Screen Space Reflection Techniques

A Thesis Submitted to
The Faculty of Graduate Studies and Research
Master of Science
in Computer Science
University of Regina

by

Anthony Paul Beug
Regina, Saskatchewan
January 2020

UNIVERSITY OF REGINA

FACULTY OF GRADUATE STUDIES AND RESEARCH

SUPERVISORY AND EXAMINING COMMITTEE

Anthony Paul Beug, candidate for the degree of Master of Science in Computer Science**,** has presented a thesis titled, ***Screen Space Reflection Techniques,*** in an oral examination held on November 28, 2019. The following committee members have found the thesis acceptable in form and content, and that the candidate demonstrated satisfactory knowledge of the subject material.

External Examiner:	Dr. Timothy Maciag, Software Systems Engineering
Co-Supervisor:	Dr. Xue Dong Yang, Department of Computer Science
Co-Supervisor:	Dr. Howard Hamilton, Department of Computer Science
Committee Member:	Dr. David Gerhard, Department of Computer Science
Chair of Defense:	Dr. Richard McIntosh, Department of Mathematics & Statistics

Abstract

Ray tracing is a rendering technique in computer graphics that can simulate a variety of optical effects, such as reflection from smooth surfaces, refraction through transparent objects, and light scattering on rough surfaces. Ray tracing can produce visual realism of a higher quality than other rendering techniques, such as rasterization techniques, but at a much higher computational cost. Screen Space Reflection (SSR) is a group of approximation techniques that utilize data already generated by common rasterization techniques, such as deferred shading, to produce limited reflection effects.

Most rasterization algorithms use two basic data structures, an image buffer storing the colour of a surface point visible from the camera at each pixel, and a depth buffer (or Z-Buffer) storing the depth from the camera to the corresponding surface point. For each surface point, SSR techniques generate a reflection ray and project the reflection ray onto the depth buffer. Values in the depth buffer along the projected path are compared with the depth of the reflection ray to determine a potential intersection. A traversal process along the projected path is necessary. If an intersection is found, the corresponding colour in the image buffer contributes to the reflection. Several SSR techniques exists. In this research, SSR techniques are defined using a common algorithm schema and five noteworthy SSR techniques including ray marching, digital differential analyzers (both conservative and non-conservative), and hierarchical depth buffers (both minimum and minimum-maximum) are implemented as instances of this schema for the purpose of performance analysis.

In the performance analysis, the average GPU time for traversing a projected path in the depth buffer was recorded for each of the five SSR techniques on different testing scenes, different image resolutions, and under variable control parameters associated with each technique. The analysis shows a statistically significant difference in average traversal time between different SSR techniques for 98% of the test configurations. Visualizations are also generated to facilitate the analysis. Detailed analysis results are presented in the thesis.

Acknowledgements

I would like to acknowledge all those who assisted me in the preparation of this thesis. Firstly, I would like to thank my co-supervisors Dr. Howard Hamilton and Dr. Xue Dong Yang, both of whom provided me with invaluable guidance during my research. Dr. Howard Hamilton provided me with helpful comments and advice during the writing process of my thesis. I would like to thank Dr. Xue Dong Yang for his many insights into the field of computer graphics and how he pushed me to explore difficult questions and discover the answers on my own. I appreciate Dr. David Gerhard's service on my supervisory committee. I would also like to thank Leigh Anne MacKnight for her consultation regarding statistics.

I would like to acknowledge the Natural Sciences and Engineering Research Council of Canada for funding from a Discovery Grant awarded to Dr. Hamilton. The Faculty of Graduate Studies and Research and the Department of Computer Science at the University of Regina also provided financial support.

Post Defense Acknowledgements

I would like to thank Dr. Timothy Maciag for serving as the external examiner for my defense and Dr. Richard McIntosh for being the chair of defense.

Table of Contents

List of Tables

List of Figures

Chapter 1 Introduction

Screen Space Reflection(s) (SSR) is a family of computer graphics techniques for reusing screen space data to calculate reflections in rendered images. These techniques operate in the *screen space* coordinate system, which is the coordinate system of the pixel grid. The main idea behind SSR is to sample points along the projection of a ray of light across the depth buffer, which stores the depth of the nearest surface in each pixel. The sampled depth values are compared with the depth of the ray to determine if an intersection has occurred between the ray and the nearest surface. If an intersection is found, the colour value of the rendered image at the screen space intersection point contributes to the colour of the reflection.

