimensions into the two dimensional screen coordinates, but with very small adjustments the algorithm also works with our UV space triangles.

The idea is to form an enlarged version of the triangle-to-be-rasterized by moving the triangle edges a half texel amount to a given edge’s normal direction. Note that as the triangle vertices are defined in UV coordinates (in the range $[0, 1]$ for both dimensions), while the texels are defined in texture space (in the range $[0, \text{textureWidth}]$ for width and $[0, \text{textureHeight}]$ for

height), we need to calculate the conversion multiplier between them. This is done simply:

$$\mathbf{halfTexel}.uv = \left(\frac{0.5}{\text{textureWidth}}, \frac{0.5}{\text{textureHeight}} \right)$$

The moving of the triangle edges is a more complex procedure. We first start by using a homogenous 3D representation for the triangle vertices. The change to 3D is made so that the calculation of the edge intersection points would be easier. Taking the UV coordinate vertices to homogenous 3D is performed by adding a third, w-coordinate which we set to the value of 1:

$$\begin{aligned}\mathbf{a}_h.uvw &= (a_u, a_v, 1) \\ \mathbf{b}_h.uvw &= (b_u, b_v, 1) \\ \mathbf{c}_h.uvw &= (c_u, c_v, 1)\end{aligned}$$

where \mathbf{a} , \mathbf{b} and \mathbf{c} are the original triangle's vertices. We can now represent the triangle edges as planes in homogenous 3D by performing cross products with the plane's defining vectors:

$$\begin{aligned}\mathbf{plane}_a.uvw &= (b_h - a_h) \times \mathbf{a}_h \\ \mathbf{plane}_b.uvw &= (c_h - b_h) \times \mathbf{b}_h \\ \mathbf{plane}_c.uvw &= (a_h - c_h) \times \mathbf{c}_h\end{aligned}$$

With this formulation the plane variables now contain the plane normal coordinates in the u and v components while the w component holds the plane's distance from origin. Knowing this and additionally that the homogenous planes correspond to the non-homogenous triangle edges, increasing the w component's value is in effect the same as enlargening the triangle by moving it's edges towards the edge normal directions. The appropriate amount to shift the planes is controlled by the `halfTexel` value:

$$\begin{aligned}\mathbf{plane}_a.w &= \mathbf{halfTexel}.uv \cdot \text{abs}(\mathbf{plane}_a.uv) \\ \mathbf{plane}_b.w &= \mathbf{halfTexel}.uv \cdot \text{abs}(\mathbf{plane}_b.uv) \\ \mathbf{plane}_c.w &= \mathbf{halfTexel}.uv \cdot \text{abs}(\mathbf{plane}_c.uv)\end{aligned}$$

where the *abs* function return the absolute value of its argument. After doing this, we can use the shifted planes to compute the intersection edges, which correspond to the non-homogenous enlarged triangle's vertices. The intersection edges are found as follows:

$$\begin{aligned}
\mathbf{intersection}_a.uvw &= \mathbf{plane}_a \times \mathbf{plane}_b \\
\mathbf{intersection}_b.uvw &= \mathbf{plane}_b \times \mathbf{plane}_c \\
\mathbf{intersection}_c.uvw &= \mathbf{plane}_c \times \mathbf{plane}_a
\end{aligned}$$

From which we gain the non-homogenous intersection points – or in our case the enlarged triangle’s vertices, as follows:

$$\begin{aligned}
\mathbf{a}_{enlarged}.uv &= \mathbf{intersection}_a.uv / \mathbf{intersection}_a.w \\
\mathbf{b}_{enlarged}.uv &= \mathbf{intersection}_b.uv / \mathbf{intersection}_b.w \\
\mathbf{c}_{enlarged}.uv &= \mathbf{intersection}_c.uv / \mathbf{intersection}_c.w
\end{aligned}$$

These are the coordinates we can use as previously as input for the *EdgeTest* function in equation 4.3. Rasterizing this enlarged triangle directly can lead to false positives where some of the texels covered by the enlarged triangle would not be overlapped at all by the original triangle. This situation can be avoided by changing the way we compute the AABB we previously noted to be using as a rasterization optimizing measure in section 4.3.5. As the AABB controls which texels we even consider to be covered by a triangle, we can simply limit it to a size beyond which no coverage is possible. In practice this means computing the AABB minimum and maximum values in the following way:

$$\begin{aligned}
\mathbf{ConservAABB}_{Umin} &= \max(0, T_{Umin} - \mathbf{halfTexel}.u) \\
\mathbf{ConservAABB}_{Vmin} &= \max(0, T_{Vmin} - \mathbf{halfTexel}.v) \\
\mathbf{ConservAABB}_{Umax} &= \min(\mathbf{textureWidth}, T_{Umax} + \mathbf{halfTexel}.u) \\
\mathbf{ConservAABB}_{Vmax} &= \min(\mathbf{textureHeight}, T_{Vmin} + \mathbf{halfTexel}.v)
\end{aligned}$$

Conservative rasterization is illustrated in figure 4.4 and an example of the results is shown in figure 4.5. A more thorough pseudocode version of the implementation is given in Appendix C.

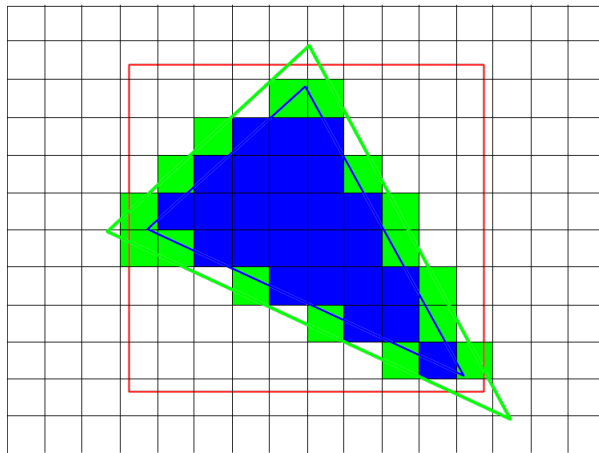


Figure 4.4: The previously depicted triangle rasterized using the conservative rasterization approach. The conservatively filled texels are colored in green, while the non-conservatively filled texels are in blue.

4.3.7 The Structure of a G-Buffer Texel

The values we store in the G-buffer texels consist of those we need for the shading phase of the object space lighting algorithm, as well as auxiliary helper variables during the rasterization and the dilation process (discussed in section 4.3.9). The G-buffer texel structure is presented in the following list:

- float3 **weights**
- float3 **localNormal**
- float4 **localTangent**
- float3 **normalSample**
- float3 **diffuseAlbedo**
- float3 **specShape**
- float3 **auxVal**
- int **triangleIndex**



Figure 4.5: Here we have used conservative rasterization, with the texels rasterized conservatively into being displayed in red. Although the result is improved over that of figure 4.3, a very close examination reveals that there are still very miniscule but visible "seams" present. This is a product of the texture filtering performed on the light map and is discussed in section 4.3.9.

- bool **isFinal**
- bool **isDilated**

The **weights** vector holds the barycentric weights for a given texel center's location on the corresponding triangle denoted by the **triangleIndex** integer, which holds the triangle's index in the index buffer. The **localNormal** and **localTangent** vectors are used to form a reference coordinate system for the **normalSample** surface normal vector. We use these vectors as part of the *normal mapping* technique in shading first mentioned in section 2.2.3. Normal mapping will be covered in more detail in the next section. **DiffuseAlbedo** is the RGB vector capturing the proportion of diffusely reflected light for the surface point as described in section 2.2.1, while the

specShape and **auxVal** vectors are additional optional vector slots we can opt to use for storing material data that may change from source model to another. Finally, we have the **isFinal** and **isDilated** boolean flags which we use signify whether a texel has been finished being rasterized or getting dilated values from its neighbors, respectively.

4.3.8 Shading and Normal Mapping

After the G-buffer’s rasterization is complete, we can run a compute shader program to perform shading for each light map texel using the material information in the corresponding G-buffer texel. We utilize two different reflectance functions to perform the shading: the primary one is the Cook-Torrance BRDF we described in detail in section 2.2.6 while the secondary one is a minor modification of the SVBRDF (*Spatially-varying BRDF*) implementation by Knuuttila [27] who used two-shot SVBRDF capture material data from Aittala et al [3]. Video material based on the renderings from both functions are utilized in the user study.

Although the SVBRDF will not be described in detail in this thesis, it should be noted that the main differences compared to the Cook-Torrance BRDF relate to the use of specular shape information generated from real-world data using a method outlined by Aittala et al. [3]. This is used to provide reflectance that more closely resembles that of the real-world counterpart material. Compared to the Cook-Torrance BRDF, the normal distribution function is altered while the geometry function is discarded completely. The Fresnel function remains identical to the one defined in the Cook-Torrance BRDF.

An implementation detail that we have not yet covered, but is a vital component to modern 3D renderers, is the **normal mapping technique**. In section 2.2.3 we discussed how storing surface normal vectors in texture maps can allow us to store higher frequency surface orientation information, when compared to using primitive or vertex normals. In practice, however, we have to remember that vectors are only meaningful when in relation to some specific coordinate system. The surface normals stored in texture maps are commonly declared relative to what is called a **tangent space**. The tangent space is simply a coordinate system that captures the geometric orientation of the surface. As it is a 3D coordinate system, it is formed by three basis vectors: **the tangent, normal and bitangent vectors**. Accordingly, this coordinate system is often called the **TBN basis**.

The tangent and bitangent vectors are the vectors deciding the surface plane, while the normal vector denotes the direction ”outwards” from the surface plane. The normal vector in this case is the vertex normal interpo-

lated for a particular G-buffer texel using the covering triangle fragment's barycentric weights and the vertex normals stored in the triangle's vertices. It can be thought of as representing the general geometric outwards facing direction of a surface. The normal vector along with the tangent vector is typically stored in a given mesh's vertex data, but functionality exists in Unity to generate both of them programmatically when needed.

The bitangent vector can be generated in a shader program by taking the cross product of the normal and tangent vectors, so it typically not stored in the per vertex data. For cases where we are using screen-space shading, care must be taken in deciding where to perform the TBN basis construction as correct (although slower) results can only be gained in the pixel shader. As the result vector of a cross product operation depends on the cross product operands' order, the developer also has to be mindful of whether a right-handed or a left-handed coordinate system is used.

After we have constructed the TBN basis we are ready to progress to lighting computations. We perform our object-space lighting in the object-local coordinate space, so in addition to transforming the virtual camera location and light source direction/position (depending on if we are using directional or point light sources, respectively) to the object-local space, we need to do the same for the normal vector sampled from a normal texture map. For the G-buffer we sample texture maps such as the diffuse albedo and normal maps using bilinear filtering. This provides a preferable approximation for the correct value per texel, when compared to results we would get from using a simple point sampling method. Mipmapping is not applicable for use in the rasterization compute shader programs as we have no information of the projected sizes of each texel onto screen pixels. There would be little benefit from using anything other than the highest resolution version of a texture anyhow, as the rasterization step is performed only during the initial program start-up and the highest resolution version provides also the highest data frequency to use as input for the bilinear filtering.

Returning to the particular topic of **normal map sampling**, Unity changes the data type format for texture maps declared as holding normal data into its own compressed format. This means a decompression operation has to be performed before transforming the sampled normal to the object-local space. Unity provides this operation readily available for screen-space shading vertex and pixel shader programs, but for our compute shaders we need to add that function manually:

$$\begin{aligned} \mathbf{N}_{\text{uncompressed.xy}} &= \mathbf{N}_{\text{compressed.wy}} * 2 - 1 \\ \mathbf{N}_{\text{uncompressed.z}} &= \sqrt{1 - \max(0, \min(\mathbf{N}_{\text{uncompressed.xy}} \cdot \mathbf{N}_{\text{uncompressed.xy}}))} \end{aligned}$$

After this, we are ready to perform the change of coordinate system transformation on the sampled normal to take it from tangent space into the object-local coordinate system. When using the *column major* matrix format, the tangent space's TBN basis vectors (expressed relative to object-local space) can be placed as the rows of a transformation matrix:

$$\mathbf{M}_{TBN} = \begin{bmatrix} T.x & T.y & T.z \\ B.x & B.y & B.z \\ N.x & N.y & N.z \end{bmatrix}$$

This matrix – when used to multiply a 3D column vector on its right, **is equal to transforming from object-local space into the tangent space defined by the TBN basis vectors T, B and N.** We desire to perform the exact inverse of this transformation, so we need to compute the inverse matrix \mathbf{M}_{TBN}^{-1} . As the basis vectors are orthogonal to each other and of unit length, \mathbf{M}_{TBN} is thus an orthogonal matrix. For an orthogonal matrix the inverse is simply its transpose. In \mathbf{M}_{TBN} 's case this means that the inverse is:

$$\mathbf{M}_{TBN}^{-1} = \mathbf{M}_{TBN}^T = \begin{bmatrix} T.x & B.x & N.x \\ T.y & B.y & N.y \\ T.z & B.z & N.z \end{bmatrix}$$

By using this matrix we transform the sampled normals into object-local space – the same coordinate system as the light source direction/position and virtual camera position vectors we use for the reflectance computation. Following this, we have all the needed material information rasterized into the G-buffer in the correct form for the Cook-Torrance and SVBRDF lighting calculations. Once the lighting calculations have been performed, we have the light map texture containing reflected light values as RGB vectors for the entire surface of a mesh.

4.3.9 Dilation

While the use of overestimating conservative rasterization (section 4.3.6 ensured that every even partially overlapped G-buffer texel gets filled in and (later on) shaded, there still exists a significant problem. As the distance of the 3D mesh's surface to the virtual camera increases, the brightness of the reflected light appears to decrease. This is caused by how mipmapping works. As the lower resolution versions of a texture are computed by averaging texel values over a certain rectangular area, it is possible for this area to contain texels which have no proper mapping to the mesh triangles.

In our case this would correspond to texels to which no triangle gets rasterized on to. As these texels contain no material data, they would be shaded to the default value of 0 for each RGB component. In other words they would appear black. When a mipmap's averaging area contains these null, black texels, the average for the whole area is weighted towards black. The amount of this weighting is related to the number of null texels contained by an averaging area.

The error can be corrected by performing **edge padding**, in which we spread the material data of the texels with actual rasterized triangles (and thus, material property values) to their neighboring empty texels. In image processing this operation is commonly referred to as **dilation**. In our implementation dilation is performed a set number of times determined to be sufficient for filling every G-buffer texel with valid material data. The dilation operation is performed as follows:

Let the group **G** of G-buffer texels be divided into two non-intersecting groups: **F for finalized texels** holding their final material values, and **D for non-finalized texels** waiting for the dilation operation.

```

foreach texel  $d_i \in \mathbf{D}$ :
    numNeighbors = 0
    foreach neighboring texel  $n_j \in \mathbf{F}$  of  $d_i$ :
        foreach interpolated material property of  $d_i$ :
             $d_i.matVal_j += n_j.matVal_j$ 
        foreach non-interpolated material property of  $d_i$ :
             $d_i.matVal_k = n_j.matVal_k$ 
        numNeighbors = numNeighbors + 1
    if numNeighbors > 0:
        foreach interpolated material property of  $d_i$ :
             $d_i.matVal_j = d_i.matVal_j / \text{numNeighbors}$ 
        remove  $d_i$  from D and add it into F

```

We go over each of the non-finalized texels in the G-buffer and set as their material property values the (uniformly) weighted sum of the their neighboring texels' corresponding material property values. The neighboring texels are only considered if they contain finalized material values either from the previous rasterization process, or by having gained them through dilation. This ensures the propagation of only proper material values – we do not want empty texels to have any influence. Note that in our G-buffer structure (section 4.3.7) some properties, like the *triangleIndex* and *weights* values, cannot be computed through the weighted sum as this would not yield correct results. This is what we mean by the *non-interpolated* material properties in the previous pseudocode. These values are simply set to those of the last finalized neighboring texel's values in the loop over neighbor texels. An example of dilation results (along with non-conservative and conservative rasterization results) is shown in figure 4.6.

After the dilation has been performed, Unity's built-in mipmap generating function is used to construct a mipmap chain for our light map, after which it is ready for use in texturing a mesh. This is done by performing a forward rendering pass for a given mesh, where in the vertex shader we simply perform the projection from object-local coordinates into clip space coordinates for each of the mesh's vertices. In the pixel shader program we perform 16x anisotropic filtering to sample the lightmap using the projected fragment's UV coordinates. The anisotropic filtering provides us with a hardware-supported anti-aliasing method to smoothen the gradation between the colors of neighboring surface fragments, while also eliminating visible shimmering artefacts that would be associated with a traditional screen-space shading method. We also avoid the problems arising from the filtering of normal maps in screen-space shading (discussed in section 2.2.5) by only performing the filtering on the final radiance values.

Returning briefly to the particular topic of dilation, in hindsight it would have been more efficient to simply first perform the shading of the covered light map texels and afterwards dilate the resulting radiance values to neighboring texels, as this can result in a considerable saving in lighting computation costs with the added per-frame dilation being minor in comparison. This should be kept in mind for any further developments.

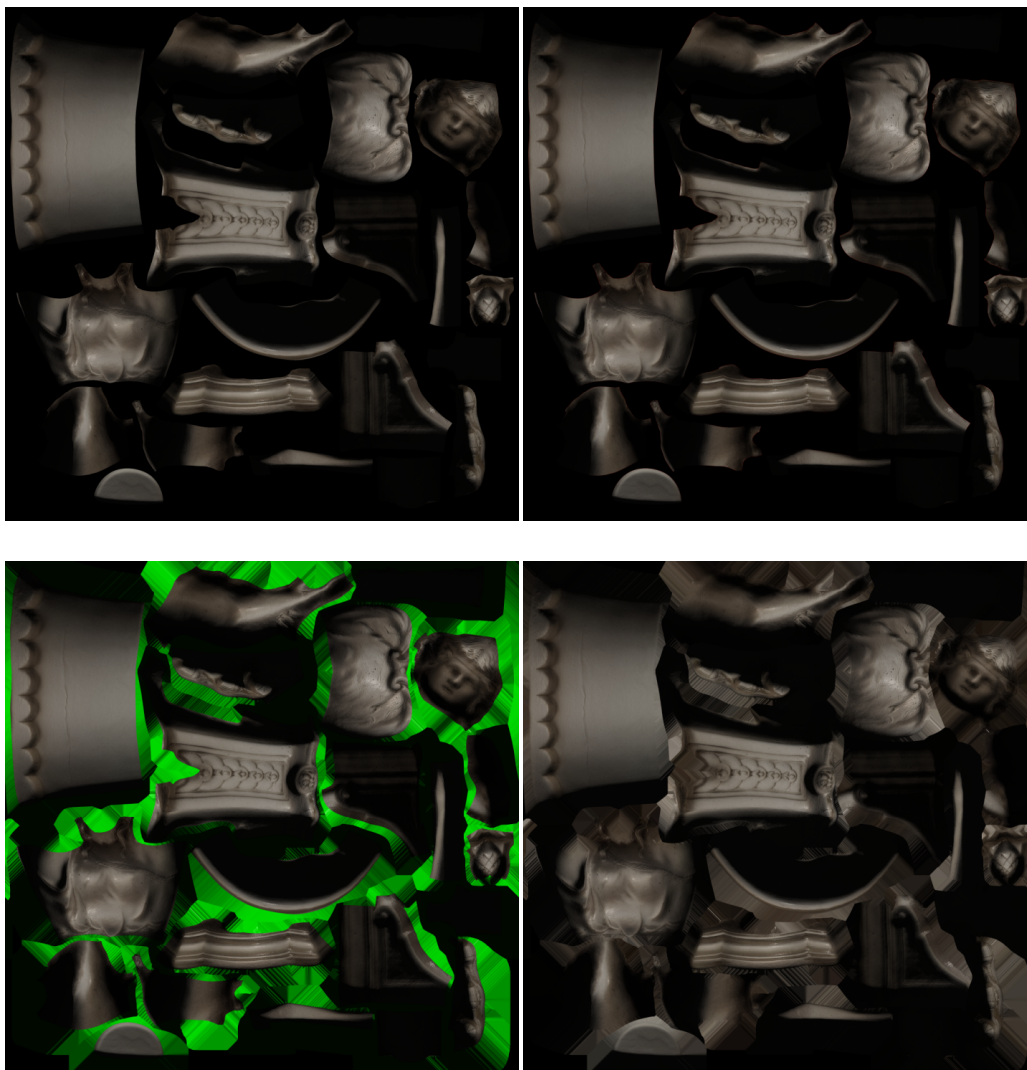


Figure 4.6: A review of the light map texture after different forms of rasterization and dilation. The description for the images is as follows: non-conservative rasterization (top left), conservative rasterization visualized using red diffuse albedo (top right), dilation visualized with green diffuse albedo (bottom left), dilation with the actual diffuse albedos for the model being used.

4.3.10 Screen-Space Shading Implementation

As part of the thesis work, screen-space shading versions of the Cook-Torrance BRDF and the SVBRDF were also implemented. These versions served two main purposes: to be utilized as comparison rendering methods in the user

study portion of the thesis work, and to additionally provide a ground truth comparison assessing the object-space lighting implementation’s correctness during development. Any possible errors in basic rendering tasks such as in the transformations between different coordinate systems could in this way be more quickly identified. The justification for using the screen-space shading versions as ground truth equivalents is given in the following reasoning.

Provided that the used lightmap resolution is high in relation to that of the screen area covered by a projected mesh, the texture-shaded and anisotropically filtered version should be visually close to the results gained from a supersampled screen-space shaded version. This is because anisotropic filtering uses mipmaps which themselves are (with the exception of the original texture) versions where the texel values have been generated through an averaging process from nearby texel values in a higher resolution version. **In other words, each texel is derived through the use of multiple sample values.**

Anisotropic filtering on the material texture maps is traditionally also used in applications which perform their shading computations in screen-space to minimize the most visible aliasing artefacts. However, this form of subjecting already the shading inputs to pre-filtering can result in final reflected radiance values that diverge more from the actual ground truth values, when compared to what we would get if we were to perform the filtering only on the final radiance values [42]. This means that our texture-space shading implementation provides results that are closer to reality than the ones gained from these more traditional, screen-space techniques. In order to directly filter the radiance values on these techniques, one needs to operate on the final back buffer color values to perform a similar averaging process as that is where the radiance values are stored.

This equals to performing supersampling in the screen-space shading method, where we render a mesh using a intermediate raster grid resolution that is higher than the final output resolution, e.g. the full screen display resolution. In this case the final output pixel values are each formed through a weighted sum of multiple intermediate raster grid’s pixel values. **This means that for a render of a given 3D mesh, the theoretical expectation is for a supersampled screen-space shaded image in which the shading inputs have only been nearest-neighbor filtered to look very similar to a non-supersampled texture space shaded, anisotropically filtered image.** It bears noting that the mesh geometry’s outline will look different in these cases – as in the texture-space shaded version we only gain anti-aliasing on the inner parts of the mesh surface and none on the primitive geometry edges. The theoretical expectation is also one that we test for in the user study portion of this thesis.

The actual screen-space shading implementation follows the traditional flow for the technique, as was described in chapter 3. Following a draw call for a 3D model, we first execute its vertex shader on its triangle vertices. In the vertex shader we transform the position, normal and tangent vectors into world space – ready for the vertex attribute interpolation during fragment rasterization. The transform to world space is done for these attributes as we deemed that to be most practical space to perform the later lighting computations.

In addition to this, we also simply pass through the texture coordinates (we perform no texture animations) and produce a copy of the vertex position that has been transformed into clip space. As covered in section 2.1.3, this is the space the GPU expects the vertex position to be expressed in relation to before applying the perspective projection.

After the vertex shader operations, triangle fragments are operated on in the pixel shader. Here we perform three main tasks:

- Sampling of material information held in texture maps
- Construction of the tangent-to-world space matrix \mathbf{M}_{TBN}
- Shading through the application of a reflectance function

It is worth noting that the transformation matrix \mathbf{M}_{TBN} here is the transformation from tangent-to-world space, unlike the similarly named matrix we described in section 4.3.8. In that section the matrix was a transformation from tangent-to-local space.