In this thesis, the term *SSR technique* refers to any of the techniques in the SSR family and the term *SSR algorithm* refers to the common algorithm that underlies all these techniques. In the literature, the term "SSR" is sometimes used to refer to reflections computed in screen space, but here it is restricted to referring to the family of techniques. Other terms for SSR are screen space ray traced reflections, real-time local reflections, and screen space local reflections.

SSR is a form of screen space ray tracing. There are other forms of screen space ray tracing, such as screen space ambient occlusion [1] and screen space refraction rendering [2], but this thesis focuses on SSR specifically. *Ray tracing* [1] is a class of computer graphics rendering algorithms in which the reverse paths of rays of light are traced to find an intersection with geometry in the scene. A *scene* is a collection of geometric objects (called *geometry*), materials, and lights that specify the virtual world that is being rendered. *Primary rays* are generated in such a way that each ray is emitted from the center of

projection of a camera, passes through a pixel on the image plane, and casts into the scene. Intersection calculations are performed for each ray against the scene geometry to determine which surface is intersected by each ray. If an intersection is found, the primary ray can interact with the surface, such as by reflecting off the surface, to create secondary rays and so forth. Any secondary ray or subsequent ray that reflects off a surface is called a *reflection ray*. The lighting contributions of the primary ray and any reflection rays are combined to determine a colour value for the primary ray's corresponding pixel. With SSR, the goal is to perform ray tracing for reflection rays using only data available in screen space.

SSR techniques have been used in the video game industry since approximately 2010 [3] [4] [5] [6] [7]. SSR was first presented in 2011 in the context of video games by Souza et al. [3], although the video game *Just Cause 2* may have utilized SSR before this in 2010 [8]. Concepts that share some similarities to the traversal process used with SSR existed before this as well [9] [10].

SSR techniques are a relatively cheap way to render reflections when compared with fully ray traced solutions. Though the quality is not as good as a fully ray traced solution, decent results can be obtained quickly. Optimizations can improve the speed of SSR, but they may also lower the quality of the results. Even with quality reducing optimizations, a decent reflection can still be generated, which makes SSR a good choice for real time graphics applications such as video games. Table 1-1 lists some of the advantages and disadvantages of SSR.

SSR reuses data that is already being produced by deferred shading, a common rendering algorithm employed by several video games. Deferred shading [1] generates a

Table 1-1: Advantages and disadvantages of SSR.

Advantages	Disadvantages
✓ Relatively fast	✗ View dependent
✓ Decent reflections	✗ Incomplete scene representation
✓ Screen space data reuse	✗ Occlusion causes artifacts
✓ Can optimize speed for quality	✗ Long rays require more samples

Geometry Buffer (or G-Buffer) that contains all the information needed to perform shading calculations for each pixel. This information can include per pixel diffuse, specular, surface normal, and depth values. The normal and depth information in the G-Buffer is sufficient for constructing a reflection ray. Thus, SSR can be added to a deferred shading pipeline with relative ease by utilizing the data in the G-Buffer. Nonetheless, some SSR techniques generate additional data structures to improve speed or quality.

Since SSR uses screen space data to render reflections, it is dependent on the view. Scene information, such as the depth buffer and normal vectors stored in the G-Buffer is only available for the main camera view. Thus, it is impossible to determine the intersection point for rays that travel outside of the view, such as rays that bounce back towards the viewer without hitting anything in the view. Thus, situations such as looking directly at a planar mirror cannot be directly handled by SSR. Instead, SSR excels at producing reflections along surfaces that reflect rays in the general view direction, because the intersection point is likely within the view.

Occlusion occurs when a point is blocked from a viewpoint by another point that is in front of the original point. The point that is in front of another point is said to occlude that point. Occlusion is a major problem for simple SSR techniques. If a ray travels behind an object, it is not possible to determine which surface the ray strikes because only information concerning visible surfaces is available. The ray might directly hit the backside of the occluding object, it might hit another unknown object that is behind the occluding object,

or it might go past the object and hit another surface that may be in view. It could also pass behind another object and travel out of the view of the camera. The problems caused by occlusion are a major source of artifacts produced by SSR. These problems can be reduced by hiding the artifacts or by generating multiple layers of scene information to potentially obtain correct intersections.

Another problem with SSR techniques is their poor handling of long screen space rays. A *long screen space ray* is a ray that, when projected into screen space, spans a large portion of the screen. A long screen space ray requires the depth buffer to be sampled at many points along the ray's path, which requires many texture samples and consequently lots of memory bandwidth on the GPU.