The sampling of material information from texture maps is for the most part simple as we just use the HLSL provided sampling function, but here we have to be mindful of what exactly we are striving to achieve. In a typical case, to minimize the shimmering artefacts caused by the possibly abrupt changes in values held from texel to texel, texture filtering techniques for the minification and magnification cases are used as described in section 2.2.4. However, for our purposes we want to use the screen-space shading implementation as a means to compare against the object space lighting implementation. In the latter we perform the shading on individual texels containing the precise material information for that particular surface location of a mesh. It is only afterwards that we use the texture filtering on the completed light map to provide anti-aliasing.

If we were to use any other texture filtering method than the *point sampling* one in the screen-space shading implementation, it would mean that we would be using pre-filtered material information as input for the reflectance function. This would be in contrast to the exact values we use in object-space

lighting. For this reason we forego the use of the more advanced texture filtering options in the screen-space shading implementation and instead limit ourselves to point sampling.

The construction of the matrix \mathbf{M}_{TBN} also calls for some consideration, as we have to be careful to perform it in the pixel shader. Although we transform the vertex normal and tangent vectors to world space in the vertex shader, it would be incorrect to build \mathbf{M}_{TBN} there. This is because most of the values output by the vertex shader, including the TBN basis vectors, need to be interpolated during rasterization before their use in the pixel shader. In the general case, this interpolation will not preserve the orthogonality of the TBN basis vectors, thus rendering the vectors to be linearly dependent to each other.

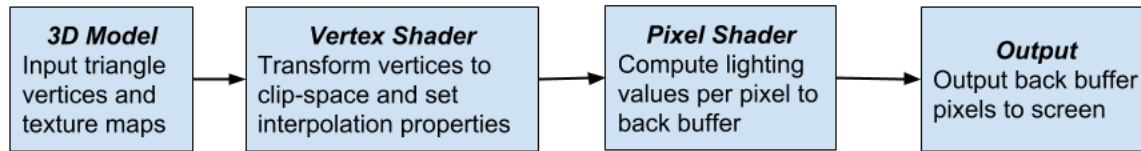
To avoid this, we perform the TBN construction in the pixel shader, with the application of the Gram-Schmidt process to orthonormalize the tangent vector with regards to the vertex normal vector. The bitangent vector is then calculated by taking the cross product of the vertex normal and tangent vectors. Unity provides a sign variable with the tangent vectors that we use to multiply the result of the cross product to ensure that the bitangent vector we get is facing the correct direction according to the selected handedness.

The shading through the use of a reflectance function is performed last in the pixel shader. Depending on the exact scene, we use one of the reflectance function previously mentioned (the Cook-Torrance BRDF and the SVBRDF) to compute the final shaded pixel colors, with the light sources ranging from a singular directional light to three point lights.

4.3.11 Summary Diagram of Texture-Space and Screen-Space Implementations

For ease of comparison, we present in figure 4.7 the high-level functional flows of the screen-space and texture-shading techniques we have implemented. Note that even though final radiance values are computed in our texture-space shading implementation for every screen update, as we were not investigating the temporal decoupling properties, in the general case it by no means has to be so.

Screen-Space Shading



Texture-Space Shading

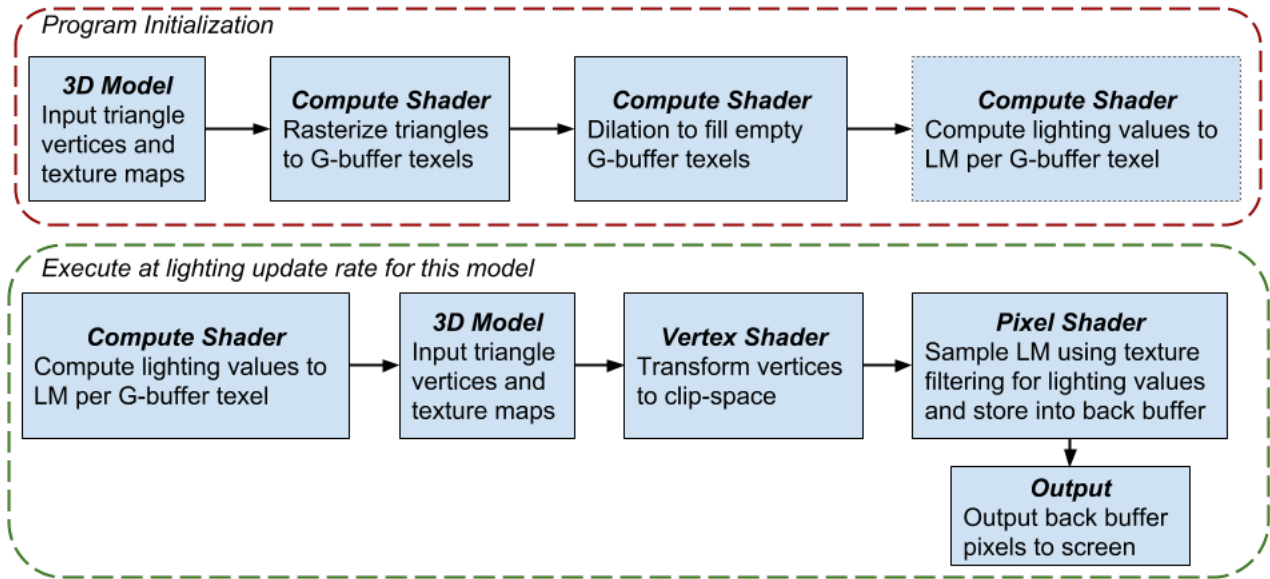


Figure 4.7: Diagram comparing the two shading algorithms.

4.3.12 Comparison of Texture-Space and Screen-Space Shaded Results

A comparison between the results from the screen-space shaded and texture-space shaded techniques can be seen in figure 4.8:

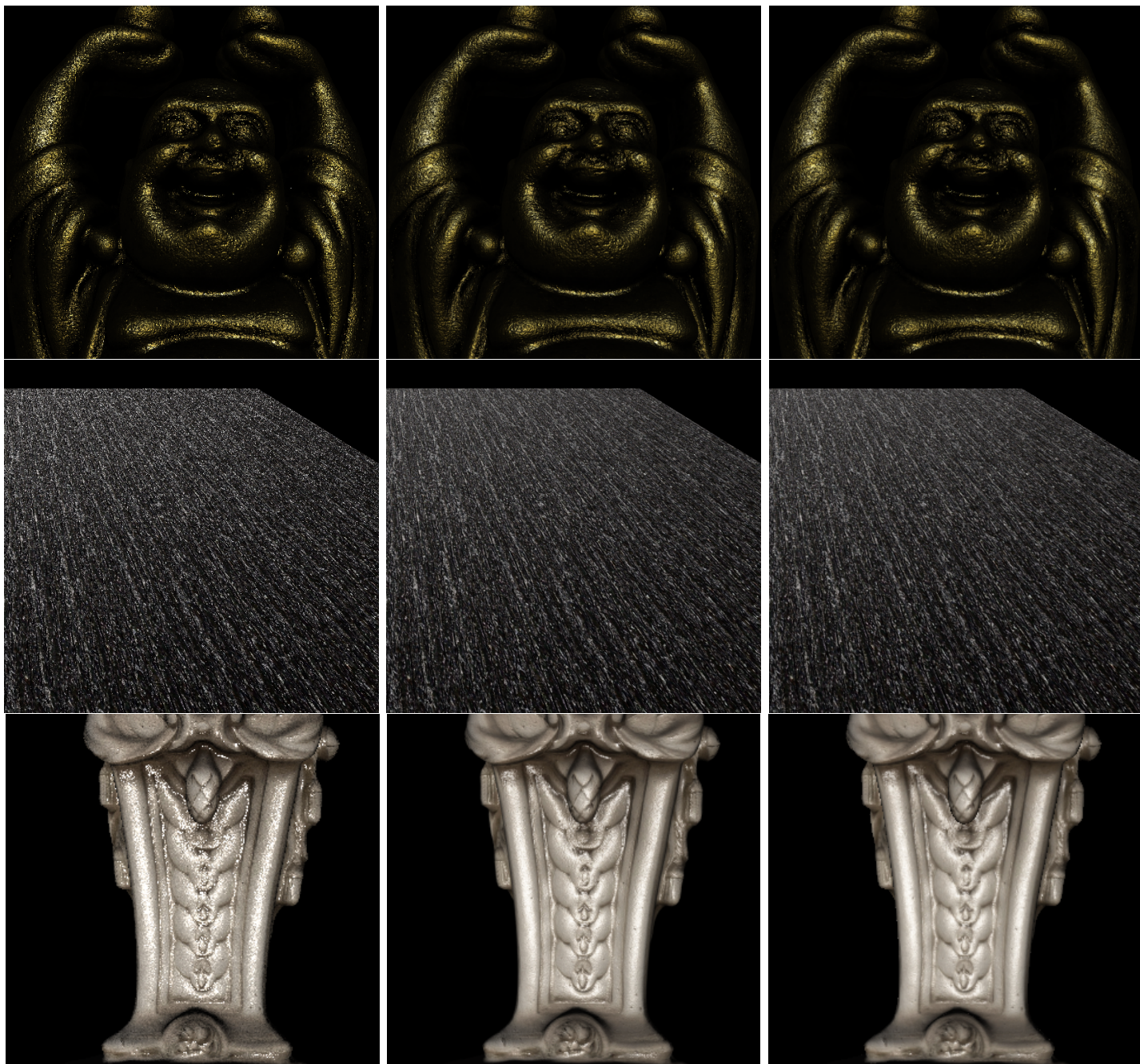


Figure 4.8: On the left-hand side we have the 1 sample per pixel (SPP) screen-space shaded images, in the middle we have the 256 SPP screen-space shaded images and on the right we have the texture-space shaded images. For all of these the screen resolution was 1920×1080 pixels, while the light map resolution for the texture-space shading was 4096×4096 texels.

From figure 4.8 it is easy to see how the 1 SPP screen-space images display noisy specular highlights on the models, which results in a appearance suggesting that the highlights could be shimmering. The supersampled 256 SPP screen-space and the texture-space shaded counterpart images in comparison have notably smoother highlights. These differences are caused by specular reflections being very sensitive to the relationship between surface microscale orientation and the view and light source directions. Small changes in these properties' values can lead to almost binary behaviour in specular reflections, causing sharp brightness changes spatially from pixel to pixel. This problem is not present in the 256 SPP and texture-space shaded images as they both benefit from taking a large number of shading samples and then low-pass filtering these samples for the final screen pixel color values. We have illustrated the differences and their cause in figure 4.9

A very close inspection of the 256SPP and texture-space shaded images reveals jagged patterns on object edges in the texture-space shading versions. This is most notable when comparing the renderings of a statue on the bottom row. This difference is due to the texture-space shading implementation not using any form of geometry anti-aliasing, meaning that we only retrieve the shading result from the Light Map for only one triangle per screen pixel. As the 256SPP screen-space version supersamples also the geometry, it enjoys demonstrably cleaner end results.