1.1 Problem Statement

Global illumination is the lighting contributions from not only direct illumination of light sources, but also from other surfaces (indirect illumination) [1]. Performing real-time global illumination remains a challenging problem in the field of computer graphics. Many real time global illumination algorithms use precomputed acceleration data structures. An SSR technique can be considered to be an image-based technique, that partially addresses the real-time global illumination problem, specifically for specular global illumination. *Specular global illumination* is the lighting contribution from specular reflection. *Specular reflection* occurs when light reflects according to the law of reflection, such as in the case of a mirror. The SSR algorithm involves calculating specular reflections for reflective surfaces to obtain reflections of other surfaces. Due to the dependence of SSR techniques on screen space information, they compute approximations rather than full solutions.

Generally, the problem to be solved is to create realistic renderings of dynamic 3D scenes in real time, with reflective or refractive effects. The goal of *photorealistic rendering* is to render an image that accurately obeys the physics of light. State of the art rendering systems are capable of producing nearly photorealistic images, but this currently cannot be done in real time. Thus, for those who require real-time performance, the objective is to render an image in real-time that approximates the physics of illumination with as few artifacts as possible.

The objective of this thesis is to provide a comprehensive review, critical analysis, and empirical performance evaluation of SSR techniques. Given the requirement of real-time rendering performance, this thesis presents an analysis of the strengths and limitations of a class of ray tracing techniques that utilize only the limited screen space information already generated with respect to the viewpoint.

1.2 Motivation

SSR is commonly used in video games to add specular global illumination for surfaces within the camera's field of view. Unfortunately, unoptimized SSR implementations can be computationally expensive. Thus, it is beneficial to have an optimized SSR implementation. Since, there are several different SSR techniques, each with their own strengths and weaknesses, it is important to understand them before deciding which technique to use.

Several existing sources provide overviews of SSR. This information is available in the background chapters of SSR papers, graphics textbooks, and online tutorials describing SSR implementations. However, at the time of writing, there is no known source that provides a comprehensive empirical evaluation of SSR techniques. Although some SSR

papers or presentations provide evaluations of their respective algorithms, perhaps with comparison to few other methods, they do not provide comprehensive evaluations.

1.3 Contributions

This thesis provides a comprehensive and up to date survey of SSR techniques. The SSR techniques presented in this thesis are not. Instead, this thesis provides novel a framework for evaluating the performance of SSR techniques. In general, the framework was designed with the intent of keeping all parts of the rendering process equal, except for the differing SSR techniques and their parameters. To do so, the SSR techniques were defined in terms of a common algorithm schema. This approach increases the fairness of the comparison between SSR techniques. Any remaining differences in implementation are clearly specified. This thesis also contributes an empirical evaluation and critical analysis of the SSR techniques. Although some results from this evaluation may be known to video game industry professionals, they are be presented here in publicly accessible format. The evaluation has several limitations, such as the exclusion of glossy reflections for rough surfaces.

1.4 Outline

SSR has been described briefly in Chapter 1. In Chapter 2, SSR is described in more detail. The SSR algorithm, as well as several SSR techniques are covered. Some background knowledge on computer graphics and linear algebra is assumed. Chapter 3 describes the evaluation performed to compare the various SSR techniques. The results of the evaluation are discussed in Chapter 4, while Chapter 5 presents the conclusion drawn from the study performed in this thesis.

Chapter 2 Background

This chapter reviews SSR and examines several variants of the SSR algorithm. First, deferred shading, a common requirement for SSR, is discussed in Section 2.1. The SSR algorithm is discussed in Section 2.2. This algorithm requires traversing the path of a ray across the screen until it intersects with an object within the view. However, there are several ways to traverse a screen space ray to determine a ray intersection. The traversal process requires a method for determining whether a ray has intersected the scene. In this thesis, such a method is referred to as an intersection scheme. Intersection schemes are formally defined in Section 2.3. Traversal algorithms, which are referred to as traversal schemes in this thesis, are formally defined in Section 2.4. SSR typically operates on only the surfaces that are visible to the user camera. Approaches that generate multiple layers of scene information are covered in Section 2.5. After the point of intersection between a ray and the surface, which is called a hit point, is determined using a traversal scheme, a shading value needs to be determined for the hit point. Section 2.6 discusses techniques for doing so, while Section 2.7 discusses how to hide some of the artifacts created by SSR due to incomplete scene information. Some techniques use multiple views to have access to more scene information. These multi-view techniques are covered in Section 2.8.