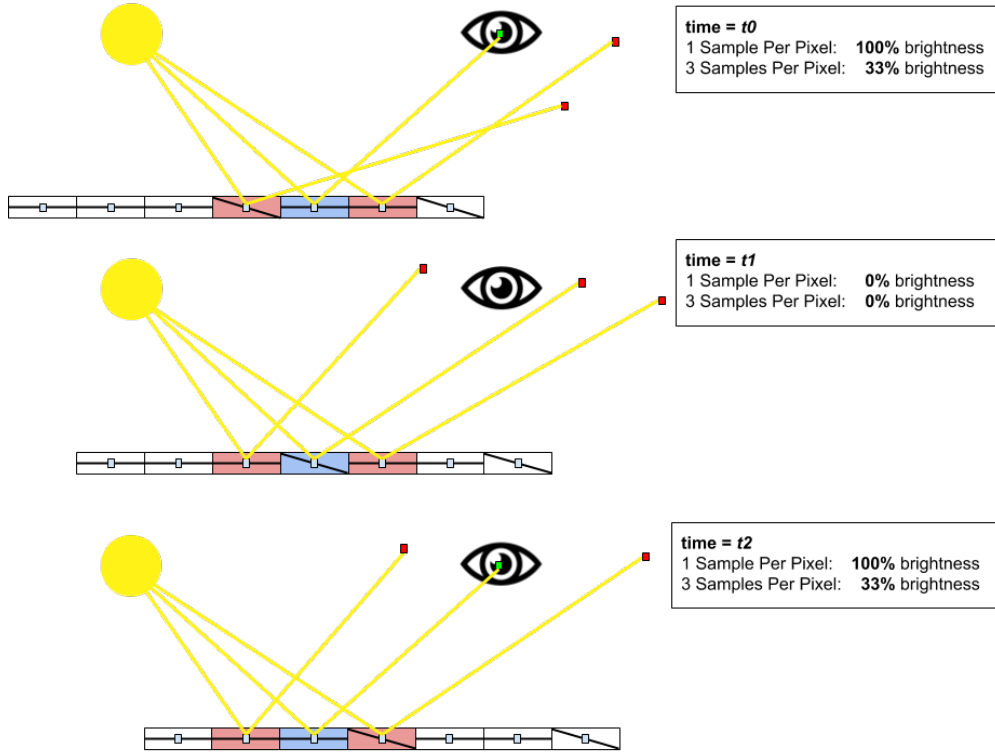


Figure 4.9: Illustration of the shading result differences for specular reflections, depending on if we take 1 or 3 samples per screen pixel. The colored rectangles in each time step case (t_0 , t_1 or t_2) indicate the surface area of a 3D model that is covered by a particular screen pixel. Each rectangle has its associated normal information inside it, which indicates the surface orientation. Of the rectangles, the one colored light blue (and situated in the middle) is the only one that is used when taking 1 sample for the screen pixel, while when we are taking 3 samples per pixel we use the surrounding light red rectangles for our sampling. From the illustration we can see that as the surface moves over the time steps (t_0 to t_2) to the right across the screen pixel, the brightness values for the final pixel color varies less dramatically when we take multiple samples per screen pixel and average them, compared to when we only take a single sample.

Chapter 5

The User Study

A user study was conducted as part of the thesis work. The purpose of the study was to investigate possible preferences users might exhibit for computer generated images based on the shading method. The shading methods under consideration were the texture-space shading technique we implemented for the thesis and the more traditional screen-space shading method. The general algorithms for both of these were described in chapter 4. For the screen-space shading method we produced two variations: one in which we take only a single sample per screen pixel, and another in which 256 samples are taken per screen pixel and then averaged to form the final pixel color value. The latter variation bears similarity to the texture-space shading implementation as they both use several shading samples and filter the results to form the screen pixel colors.

The increased spatial sampling frequency means that we are able to sample high frequency patterns (such as those originating from parameter-sensitive glossy reflections) without as much information loss, while the filtering provides a way for us to take into account the possible variations in surface color across the area subtended by a screen pixel. The end result should thus not only display less spatial aliasing for static rendered images, but also less temporal aliasing provided that the rendered object's surface moves slowly across the pixels during a video sequence.

Recent research conducted by Waldin et al. [53] suggests that flickering in displayed images can guide the user's gaze, and thus attention, towards the flicker areas of a screen. They state this attention-drawing mechanism to be variable in the sense that once the user's vision has focused on the flicker area, the flickering becomes less salient. This observation is connected to previous research on human vision, where the foveal region (which is used for discerning detail in directly gazed objects) has been found to have a lower critical fusion frequency (CFF) compared to that of its surrounding

visual periphery [50]. The CFF is a measure for the frequency at which flickering images begin to be seen as a continuous, stable image.

Based on these attention-drawing properties, and the uncovered notion that flickering is perceived as annoying by users [14], we are ready to present our hypothesis. We hypothesized that users should be able to discern differences when comparing video sequences of identical scenes rendered using the different methods, provided that there is a significant difference in the amount of temporal aliasing. In addition to the ability to differentiate between results derived from different rendering methods, we also expected users to show preference for methods resulting in less temporal aliasing. The results from the user study indicated these expectations and our hypothesis to have been correct.

5.1 Test Material

5.1.1 Test Scenes

For the user study a total of five different 3D models were used. Two of these were simple geometric shapes generated by Unity, while three were more intricate creations. These three models were downloaded from the 3D model website TurboSquid [49] and all of them are available under a royalty free license. As the material data for the sphere and plane 3D models we used texture maps provided by [4], while with the other models we used the material texture maps custom made and included with each of them. A list of all the models used is provided next:

- **Plane**, 4K materials
- **Sphere**, 4K materials
- **Gnome model**, 12343 polygons, 1K materials
- **Statue model**, 25277 polygons, 4K materials
- **Buddha model**, 15321 polygons, 4K materials

The Gnome model was authored by TurboSquid user "Andromeda_vfx", while the Statue and Buddha models were created by user "Mellowmesher" on the same site.

Here *4K materials* means that the texture maps used for containing material information for a model have the resolution 4096*4096 texels. Similarly, *1K materials* means that the texture map resolution in that case is 1024*1024

texels. From the list, one should note how the Gnome model is the only one to not have 4K materials available. This provides an interesting opportunity to test for the possible effects the use of bilinear texture filtering to upscale the lower resolution texture assets might have on the perceived quality and the users' ability to discriminate between the object-space lighting and screen-space shaded results. As only nearest neighbor filtering is used for the screen space shaded version, the comparison is not like-for-like, but it should serve as a measure through which we can validate that the test worked as intended and that the participants were able to discern differences to the precision expected of them.

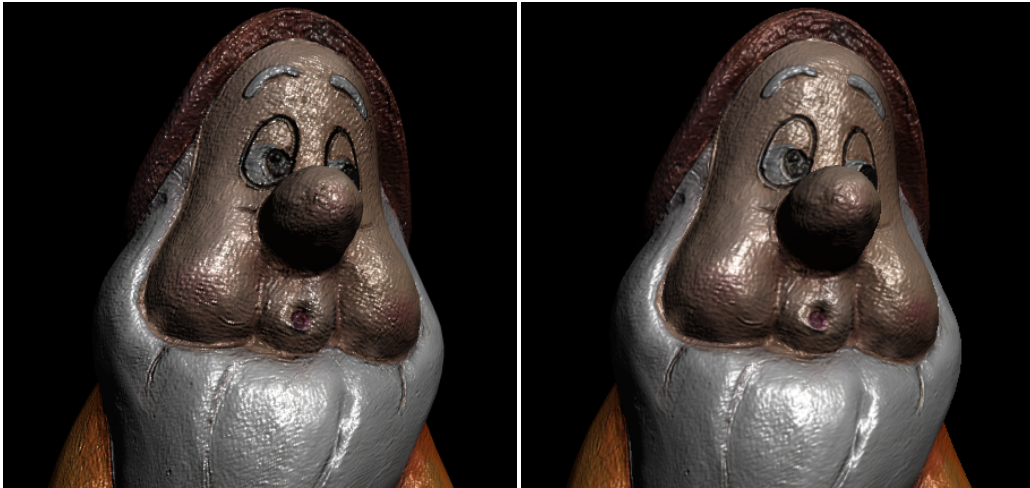


Figure 5.1: A comparison between the 256SPP screen-space shaded image (left) of the Gnome model and the texture-space shaded version (right). Note the difference in appearance caused by the differing texture filtering methods employed.

5.1.2 Video Production

While differences between the two rendering methods we are comparing can be visible even when judged using static rendered images (depending on the exact 3D configuration), differences should be more pronounced during motion due to the presence of temporal aliasing artefacts. This is why the test material for the study was decided to consist of video material. Performing the test in real-time would have been impossible due to the amount of rendering time the production of even a single supersampled screen-space shaded frame takes (resulting in decidedly non-interactive frame update rates), so

the comparison videos were pre-rendered before hand. To provide a like-for-like comparison, all of the other versions of the scenes (including the non-supersampled ones) used as test material were also pre-rendered.

The video creation was done using Unity's built-in image capture functionality, which also provided for a simple way of producing supersampled images. **The native rendering resolution used for the non-supersampled images was 1920*1080**, while the **supersampled screen-space shaded images were rendered at a resolution of 30720*17280**, which is sixteen times the resolution per dimension and 256 times the total image resolution – the maximum allowed by Unity's built-in supersampling. Explained differently, in the supersampled version we are taking 256 samples for each of the screen pixels. The supersampled images were after capturing downsampled to the 1920*1080 resolution using the *IrfanView* [22] image utility program's bulk resize function, with Lanczos filtering utilized for the best available resampling quality.

A total of six 3D scenes were created for the pre-rendering work, with each of them having their unique model-material combination, and light source and virtual camera settings. The discrepancy between us having only five models yet producing six different scenes is explained by us using two different material texture map sets for the plane model during pre-rendering. The intention of this was to provide more variety among the test samples, and to see if the results would differ solely based on material property differences, when the mesh itself is kept the same. Example images displaying the six used scenes are shown in figure 5.2.

Each of the scenes was pre-rendered using three different rendering settings:

- **1SPP, Non-Supersampled Screen-Space Shaded**
- **256SPP, Supersampled Screen-Space Shaded**
- **Texture-Space Shading, Using 4096*4096 Texel Light Map**

In the list, the acronym SPP means *Samples Per Pixel*. The three different rendering settings for each of the six scenes meant a total 18 videos had to be produced. To achieve the test's purpose we arranged these 18 videos into 12 pairs: for each scene there was pair comparison between the 1SPP screen-space shaded and the texture-space shaded version, as well as a pair comparison between the 256SPP screen-space shaded and texture-space shaded version.

The light sources for all of the scenes, except the one using the sphere model, consist of a single directional light. To enable specular reflections to

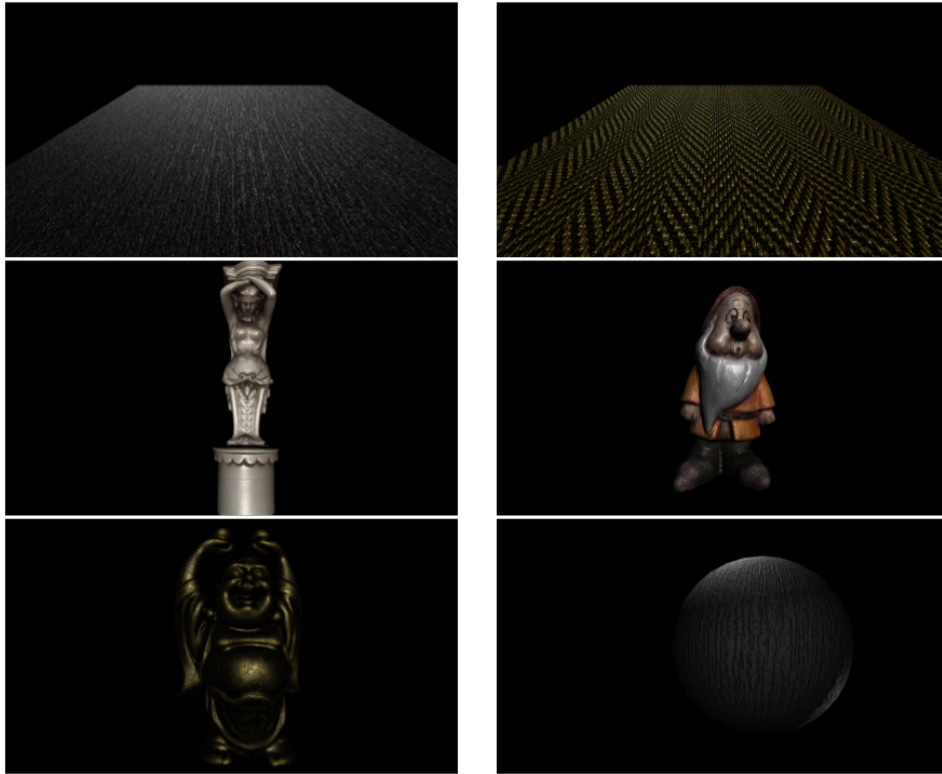


Figure 5.2: On the first row, left to right, we have the Metal Plane and Fabric Plane scenes. The Statue and Gnome scenes are displayed on the second row, while the Buddha and Sphere scenes make up bottom row. The scene names used here are the ones we will adhere to when discussing the survey results.

be more visible with the sphere model, a total of three point light sources with distance-based attenuation were used with no directional light source. One of these point lights' position was animated to further improve the light set up. In each of the scenes the light source colors were pure white light.

For the motion of the virtual camera, model-specific simple animations were created for moving the camera from left-to-right and then right-to-left back to the original position. The duration of the animations was the same as that of the sample videos, 15 seconds, with each pass to one direction taking 7.5 seconds. The smooth movement was to not only provide better view of the models for the test participants, but to also highlight possible temporal aliasing in the form of shimmering specular highlights.

After the images for all of the scenes had been rendered (and in the case of supersampled images, downscaled) the final videos were produced using

FFmpeg [13] to encode the images into 60Hz H.264 videos with 1920*1080 resolution. To enable smooth playback with minimal image quality loss the *high profile, level 4.2* and *crf=18* settings were used for encoding. The choice of 60Hz display rate for the videos is important as while the critical flicker fusion (CFF), mentioned at the beginning of this chapter, ranges between " <10 Hz and ~ 45 Hz" [53] for foveal vision, in the case of peripheral vision it can be between 60 and 70 Hz [50]. This means that temporal flickering occurring at an area where the user focuses on should be perceived as a stable image, while the flickering perceived through peripheral vision should be able to be seen.

5.1.3 Testing methods

In addition to producing the test videos, we still needed to devise a way to present the videos to the users as well as to gather the test data gained from the participants. To this end we developed a web application using the popular Python-based web framework Django [10], and the Formtools extension available for it. Regardless of its web underpinnings, the study was designed to be performed locally on an offline computer in a controlled setting using the Chrome web browser. The test videos and their related questionnaires were presented to the users as a sequence of forms, with the test results being stored into a database for further analysis. During the design phase of the user study, the possibility of displaying the comparison videos side-by-side was explored but discarded due to users possibly not being able to focus properly on the details present in each video. The drawback of the sequential design we finally chose is naturally that it can be limited by the burden it places on a test participant's visual memory.

On the server side, in addition to the proper storing of study results, the main focus was on ensuring that the sequence in which the videos were presented was randomized for every run through of the test. This is because we used a within-groups design for the study where every participant is shown all of the test videos, and we need to counter the possible effect the order of the videos has on their evaluation [46]. In practice this meant that we not only needed to randomize the order between all of the 12 video pairs, but also the order within each of the pairs. The number of videos shown was decided based on a previously used test duration limit [2] and confirmed to be valid in discussion with J. Häkkinen (personal communication, March 28, 2018).

In addition to the questionnaire results, the application stored the following three personal information details of the participants: gender, age and experience level in playing games on computer, console or mobile devices.

The questionnaire that was presented to the participants after each of the 12 shown video pairs was as follows:

<p>Questionnaire</p> <ol style="list-style-type: none">1. Video A and Video B are:<ul style="list-style-type: none">– One video played twice.– Two different videos.2. The better looking video was (choose randomly if they were the same):<ul style="list-style-type: none">– Video A– Video B

The participants were explained that in the first question they were expected to understand that any differences in Video A and Video B of a pair would mean that they were two different videos, while if absolutely no differences could be discerned they would be the same video. In the second question, we also specify that in the case the videos were the same the participants could choose the answer randomly.

This comment was made for two reasons. Firstly, the answer would be non-sensical if the participant did not see any difference between the videos and secondly, we wanted it to appear to the participants as if perceiving the videos as the same was a completely plausible option. If the participants were always expected to make a preference choice, it could lead them to question if differences were in fact always expected to be seen.

To reduce possible noise in the user study results, the participants had to repeat the sequence of watching the 12 video pairs and answering the questionnaire questions three times in total. The final answer for each questionnaire question was taken to be the mode (the most common value) of the three answers. The sequences were always uniquely ordered with regard to the order of the pairs and the videos themselves comprising them, to minimize any effect the ordering might otherwise have had on the answers. After each sequence the participants were allowed to take a short break before continuing on to the next sequence, so as to ensure that they could maintain their focus.

A flow chart of describing a test sequence is shown in figure 5.3.

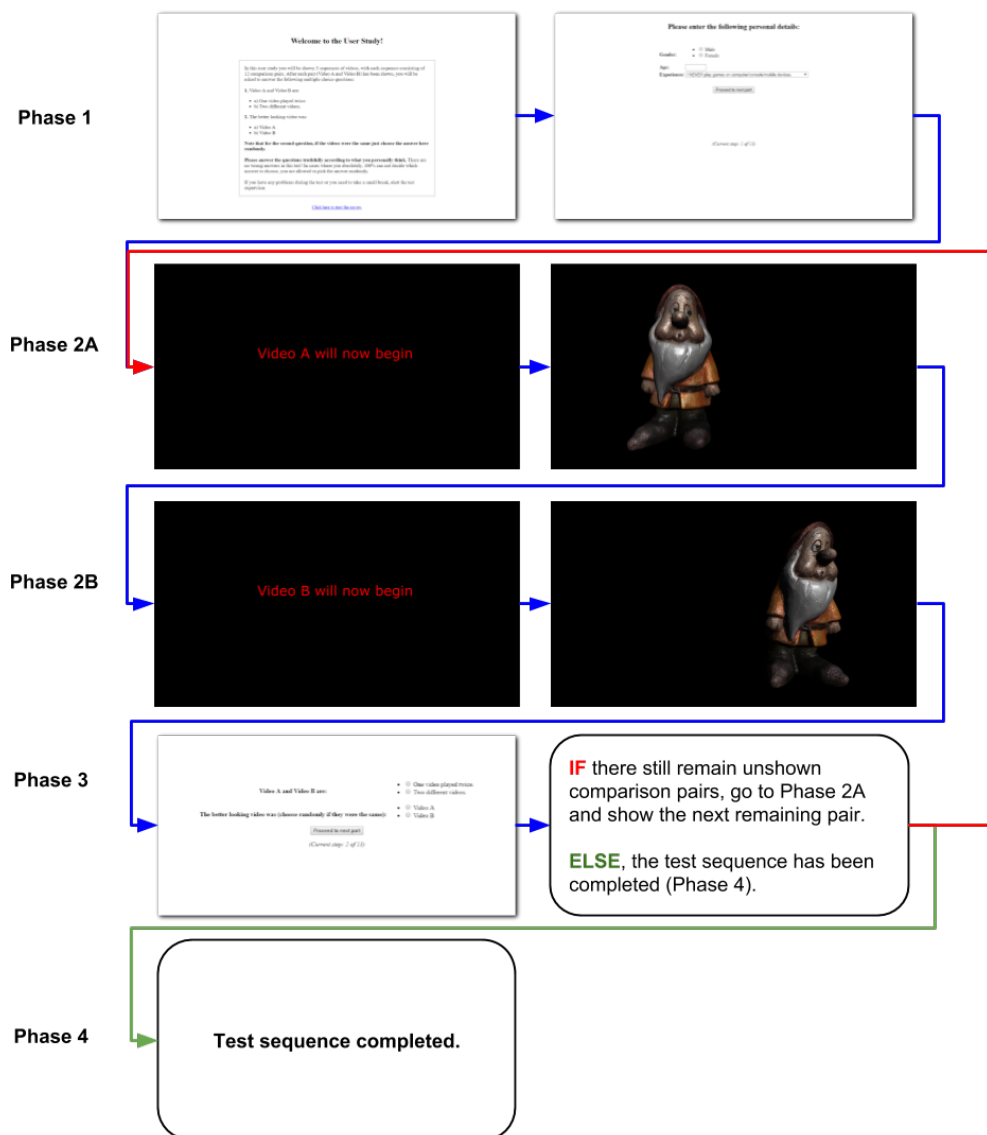


Figure 5.3: Flow chart of a test sequence. In Phase 1 the participant is given an introduction to the survey and asked to enter their personal details. In Phase 2A and 2B the videos for a comparison pair are shown. The videos are preceded by an alert message to attract the participant's focus and to ensure they are aware of the particular video shown (A or B). In Phase 3 the participant is prompted to answer the multiple choice questions. If unshown video pairs remain, we return to Phase 2A with the next (randomized) unshown pair. If no unshown pairs remain, the sequence has been completed.

Once a participant had completed the video comparison part of the user study, we interviewed them immediately afterwards to uncover further details about their choices. The interview was informal in structure and the questions included the following:

- **Generally, was it easy or difficult to see differences?**
- Was there any change in difficulty from the first sequence to the last?
- Were some models more difficult to spot differences in?
- How often would you estimate you answered that the videos were the same?
- **Would you be surprised if you were told that none of the video pairs showed the same video?**

The bolded questions were asked from every participant while the other questions were asked from many, but not from all. Additionally, some participants provided details during the interview that were not specifically asked for but proved to be interesting.

5.1.4 User Study Results

A total of 20 people participated in the user study, with the recruitment taking place either on-site at the Aalto University Department of Computer Science or through electronic advertisements to students and staff at the CS department. All but one of the participants had normal vision, with the lone exception case reporting to having a mild case of red-green color blindness.

The gender distribution of the participants was male dominated, with 75% being male and 25% being female. The average age of the participants was 27.35 years. For the experience levels in playing computer games, 80% of the participants answered to *sometimes* playing computer, console or mobile games, with 15% answering that they played games *often* and only 5% answered they *never* play games.

The results from the questionnaire portion of the user study are displayed in the following table:

Comparison Pair	Perceived as different	Preference	Notes
Metal Plane LM vs 1SPP	100%	LM (100%)	SVBRDF
Metal Plane LM vs 256SPP	10%	LM/256SPP (50%)	SVBRDF
Fabric Plane LM vs 1SPP	100%	LM (85%)	SVBRDF
Fabric Plane LM vs 256SPP	25%	256SPP (80%)	SVBRDF
Sphere LM vs 1SPP	95%	LM (89.5%)	3 point light sources, 1 animated, SVBRDF
Sphere LM vs 256SPP	15%	256SPP (100%)	3 point light sources, 1 animated, SVBRDF
Statue LM vs 1SPP	100%	LM (100%)	Cook-Torrance
Statue LM vs 256SPP	15%	256SPP (100%)	Cook-Torrance
Gnome LM vs 1SPP	95%	LM (94.7%)	1K material texture maps, Cook-Torrance
Gnome LM vs 256SPP	30%	256SPP (83.3%)	1K material texture maps, Cook-Torrance
Buddha LM vs 1SPP	100%	LM (90%)	Cook-Torrance
Buddha LM vs 256SPP	5%	LM (100%)	Cook-Torrance

The overall aggregated results are summarized in the next table:

Comparison Pair	Perceived as different	Preference
<i>Male LM vs 1SPP</i>	100%	LM (94.4%)
<i>Female LM vs 1SPP</i>	93%	LM (93.3%)
<i>Male LM vs 256SPP</i>	17.7%	256SPP (70.8%)
<i>Female LM vs 256SPP</i>	13.3%	LM/256SPP (50%)
Overall LM vs 1SPP	98.3%	LM (93.2%)
Overall LM vs 256SPP	16.7%	256SPP (68.9%)

From these results it is apparent that a vast majority of the **participants were in each case easily able to distinguish between the non-supersampled screen-space shaded (1SPP) version and the texture-space shaded (LM) version, with the percentages for individual scenes ranging between 95% and full 100%**. In all of the cases where differences could be perceived, participants displayed a strong preference towards the image quality provided by the LM version in comparison to the 1SPP – though there was more spread in the answers, with percentages varying in the 85% to 100% range. The choice of the particular BRDF (either the SVBRDF or the Cook-Torrance model) did not appear to have any effect on the results.

Both the ability to differentiate between the LM and 1SPP versions as well as the preference towards the LM version were in line with the theoretical basis. This was expected as all of the test scenes contained models with high specular reflectivity and thus final surface appearances that were highly sensitive to the particular virtual camera position, light source direction and surface point orientation for a given rendered frame. This sensitivity then lead to temporal aliasing, which we hypothesized at the beginning of this chapter to be a major factor for differentiating between the shading techniques.