2.1 Deferred Shading

Deferred shading [1] is a rendering process that decouples the shading calculations from the rasterization of the scene. Deferred shading works by performing rendering in two main steps: the geometry pass and the lighting pass (or passes). In contrast, *forward shading* is the more traditional rendering process that simply shades pixels as geometry is submitted to the graphics pipeline. Both forward and deferred shading can suffer from

overdraw, which occurs when a pixel value calculated when rendering one object is overwritten by another value calculated when another object is rendered over top of it. However, when overdraw occurs with forward shading, calculations performed to light the pixel are wasted since the value is overwritten. For complex shading models, the cost of overdraw starts to add up. To avoid the cost of wasteful shading calculations, deferred shading instead collects all the properties required to evaluate the shading for every pixel and stores them in buffers. This is the geometry pass. The actual shading calculations are deferred to a separate pass called the lighting pass. Thus, each pixel is only shaded once. Overdraw can also be limited by techniques such as coarse depth sorting [1].

To apply an SSR technique, it is necessary to have depth and normal information for every pixel. Deferred shading naturally produces these two pieces of information, which makes deferred shading a natural rendering algorithm for SSR. Depth and normal information are required to determine how a ray interacts with the surface in a pixel including how it reflects. However, SSR does not necessarily require deferred shading. SSR could theoretically use forward shading and determine the surface normal vectors by reconstructing them using the depth buffer and screen space partial derivatives. However, these reconstructed normal vectors could contain artifacts at depth discontinuities if special care is not taken. Also, reconstructed normal vectors only describe the polygonal surfaces of the scene's geometry. Any modification of surface normal vectors by techniques such as normal mapping will not be included in the reconstructed normal vectors. Due to these limitations, SSR is often used with deferred shading as the rendering algorithm.

An overview of the deferred shading process with SSR is shown in Figure 2-1. As previously mentioned, the first step in deferred shading is the geometry pass. The geometry

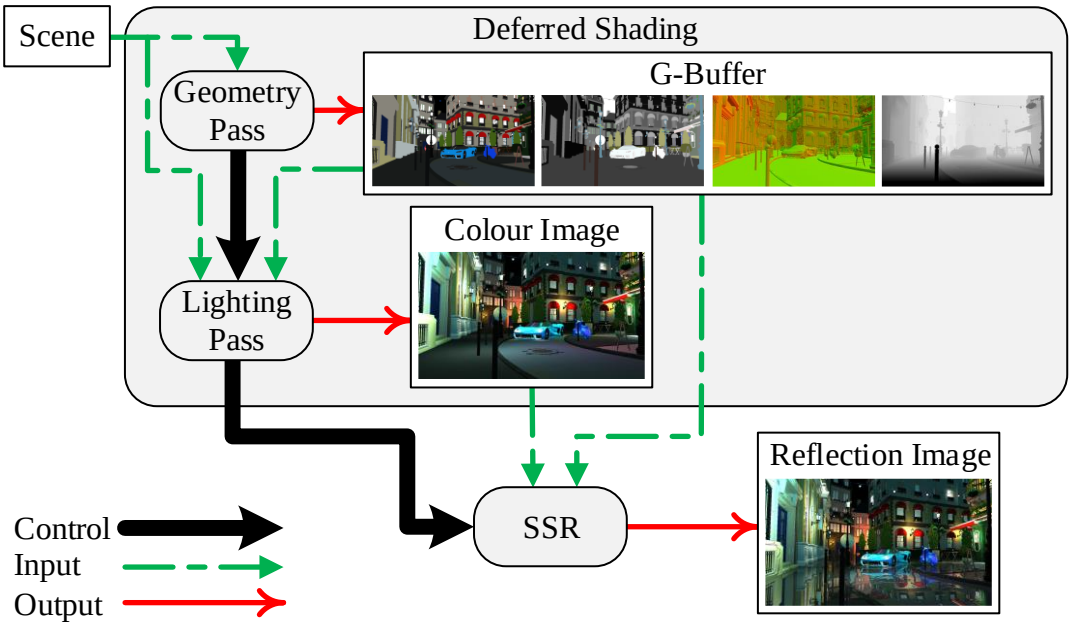
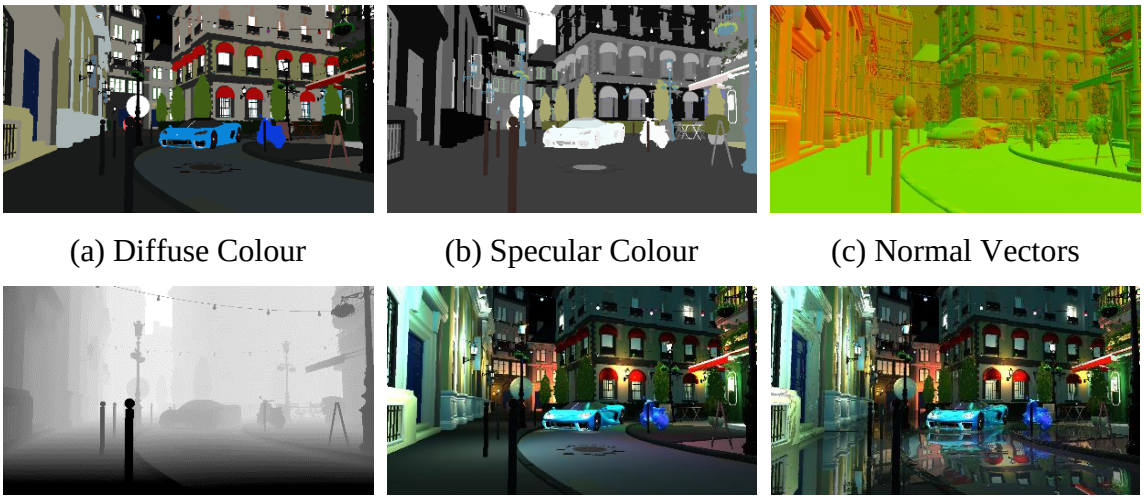


Figure 2-1: Control and data flow for deferred shading and SSR. (original in colour)

pass works by first rendering all properties needed to shade the scene into separate buffers which together are called the *Geometry Buffer* or *G-Buffer*. The exact format of a G-Buffer depends on what shading model is used for rendering. In general, the G-Buffer must include normal vectors, depth information and any material properties that affect the shading. The example G-Buffer shown in Figure 2-2, contains diffuse (Figure 2-2a) and specular colour (Figure 2-2b), or more precisely, the material diffuse reflection coefficients and the material specular reflection coefficients. View space normal vectors are shown in Figure 2-2c. In this example, the normal vectors are encoded as two 16-bit scalars with a Lambert azimuthal equal-area projection [11]. This encoding scheme for the normal vectors could be replaced by any other encoding scheme, such as ones that exhibit fewer encoding errors [12]. A depth buffer is also a part of the G-Buffer, as shown in Figure 2-2d.

In the lighting pass, light culling is performed and the light contributions for each pixel are evaluated. *Light culling* is the process of reducing the number of lights that need to be evaluated for a pixel. Ideally, only the lights that affect a pixel should be evaluated for that



(a) Diffuse Colour (b) Specular Colour (c) Normal Vectors

(d) Depth Buffer (e) Lighting Pass (f) Reflections

Figure 2-2: G-Buffer properties (a) through (d) are generated by the geometry pass. The lighting pass generates the lit image in (e). Finally, SSR is applied to the image in (f). (original in colour)

pixel. Since lights can be culled in multiple ways, there are multiple ways to perform the lighting pass. The lighting pass can be done in a single pass that evaluates all the lights or in multiple passes where each pass evaluates the lighting contribution of pixels that are affected by one light.

One way to perform the lighting pass is to use a multi-pass model with stenciled light volumes, where the results of each pass are accumulated into an output buffer [13]. With this model, light culling is performed by drawing a simplified mesh of the light's volume of influence with a read only depth buffer. A stencil buffer is used to keep track of whether the light affects the pixel. If the depth test fails, the stencil value is incremented for the front faces of the simplified light volume mesh and decremented for its back faces. After the light volume mesh is rendered, if a pixel has a stencil value greater than zero, it indicates that the light affects the pixel, otherwise the light is culled. Special care must be taken when the camera is inside the light volume.