The comparisons between the LM version and the supersampled screen-space shaded (256SPP) version again provided results

that closely followed the expectations. **The percentage amount of participants able to perceive any differences was between 5% to 30% depending on the test scene.** Theoretically, the perceived differences should arise from the lack of geometry anti-aliasing on the LM version, as well as the shader aliasing still present even in the supersampled 256SPP version and possible aliasing from the hardware-based texture filtering on the LM version.

When comparing the gender specific results, it can be seen that male participants were able to slightly more frequently differentiate between the LM and the two screen-space versions, though it has to be kept in mind that the sample size for female participants was very small.

The two scenes which appeared to have been the easiest for the participants to tell differences between the LM and 256SPP versions were the Fabric Plane and the Gnome scenes. Both of these are interesting cases.

The Fabric Plane contains material texture maps with very high frequency patterns, which should render it more susceptible to sampling rate sensitive aliasing compared to the lower frequency details in the Metal Plane scene. This is a possible explanation for the discrepancy between the Fabric Plane and Metal Plane results, as all of the other parameters including the model meshes, light source direction and intensity as well as the virtual camera positions are identical for both of these test scenes.

The Gnome scene was already from an asset point of view distinctive compared to any of the other scenes, with its 1K material texture maps being bilinearly filtered during the G-buffer's construction to 4K resolution. This in conjunction with the geometry aliasing being relatively more present with higher geometric-complexity models meant that it was not surprising for the participants to perceive differences most easily in this case.

The results presented in the table are given more context when additionally taking into account the answers from the **interview portion of the user study**. When asked to freely assess the difficulty of deciding on the choices in the questionnaire, a majority of the participants expressed (without being prompted) that it was easy to differentiate in cases where there was significant "flickering" present, with a majority also ranking the scenes containing planar models (the Metal Plane and Fabric Plane scenes) as the easiest for perceiving differences. The complex 3D model scenes (Statue, Gnome and Buddha) were generally ranked as the second most difficult group, with the Sphere scene being regarded on average as the most difficult. Several participants specifically highlighted the Sphere scene as being "tricky" to see the differences in, with comments being made on it being "difficult to know where to focus". These participants specified this to be related to the moving

light source in that particular scene. The Gnome scene also received frequent comments from the participants, with some describing it as "confusing" and as exhibiting "different kind of differences", while others remarked it as being "relatively easy to see differences in".

We were also interested to learn about any possible learning effects the participants might have experienced during the test. To this end, a majority (75%) of the participants were specifically asked to describe whether they had felt any change in difficulty between the first and the last sequence of the pair comparisons. 67% answered that they felt the differentiating was most difficult during the first sequence, while 20% said that the first sequence was the easiest and 13% felt there was no change in difficulty

The participants who regarded the first sequence as the most difficult remarked that it initially took some time for them to "get accustomed to noticing the differences" or "get a hang of things" and that towards the end they "stopped overanalyzing" and "knew what to focus on". Those who felt the test to become more difficult by the third sequence commented they believed to have been "more easily able to see differences at first" and later on started to worry whether they had developed a "bias in evaluation" and that "more differences might be present" than they had initially thought and answered during the first sequence.

The general trend of these comments suggests a learning effect, which might partially explain why the participants were poorly able to perceive the geometry aliasing in the LM versions of the test scenes. As the shading aliasing with the highly reflective test materials was so significant, it is perhaps likely that its presence on one hand in the 1SPP version and its absence on the other hand in the LM version resulted in such a dramatic change in visual appearance to the participants that the much more minor geometry edge aliasing was mostly overlooked. This is only logical as the number of screen pixels covered by a model surface (and thus subjected to shading aliasing) far outnumbers the number of screen pixels covered by multiple geometry primitives for any of the test scenes' frames. Earlier research by Jukarainen [24] also supports the hypothesis of geometry anti-aliased images being difficult to differentiate from images which have not undergone such anti-aliasing.

Chapter 6

Conclusions

6.1 Summary of Results

In this thesis we implemented an object-space lighting technique where the shading is performed in texture-space using a unique parametrization for the 3D model surface. This implementation was verified to be correct in evaluations against ground truth renderings produced using a supersampled screen-space equivalent of the shading algorithm. Further verification was gained in the user study we conducted, where the texture-space shading was again pitted against a traditional screen-space shading technique utilizing the same reflectance algorithms as the texture-space variant.

The results from the user study suggest that texture-space shading onto a 4096*4096 texel resolution light map and then anisotropically filtering the results provides visual quality that is difficult to distinguish from that afforded by a screen-space shaded, 256 samples per pixel technique, where the base screen resolution is 1920*1080 pixels. This is especially significant, as the texture-space shaded variant in the user study did not employ the use of any geometry anti-aliasing such as MSAA due to technical difficulties. This suggests that the aliasing artefacts born from surface shading operations are more visible to users than geometry-based aliasing. We consider it to be likely that the results from the shading methods would have been even more difficult to appreciate had MSAA been used for the texture-space shading version.

We had hypothesized for the temporal aliasing observable as flickering to be the major distinguishing factor through which users would be able to perceive differences in the results produced by the different shading methods and this was indicated to be correct both in the multiple-choice questionnaire results as well as the answers given during the informal interview portion of

the user study.

In addition to the ability to distinguish images created using the different techniques, we also tested for possible preferences users might exhibit. The results indicated that users almost unilaterally prefer the texture-space shaded results to screen-space shaded ones using 1 sample per pixel, while a slight preference for the 256 samples per pixel version was seen compared to the texture-space shaded version. The sample size for the latter preference test was small (we only counted cases where a test participant could distinguish that the renderings actually were different), which means that this result should be treated carefully.

The overall results give reason to believe that texture-space shading can give results which equal those of highly supersampled screen-space equivalents, without having their performance cost linked to the used back buffer resolution or even the back buffer update rate. This can be of notable benefit in cases where a high screen resolution or refresh rate is mandated, as the shading precision and frequency can be set to vary on arbitrary variables that might change from one scene and situation to another. One must also consider that the fact that we perform filtering in the texture-space shading version only on the final radiance values means that we can also reduce the error traditionally introduced by the pre-filtering of shading input values – particularly the mip-mapping of normal maps [42], in lighting algorithms. Texture-space shading can thus offer an option with a great degree of flexibility, while theoretically being more accurate than traditional screen-space techniques.

6.2 Avenues for Future Developments

The user study conducted for this thesis could not completely separate the effects of geometry and surface shading aliasing from each other. Although the results suggest shader aliasing to be perceptually the dominant factor, a more extensive study is needed in order to form fully satisfactory conclusions.

In order to draw the full benefits from the possible performance improvements of texture-space shading, research needs to be done to uncover the relationship between achieved image quality and factors such as the light map resolution and update rate, material reflectivity and the rate of change for the viewer position or the light source direction in a scene. This information would be of vital importance to control the size of the memory footprint requirement imposed by the light map, as well as to avoid the perceptually wasteful shading operations. Similarly, for implementations intended for use in commercial products, it is advisable to look for ways in which one can

limit the shaded texels to only those visible to the viewer during a given shading update. This and various other practical considerations for object-space lighting were discussed by Baker in his 2016 GDC presentation [7]. Baker was involved in the development of the computer game *Ashes of the Singularity* [43], which is the most prominent recent commercial release utilizing object-space lighting.

Finally, as object-space lighting presents a significant departure from the conventional screen-space lighting algorithms, thorough understanding of its properties may lead to novel techniques beyond those of simple decouplings from back buffer properties.

6.3 Final Thoughts

The findings of this thesis indicate that the use of object-space lighting can result in extremely high quality shading results that have the means to be theoretically more accurate to reality compared to conventional screen-space lighting algorithms. This means that it is able to meet even high qualitative expectations, leaving performance to be main area of focus going forward. Given further research on its optimization, object-space lighting holds promise to be able to compliment traditional methods in the continuing progress towards ever-higher visual realism and enhanced user experiences.

Bibliography

- [1] ABRASH, M. What VR could, should, and almost certainly will be within two years. *Steam Dev Days, Seattle* (2014).
- [2] AFLAKI, P., HANNUKSELA, M. M., HAKALA, J., HÄKKINEN, J., AND GABBOUJ, M. Joint adaptation of spatial resolution and sample value quantization for asymmetric stereoscopic video compression: A subjective study. In *2011 7th International Symposium on Image and Signal Processing and Analysis (ISPA)* (Sept 2011), pp. 396–401.
- [3] AITTALA, M., WEYRICH, T., AND LEHTINEN, J. Two-shot SVBRDF Capture for Stationary Materials. *ACM Trans. Graph.* *34*, 4 (July 2015), 110:1–110:13.
- [4] AITTALA, M., WEYRICH, T., AND LEHTINEN, J. Two-Shot SVBRDF Capture for Stationary Materials, (Supplemental material archive (ZIP)), 2015. <https://mediatech.aalto.fi/publications/graphics/TwoShotSVBRDF/>. Accessed 29.4.2018.
- [5] AKENINE-MÖLLER, T., HAINES, E., AND HOFFMAN, N. *Real-Time Rendering, Third Edition*. A K Peters/CRC Press, 2008.
- [6] AUTODESK INC. List Of All Products — New Releases — Autodesk, 2018. <https://www.autodesk.com/products>. Accessed 29.4.2018.
- [7] BAKER, D. Object Space Lighting. Game Developers Conference 2016, 2016.
- [8] CHANG, C.-F., CHEN, K.-W., AND CHUANG, C.-C. Performance comparison of rasterization-based graphics pipeline and ray tracing on GPU shaders. In *2015 IEEE International Conference on Digital Signal Processing (DSP)* (2015), pp. 120–123.

- [9] COOK, R. L., AND TORRANCE, K. E. A reflectance model for computer graphics. In *SIGGRAPH '81 Proceedings of the 8th annual conference on Computer graphics and interactive techniques* (1981), pp. 307–316.
- [10] DJANGO SOFTWARE FOUNDATION. The Web framework for perfectionists with deadlines — Django, 2018. <https://www.djangoproject.com/>. Accessed 11.5.2018.
- [11] DONNER, C., AND JENSEN, H. W. A Spectral BSSRDF for Shading Human Skin. In *Proceedings of the 17th Eurographics Conference on Rendering Techniques* (Aire-la-Ville, Switzerland, Switzerland, 2006), EGSR '06, Eurographics Association, pp. 409–417.
- [12] EPIC GAMES. What is Unreal Engine 4, 2018. <https://www.unrealengine.com/en-US/what-is-unreal-engine-4>. Accessed 9.5.2018.
- [13] FFMPEG. A complete, cross-platform solution to record, convert and stream audio and video., 2018. <https://www.ffmpeg.org/>. Accessed 11.5.2018.
- [14] GLUCK, J., BUNT, A., AND MCGRENERE, J. Matching Attentional Draw with Utility in Interruption. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2007), CHI '07, ACM, pp. 41–50.
- [15] HARDING-ROLLS, P., KENT, J., CARRERA, P., HANCOCK, D., CUI, C., BAILEY, S., CRYAN, D. Immersive Computing - Consumer Augmented & Virtual Reality Report - 2018, 2018. <https://technology.ihs.com/591822/immersive-computing-consumer-augmented-virtual-reality-report-2018>. Accessed 7.5.2018.
- [16] HASSELGREN, J., AKENINE-MÖLLER, T., AND OHLSSON, L. *Conservative Rasterization*. GPU Gems 2. Addison-Wesley, 2005, pp. 677–690.
- [17] HECKBERT, P. S. Fundamentals of Texture Mapping and Image Warping. Tech. rep., Berkeley, CA, USA, 1989.
- [18] HENNESSY, J. L., AND PATTERSON, D. A. *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2011.