The multi-pass model with stenciled light volumes allows deferred shading to evaluate shading calculations only for pixels that are affected by a particular light. In contrast,

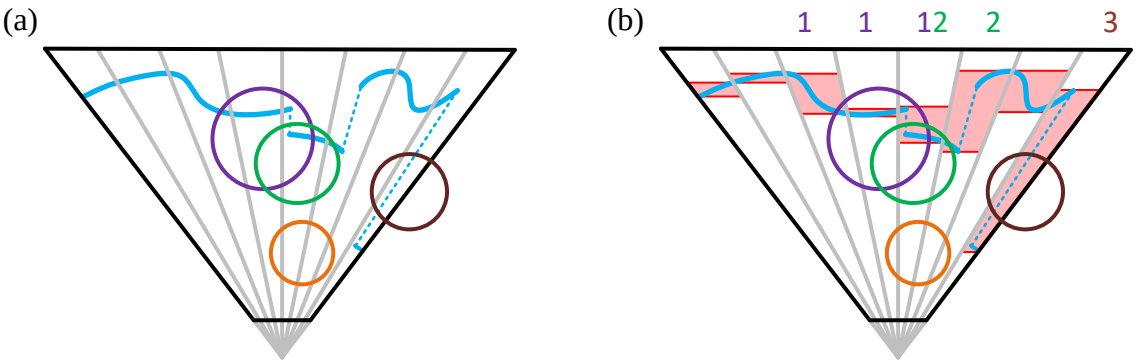


Figure 2-3: Tiled deferred shading. (a) Top down view of tile frustums and point lights. (b) Tile depth bounds (red) and culled lights (numbers at top). Adapted from [1]. (original in colour)

classic forward shading needs to evaluate the contribution of each light for a given pixel. The output of the lighting passes is shown in Figure 2-2e. After deferred shading has been performed, SSR can be applied to the lit image in Figure 2-2e to produce the result in Figure 2-2f.

Performing the lighting pass with the multi-pass model described above does have a major problem. Each time a light is processed, the surface properties stored in the G-Buffer must be read from GPU memory. Thus, for pixels that have many contributing light sources, the surface properties must be loaded from the G-Buffer into GPU registers multiple times. One solution to this problem is to perform tiled deferred shading [1] [14] [15].

Tiled deferred shading divides the screen into a grid of equal sized tiles. Each tile represents a 3D subvolume of the view frustum. The *view frustum* is the volume or truncated pyramid shape that contains the field of view of the camera. In tiled deferred shading, the volume a tile occupies is a skewed frustum or subvolume of the view frustum. Figure 2-3a shows a top down view of the view frustum subdivided into tiles along with the sphere of influence of several point lights.

For each tile, a list of lights that intersect the tile is generated (shown in Figure 2-3b). Lights that do not intersect the tile are culled. The light culling process for a tile requires the depth bounds, which consists of both the minimum and maximum depth values of any pixel inside the tile. The depth bounds can be computed using the hierarchical depth buffer approach discussed in Section 2.4.2 or by using parallel reduction [15] [16]. By knowing the depth bounds of the tile, a bounded, skewed frustum of the tile can be constructed (shown in light red in Figure 2-3b). Only lights which intersect this volume can affect the pixels in the tile. Accurate tile frustum planes can be constructed or precomputed to define this volume. Alternatively, a simplified *axis aligned bounding box* (AABB) can be constructed based on the tile depth bounds [15]. An AABB, however, may include false positive lights that do not affect the tile.

Tiled deferred shading can be performed by dispatching a compute shader to perform light culling to store lists of lights that affect the tile to shared memory. Each thread performs culling on a light inside of its tile until there are no more lights. Then each thread is synchronized with the thread group and switches into shading mode to loop over the list of lights for the tile to evaluate the lighting in the thread's pixel. The shading process involves only one sampling of the G-Buffer per pixel as opposed to the potentially multiple samplings of basic non-tiled deferred shading with multi-pass stenciled light volumes.

One problem with tiled deferred shading is that the tile depth bounds can cover large distance intervals. This problem can easily occur at depth discontinuities and can result in the tile depth bounds including many lights in the center of the interval that may not affect any pixels in the tile. Such a case is shown for the brown coloured light (rightmost light) in Figure 2-3b. There are solutions to this problem, such as subdividing the depth bounds

in half [15] or using a uniformly subdivided depth bounds with a bitmask for which subdivided ranges have lights and geometry [1]. However, this problem and its possible solutions are not further discussed in this thesis.

After the G-Buffer has been constructed via deferred shading or tiled deferred shading, the SSR algorithm can begin. As mentioned, this process reuses the G-Buffer data that was required for shading. Tiled deferred shading was used in this thesis to perform deferred shading for multi-layered SSR approaches. Multi-layer approaches are discussed in Section 2.5. Due to implementation choices, the multi-layer generation used in this thesis does not have access to a stencil buffer for the multi-pass deferred shading with stenciled light volumes, so tiled rendering was used instead.