- [19] HILLESLAND, K. E., AND YANG, J. C. Texel Shading. In *Proceedings of the 37th Annual Conference of the European Association for Computer Graphics: Short Papers* (Goslar Germany, Germany, 2016), EG '16, Eurographics Association, pp. 73–76.
- [20] HILLESLAND, K. Texel Shading, 2016. <https://gpuopen.com/texel-shading/>. Accessed 2.5.2018.
- [21] HOFFMAN, N. Background: Physics and Math of Shading, July 2013. http://blog.selfshadow.com/publications/s2013-shading-course/hoffman/s2013_pbs_physics_math_notes.pdf. Accessed 7.5.2018.
- [22] IRFAN SKILJAN. IrfanView - Official Homepage - One of the Most Popular Viewers Worldwide, 2018. <https://www.irfanview.com/>. Accessed 11.5.2018.
- [23] JIMENEZ, J., ECHEVARRIA, J. I., SOUSA, T., AND GUTIERREZ, D. SMAA: Enhanced Morphological Antialiasing. *Computer Graphics Forum (Proc. EUROGRAPHICS 2012)* 31, 2 (2012).
- [24] JUKARAINEN, P. Comparison of Real-Time Anti-Aliasing Methods on a Head-Mounted Display. Master's thesis, Department of Computer Science, Aalto University School of Science and Technology, Espoo, Finland, 2016. <https://aaltodoc.aalto.fi/handle/123456789/23342>.
- [25] KARIS, B. Graphic Rants: Specular BRDF Reference, 2013. <http://graphicrants.blogspot.fi/2013/08/specular-brdf-reference.html>. Accessed 10.4.2018.
- [26] KATO, S., LAKSHMANAN, K., RAJKUMAR, R., AND ISHIKAWA, Y. TimeGraph: GPU Scheduling for Real-time Multi-tasking Environments. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2011), USENIXATC'11, USENIX Association, pp. 2–2.
- [27] KNUUTTILA, J. A Direct3D 11 program to render captured materials using Oculus Rift, using materials from Two-Shot SVBRDF Capture for Stationary Materials by Aittala et al (2015)., 2016. <https://github.com/jknuuttila/svbrdf-oculus>. Accessed 29.4.2018.
- [28] LAURITZEN, A. Deferred Rendering for Current and Future Rendering Pipelines.

- [29] LING, Y., BRINKMAN, W.-P., NEFS, H. T., QU, C., AND HEYNDERICKX, I. Effects of Stereoscopic Viewing on Presence, Anxiety, and Cybersickness in a Virtual Reality Environment for Public Speaking. *Presence: Teleoperators and Virtual Environments* 21, 3 (2012), 254–267.
- [30] LOTTES, T. FXAA (Fast Approximate Anti-Aliasing), 2009. http://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_WhitePaper.pdf. Accessed 29.4.2018.
- [31] LUNA, F. *Introduction to 3D Game Programming with DirectX 11*. Mercury Learning & Information, 2012.
- [32] MACEDONIA, M. The GPU Enters Computing’s Mainstream. *Computer* 36, 10 (Oct. 2003), 106–108.
- [33] MICROSOFT. 0x/2x/4x MSAA Variants, 2016. <https://docs.microsoft.com/en-us/visualstudio/debugger/graphics/0x-2x-4x-msaa-variants>. Accessed 2.5.2018.
- [34] MICROSOFT. Texture addressing modes - UWP app developer, 2017. <https://docs.microsoft.com/en-us/windows/uwp/graphics-concepts/texture-addressing-modes>. Accessed 10.4.2018.
- [35] MICROSOFT. Announcing Microsoft DirectX Raytracing!, 2018. <https://blogs.msdn.microsoft.com/directx/2018/03/19/announcing-microsoft-directx-raytracing/>. Accessed 3.5.2018.
- [36] MICROSOFT. DirectX Graphics and Gaming, 2018. [https://msdn.microsoft.com/en-us/library/windows/desktop/ee663274\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ee663274(v=vs.85).aspx). Accessed 7.5.2018.
- [37] MICROSOFT. HLSL, 2018. [https://msdn.microsoft.com/en-us/library/windows/desktop/bb509561\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/bb509561(v=vs.85).aspx). Accessed 5.5.2018.
- [38] MICROSOFT. Pipelines and Shaders with Direct3D 12, 2018. [https://msdn.microsoft.com/en-us/library/windows/desktop/dn899200\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/dn899200(v=vs.85).aspx). Accessed 29.4.2018.
- [39] NEUBELT, D., AND PETTINEO, M. Crafting a Next-Gen Material Pipeline for The Order: 1886, July 2013. http://blog.selfshadow.com/publications/s2013-shading-course/rad/s2013_pbs_rad_slides.pdf. Accessed 5.5.2018.

- [40] NVIDIA. Falcor — NVIDIA Developer, 2018. <https://developer.nvidia.com/falcor>. Accessed 9.5.2018.
- [41] OCULUS VR. VR Best Practices - Rendering, 2018. <https://developer.oculus.com/design/latest/concepts/bp-rendering/>. Accessed 29.4.2018.
- [42] OLANO, M., AND BAKER, D. LEAN Mapping. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2010), I3D '10, ACM, pp. 181–188.
- [43] OXIDE GAMES. Ashes of the Singularity: Planetary Warfare on a massive scale, 2016. <https://www.ashesofthesingularity.com/>. Accessed 30.5.2018.
- [44] PINEDA, J. A Parallel Algorithm for Polygon Rasterization. *SIGGRAPH Comput. Graph.* 22, 4 (June 1988), 17–20.
- [45] SCHLICK, C. An Inexpensive BRDF Model for Physically-based Rendering. *Computer Graphics Forum* 13 (1994), 233–246.
- [46] SLATER, M., KHANNA, P., MORTENSEN, J., AND YU, I. Visual Realism Enhances Realistic Response in an Immersive Virtual Environment. *IEEE Comput. Graph. Appl.* 29, 3 (May 2009), 76–84.
- [47] THE KHRONOS GROUP. OpenGL Overview, 2017. <https://www.khronos.org/opengl/>. Accessed 7.5.2018.
- [48] TOKSVIG, M. Mipmapping Normal Maps. *Journal of Graphics Tools* 10, 3 (2005), 65–71.
- [49] TURBOSQUID. TurboSquid: 3D Models for Professionals, 2018. <https://www.turbosquid.com/>. Accessed 30.5.2018.
- [50] TYLER, C. W., AND HAMER, R. D. Eccentricity and the Ferry–Porter law. *J. Opt. Soc. Am. A* 10, 9 (Sep 1993), 2084–2087.
- [51] UNITY TECHNOLOGIES. Unity - Products, 2018. <https://unity3d.com/unity>. Accessed 9.5.2018.
- [52] VAN WAVEREN, J. M. P. The Asynchronous Time Warp for Virtual Reality on Consumer Hardware. In *Proceedings of the 22Nd ACM Conference on Virtual Reality Software and Technology* (New York, NY, USA, 2016), VRST '16, ACM, pp. 37–46.

- [53] WALDIN, N., WALDNER, M., AND VIOLA, I. Flicker Observer Effect: Guiding Attention Through High Frequency Flicker in Images. *Computer Graphics Forum* 36, 2, 467–476.

Appendix A

Unity Script Pseudocode

```
public ComputeShader RasterizerCS;  
public ComputeShader LightCS;  
public ComputeShader ConservativeCS;  
public ComputeShader DilationCS;  
public ComputeShader DilationHelperCS;  
public RenderTexture lightmap;  
ComputeBuffer Gbuffer;  
  
// Adjust Texture-Space Shading resolution:  
public int lightMapWidth = 1024;  
public int lightMapHeight = 1024;  
  
struct LMVertex  
{  
    public Vector3 position;  
    public Vector3 normal;  
    public Vector4 tangent;  
    public Vector2 texC;  
    public Vector2 originalUVs;  
}  
  
struct Texel  
{  
    public Vector3 weights;  
    public Vector3 localNormal;  
    public Vector4 localTangent;
```

```

    public Vector2 texCoords;
    public Vector3 diffuseAlbedo;
    public Vector3 normalSample;
    public Vector3 specVal;
    public Vector3 specShape;
    int triangleIndex;
    bool isDilated;
    bool isFinal;
}

...

void Start()
{
    /* Initialize Compute Buffers
     * (Vertex, Index, G-Buffer and LightMap)
     * and Constants */

    // Non-Conservative Rasterization:
    int rasterizerHandle = RasterizerCS.FindKernel("CSMain");

    /* Set Textures, Buffers and Constants
     *for the Rasterizer CS, such as: */
    RasterizerCS.SetBuffer(rasterizerHandle,
        "gVertices", vertexBuffer);
    RasterizerCS.SetBuffer(rasterizerHandle,
        "gIndices", indexBuffer);
    RasterizerCS.SetBuffer(rasterizerHandle,
        "gGBuffer", Gbuffer);

    RasterizerCS.Dispatch(rasterizerHandle, indices.Count / 3, 1, 1);

    // Conservative Rasterization:
    int conservativeHandle = ConservativeCS.FindKernel("CSMain");

    /* Set Textures, Buffers and Constants
     *for the Conservative CS, such as: */
    ConservativeCS.SetBuffer(
        conservativeHandle, "gVertices", vertexBuffer);
    ConservativeCS.SetBuffer(
        conservativeHandle, "gIndices", indexBuffer);

```

```

ConservativeCS.SetBuffer(
    conservativeHandle, "gGBuffer", Gbuffer);

ConservativeCS.Dispatch(conservativeHandle,
    indices.Count / 3, 1, 1);

// Dilation:
for (int i = 0; i < 40; ++i)
{
    int dilationHandle = DilationCS.FindKernel("CSMain");

    /* Set Textures, Buffers and Constants
       *for the Dilation CS, such as: */
    DilationCS.SetBuffer(dilationHandle, "gGBuffer", Gbuffer);
    DilationCS.SetFloat(Shader.PropertyToID("kWidth"),
        computeConstants[3]);
    DilationCS.SetFloat(Shader.PropertyToID("kHeight"),
        computeConstants[4]);

    // Dilate texel values to empty neighboring texels:
    DilationCS.Dispatch(dilationHandle,
        lightMapWidth, lightMapHeight, 1);

    /* Set Textures, Buffers and Constants
       *for the Dilation Helper CS, such as: */
    int helperHandle = DilationHelperCS.FindKernel("CSMain");
    DilationHelperCS.SetBuffer(helperHandle, "gGBuffer", Gbuffer);
    DilationHelperCS.SetFloat(
        Shader.PropertyToID("kWidth"), computeConstants[3]);
    DilationHelperCS.SetFloat(
        Shader.PropertyToID("kHeight"), computeConstants[4]);

    // Finalize the texels we just dilated:
    DilationHelperCS.Dispatch(
        dilationHandle, lightMapWidth, lightMapHeight, 1);
}

// Texture-Space Shading:
int lightingHandle = LightCS.FindKernel("CSMain");

LightCS.SetBuffer(lightingHandle, "gGBuffer", Gbuffer);

```

```

    LightCS.SetBuffer(lightingHandle, "gVertices", vertexBuffer);
    LightCS.SetBuffer(lightingHandle, "gIndices", indexBuffer);
    LightCS.SetTexture(lightingHandle, "gBackBuffer", lightmap);
    LightCS.SetFloat(
        Shader.PropertyToID("kF0"), computeConstants[0]);
    LightCS.SetVector(
        Shader.PropertyToID("gCamLocalPos"), lConstants[2]);

    ...

    LightCS.Dispatch(lightingHandle,
        lightMapWidth / 32, lightMapHeight / 32, 1);

    RenderTexture.active = lightmap;
    lightmap.GenerateMips();
    gameObject.GetComponent<Renderer>().material.mainTexture
        = lightmap;
}

// Update is called once per frame
void Update()
{
    int lightingHandle = LightCS.FindKernel("CSMain");

    LightCS.SetBuffer(lightingHandle, "gGBuffer", Gbuffer);
    LightCS.SetBuffer(lightingHandle, "gVertices", vertexBuffer);
    LightCS.SetBuffer(lightingHandle, "gIndices", indexBuffer);
    LightCS.SetTexture(lightingHandle, "gBackBuffer", lightmap);
    LightCS.SetFloat(Shader.PropertyToID("kF0"),
        computeConstants[0]);
    LightCS.SetFloat(Shader.PropertyToID("kDiffuse"),
        computeConstants[1]);
    LightCS.SetFloat(Shader.PropertyToID("kAlphaVal"),
        computeConstants[2]);
    LightCS.SetFloat(Shader.PropertyToID("kWidth"),
        computeConstants[3]);
    LightCS.SetFloat(Shader.PropertyToID("kHeight"),
        computeConstants[4]);

    /* Update Light Source and Camera Data */