2.2 The SSR Algorithm

The main idea of SSR is to reuse screen space information to perform ray tracing against the depth buffer of the scene rather than against the scene geometry, which is typically a collection of triangles. The depth buffer holds a representation of the visible surfaces in the scene. Traversing the path of a reflection ray across the depth buffer is analogous to tracing a ray across a height field.

Some intuition for SSR is shown in Figure 2-4. Take any point P in Figure 2-4a that has a reflection and look for the corresponding point Q in the image that the first point reflects. The line segment PQ defined by these two points is described as being along the reflection ray in screen space. The SSR algorithm works calculating a reflection ray starting at P and sampling the depth buffer at pixels along the ray, as shown in Figure 2-4b. In this figure, smaller values in the depth buffer are indicated by darker pixels and larger values are indicated by lighter pixels. The transition from white to red on the ray sample points

(a)

(b)

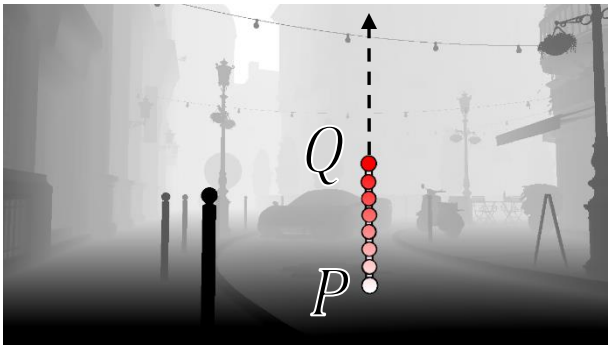


Figure 2-4: (a) A screen space reflection ray is shown superimposed over the final image with reflections. (b) Depth Buffer pixels along the ray are sampled to find an intersection point. (original in colour)

indicates the increasing number of samples required to find an intersection. If the ray intersects the depth plane defined by the depth buffer value in a pixel, then an intersection has occurred within the pixel Q. The *intersection point* is defined as the point where the ray intersects the depth plane within the pixel or sample point.

Before the SSR algorithm can begin, depth and surface normal information is required as input. The deferred shading algorithm discussed in Section 2.1 generates this information in a G-Buffer. The basic process for SSR is shown in Figure 2-5 and consists of the following steps:

1. Ray Construction
2. Ray Traversal (using a traversal scheme)
3. Ray Shading

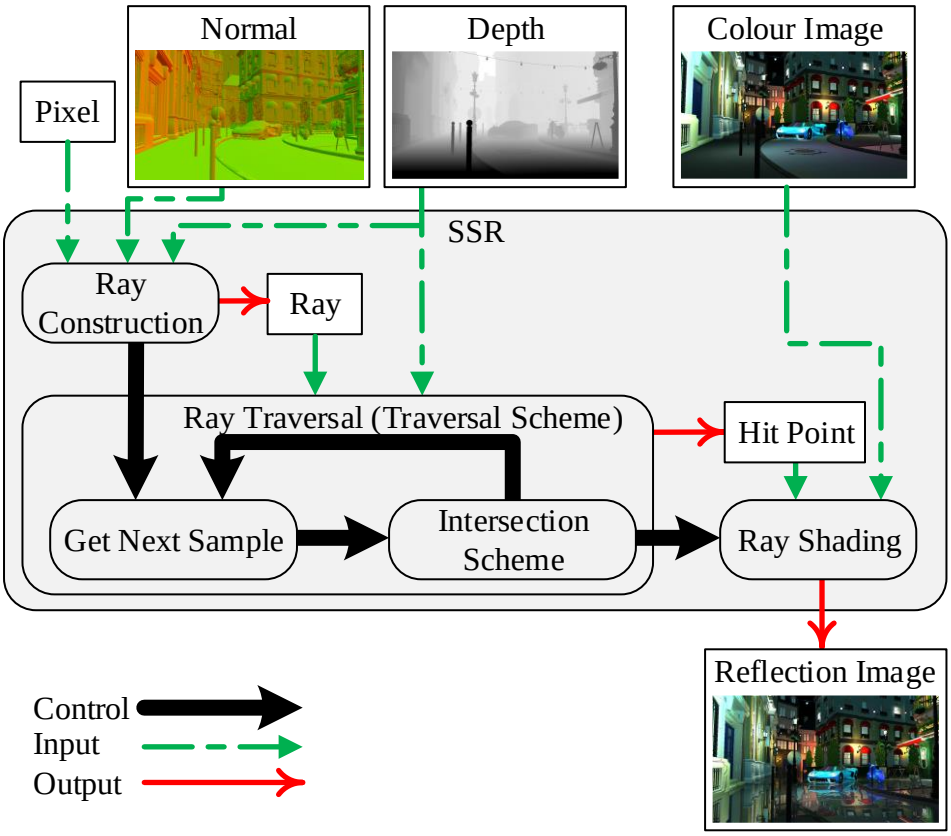


Figure 2-5: Control and data flow diagram for SSR. (original in colour)