```

```
...

// Update Light Map:
LightCS.Dispatch(lightingHandle,
    lightMapWidth / 32, lightMapHeight / 32, 1);

lightmap.GenerateMips();
lightmap.anisoLevel = 16;
gameObject.GetComponent<Renderer>().material.mainTexture
    = lightmap;
}
```

Appendix B

Rasterization Pseudocode

```
struct Texel
{
    float3 weights;
    float3 localNormal;
    float4 localTangent;
    float2 texCoords;
    float3 diffuseAlbedo;
    float3 normalSample;
    float3 specVal;
    float3 specShape;
    int triangleIndex;
    bool isDilated;
    bool isFinal;
};
```

```
struct Vertex
{
    float3 positionL;
    float3 normalL;
    float4 tangent;
    float2 texC;
    float2 originalUVs;
};
```

```
struct Index
{
```

```

        int num;
    };

    SamplerState MyPointClampSampler;
    Texture2D gDiffuseMap;
    Texture2D gNormalMap;
    Texture2D gSpecMap;
    Texture2D gSpecShapeMap;
    RWStructuredBuffer<Texel> gGBuffer;

    StructuredBuffer<Vertex> gVertices;
    StructuredBuffer<Index> gIndices;
    float kWidth;
    float kHeight;

    float edgeTest(float2 a, float2 b, float2 p)
    {
        return (p.x - a.x) * (b.y - a.y) - (p.y - a.y) * (b.x - a.x);
    }

    [numthreads(1, 1, 1)]
    void CSMain(uint3 DTid : SV_DispatchThreadID,
              uint3 GroupID : SV_GroupID)
    {
        int lightmapWidth = (int)kWidth;
        int lightmapHeight = (int)kHeight;

        Texel texelData;
        /* texelData initialization */

        Vertex v0 = gVertices[gIndices[GroupID.x * 3].num];
        Vertex v1 = gVertices[gIndices[GroupID.x * 3 + 1].num];
        Vertex v2 = gVertices[gIndices[GroupID.x * 3 + 2].num];

        // calculate bounding box for the current triangle:
        float minX = min(v0.texC.x, min(v1.texC.x, v2.texC.x)) * kWidth;
        float minY = min(v0.texC.y, min(v1.texC.y, v2.texC.y)) * kHeight;
        float maxX = max(v0.texC.x, max(v1.texC.x, v2.texC.x)) * kWidth;
        float maxY = max(v0.texC.y, max(v1.texC.y, v2.texC.y)) * kHeight;

        // traversal is done in texel coordinates, so perform conversion:

```

```

int XMIN = max(int(0), int(minX));
int XMAX = min(int(kWidth) - 1, int(maxX));
int YMIN = max(int(0), int(minY));
int YMAX = min(int(kHeight) - 1, int(maxY));

// calculate the triangle area:
float triangleArea = edgeTest(v0.texC, v1.texC, v2.texC);

float3 w; // barycentric coordinates, "weights"
float4 packedNormal;
float3 unpackedNormal;

for (int y = YMIN; y <= YMAX; ++y)
{
    for (int x = XMIN; x <= XMAX; ++x)
    {
        // we only want to rasterize to empty texels:
        if (gGBuffer[y*kWidth + x].hasOwner)
            continue;

        // sample texel center:
        float2 lightMapCoord = float2((float(x) + 0.5f) / kWidth,
                                     (float(y) + 0.5f) / kHeight);

        w.x = edgeTest(v1.texC, v2.texC, lightMapCoord);
        w.y = edgeTest(v2.texC, v0.texC, lightMapCoord);
        w.z = edgeTest(v0.texC, v1.texC, lightMapCoord);

        // non-negative results mean point is inside the triangle:
        if (w.x >= 0 && w.y >= 0 && w.z >= 0)
        {
            // calculate interpolation weights:
            w.x /= triangleArea;
            w.y /= triangleArea;
            w.z /= triangleArea;

            texelData.weights = float3(w.x, w.y, w.z);
            texelData.localNormal = normalize((w.x) * v0.normalL
                                              + (w.y) * v1.normalL + (w.z) * v2.normalL);
            texelData.localTangent.xyz = normalize((
                (w.x) * v0.tangent + (w.y) * v1.tangent

```

```

        + (w.z) * v2.tangent).xyz);
texelData.localTangent.xyz =
    normalize(texelData.localTangent.xyz -
        dot(texelData.localTangent.xyz,
            texelData.localNormal)*texelData.localNormal);
texelData.localTangent.w = v0.tangent.w;
texelData.texCoords = (w.x) * v0.texC
    + (w.y) * v1.texC + (w.z) * v2.texC;

float2 origUVs = (w.x) * v0.originalUVs
    + (w.y) * v1.originalUVs + (w.z) * v2.originalUVs;
texelData.diffuseAlbedo = gDiffuseMap.SampleLevel(
    MyPointClampSampler, origUVs, 0).xyz;

// unpack the sampled tangent space normal:
packedNormal = gNormalMap.SampleLevel(
    MyPointClampSampler, origUVs, 0);
unpackedNormal.xy = packedNormal.wy * 2 - 1;
unpackedNormal.z = sqrt(1 - saturate(
    dot(unpackedNormal.xy, unpackedNormal.xy)));

texelData.normalSample = normalize(unpackedNormal);
texelData.specVal = gSpecMap.SampleLevel(
    MyPointClampSampler, origUVs, 0).xyz;
texelData.specShape = gSpecShapeMap.SampleLevel(
    MyPointClampSampler, origUVs, 0).xyz;
texelData.triangleIndex = GroupID.x;
texelData.isDilated = false;
texelData.isFinal = true;

gGBuffer[y*kWidth + x] = texelData;
    }
}
}
}

```

Appendix C

Conservative Rasterization Pseudocode

```
// The buffers, variables and the edgeTest function used here
// are the exact same as those in the Rasterization Pseudocode

[numthreads(1, 1, 1)]
void CSMain(uint3 DTid : SV_DispatchThreadID,
            uint3 GroupID : SV_GroupID)
{
    int lightmapWidth = (int)kWidth;
    int lightmapHeight = (int)kHeight;

    Vertex v0 = gVertices[gIndices[GroupID.x * 3].num];
    Vertex v1 = gVertices[gIndices[GroupID.x * 3 + 1].num];
    Vertex v2 = gVertices[gIndices[GroupID.x * 3 + 2].num];

    // dimensions of half-texel cell
    float2 hPixel = float2(0.5f / kWidth, 0.5f / kHeight);

    // calculate bounding box using original vertices:
    float minX = max((min(v0.texC.x,
                          min(v1.texC.x, v2.texC.x)) - hPixel.x) * kWidth, 0.0f);
    float minY = max((min(v0.texC.y,
                          min(v1.texC.y, v2.texC.y)) - hPixel.y) * kHeight, 0.0f);
    float maxX = min((max(v0.texC.x,
                          max(v1.texC.x, v2.texC.x)) + hPixel.x) * kWidth, kWidth);
```

```

float maxY = min((max(v0.texC.y,
    max(v1.texC.y, v2.texC.y)) + hPixel.y) * kHeight, kHeight);

// take vertices to homogenous 3D:
float3 vh0 = float3(v0.texC, 1.0f);
float3 vh1 = float3(v1.texC, 1.0f);
float3 vh2 = float3(v2.texC, 1.0f);

// compute equations for the planes through the triangle edges:
float3 plane[3];
plane[0] = cross(vh1 - vh0, vh0);
plane[1] = cross(vh2 - vh1, vh1);
plane[2] = cross(vh0 - vh2, vh2);

// move the planes by the appropriate semidiagonal:
plane[0].z += dot(hPixel.xy, abs(plane[0].xy));
plane[1].z += dot(hPixel.xy, abs(plane[1].xy));
plane[2].z += dot(hPixel.xy, abs(plane[2].xy));

// compute the intersection points of the planes:
float3 intersect0 = cross(plane[0], plane[1]);
float3 intersect1 = cross(plane[1], plane[2]);
float3 intersect2 = cross(plane[2], plane[0]);

float2 pos0 = intersect0.xy / intersect0.z;
float2 pos1 = intersect1.xy / intersect1.z;
float2 pos2 = intersect2.xy / intersect2.z;

Texel texelData;
    /* initialize texelData */

// calculate the triangle area:
float triangleArea = edgeTest(pos0.xy, pos1.xy, pos2.xy);
float origTriArea = edgeTest(vh0.xy, vh1.xy, vh2.xy);

float3 w; // barycentric coordinates, "weights"
float4 packedNormal;
float3 unpackedNormal;
float3 origTriWeights = float3(0.0f, 0.0f, 0.0f);

int XMIN = max(int(0), int(minX));

```

```

int XMAX = min(int(kWidth) - 1, int(maxX));
int YMIN = max(int(0), int(minY));
int YMAX = min(int(kHeight) - 1, int(maxY));

for (int y = YMIN; y <= YMAX; ++y)
{
    for (int x = XMIN; x <= XMAX; ++x)
    {
        if (gGBuffer[y*kWidth + x].hasOwner)
            continue;

        float2 lightMapCoord =
            float2((float(x) + 0.5f) / kWidth,
                  (float(y) + 0.5f) / kHeight);

        w.x = edgeTest(pos1.xy, pos2.xy, lightMapCoord);
        w.y = edgeTest(pos2.xy, pos0.xy, lightMapCoord);
        w.z = edgeTest(pos0.xy, pos1.xy, lightMapCoord);

        if (w.x >= 0 && w.y >= 0 && w.z >= 0)
        {
            //interpolate values:
            w.x /= triangleArea;
            w.y /= triangleArea;
            w.z /= triangleArea;

            origTriWeights.x =
                edgeTest(vh1.xy, vh2.xy, lightMapCoord)
                / origTriArea;
            origTriWeights.y =
                edgeTest(vh2.xy, vh0.xy, lightMapCoord)
                / origTriArea;
            origTriWeights.z =
                edgeTest(vh0.xy, vh1.xy, lightMapCoord)
                / origTriArea;

            texelData.weights = origTriWeights;
            texelData.localNormal
                = normalize((w.x) * v0.normalL
                           + (w.y) * v1.normalL + (w.z) * v2.normalL);
            texelData.localTangent.xyz

```

APPENDIX C. CONSERVATIVE RASTERIZATION PSEUDOCODE 117

```

        = normalize(((w.x) * v0.tangent
        + (w.y) * v1.tangent + (w.z) * v2.tangent).xyz);
texelData.localTangent.xyz =
    normalize(texelData.localTangent.xyz
    - dot(texelData.localTangent.xyz,
    texelData.localNormal) * texelData.localNormal);
texelData.localTangent.w = v0.tangent.w;
texelData.texCoords = (w.x) * v0.texC
    + (w.y) * v1.texC + (w.z) * v2.texC;

float2 origUVs = (w.x) * pos0.xy
    + (w.y) * pos1.xy + (w.z) * pos2.xy;
texelData.diffuseAlbedo = gDiffuseMap.SampleLevel(
    MyPointClampSampler, origUVs, 0).xyz;

// unpack the sampled tangent space normal:
packedNormal = gNormalMap.SampleLevel(
    MyPointClampSampler, origUVs, 0);
unpackedNormal.xy = packedNormal.wy * 2 - 1;
unpackedNormal.z = sqrt(1 - saturate(
    dot(unpackedNormal.xy, unpackedNormal.xy)));

texelData.normalSample = normalize(unpackedNormal);
texelData.specVal = gSpecMap.SampleLevel(
    MyPointClampSampler, origUVs, 0).xyz;
texelData.specShape = gSpecShapeMap.SampleLevel(
    MyPointClampSampler, origUVs, 0).xyz;
texelData.triangleIndex = GroupID.x;
texelData.isDilated = true;
texelData.isFinal = true;

gGBuffer[y*kWidth + x] = texelData;

    }

}

}

```