Step 1 involves calculating the screen space reflection ray for a given pixel. In Step 2, the path the reflection ray is traversed across the depth buffer in screen space to find a potential intersection. If one is found, it is called a *hit point*. Finally, in Step 3, a reflection colour is determined for the hit point and applied to the rendered image. Steps 1, 2, and 3 can be implemented in a single screen pass pixel shader using a screen aligned triangle or in a compute shader. Alternatively, steps 1 and 2 can be performed together and step 3 can be performed in a separate pass to apply filtering effects before determining a reflection value.

2.2.1 Ray Construction

Here, the ray construction process is described, after giving some notation. A subscript is given to vectors and points to denote the coordinate system in which they are defined.

The subscript SS denotes screen space points and the subscript VS denotes view space (or camera space) points. The subscript HS denotes homogeneous space or homogenous clip space. Dot notation is used to access a parameter or component of a structure. This notation should not to be confused with dot product notation. For example, $p.z$ refers to the z component of a point p in a 3D coordinate system. A left-handed coordinate system, as is often used with Direct3D, is used for mathematical definition and formulations in this thesis. The mathematics would be slightly different if a right-handed coordinate system was used.

After the G-Buffer has been constructed via deferred shading, a depth buffer D and a normal buffer $\vec{N}$ are available. Elements from the depth buffer and normal buffer are specified as $D(u,v)$ and $\vec{N}(u,v)$, respectively. Here, $u \in [0,W]$ and $v \in [0,H]$ are the screen space coordinates of a pixel, where W and H represent the width and height of the image resolution, respectively. The screen space coordinate u is at the center of a pixel when $u = x + 0.5$ for any integer value $x \in [0,W-1]$ and similarly for v. The depth buffer can store different representations of depth, including hyperbolic or linear depth. If hyperbolic or normalized linear depth is used, then $D(u,v) \in [0,1]$. If unnormalized linear depth is used, then $D(u,v)$ is defined with a range such that $D(u,v) \in [z_n, z_f]$, where z_n and z_f are the near and far clipping plane distances, respectively. The normal buffer can store either world space or view space normal vectors.

To traverse the path of a reflection ray across the screen, the ray must be projected into screen space. The parametric ray equation for any ray is $r(t) = o + \vec{d} \times t$, where o is the origin of the ray and $\vec{d}$ is the direction of the ray. Any point along the ray can be uniquely

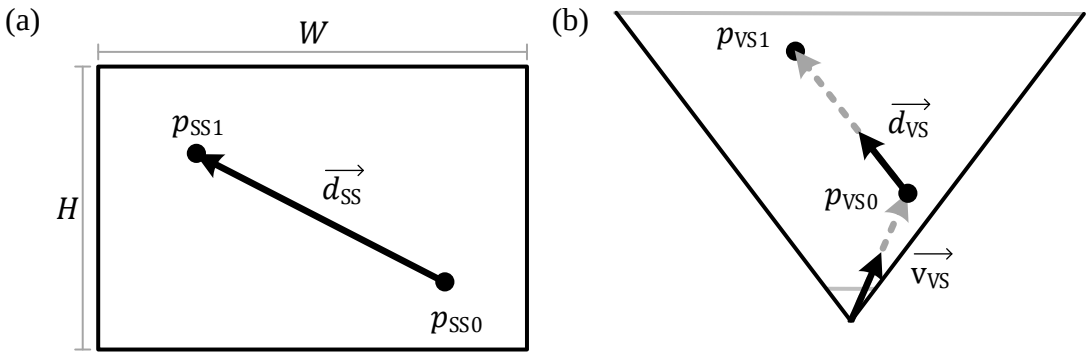


Figure 2-6: (a) Screen space ray. (b) View space ray (top down view).

determined by varying the parametric distance parameter t. To construct a screen space ray for a pixel (u, v), an origin point and a direction vector must be obtained in screen space.

To determine the direction of the screen space ray, two points along the ray, p_{SS0} and p_{SS1}, are used. The point p_{SS0} is the origin of the screen space ray at pixel (u, v). Point p_{SS1} is another point along the ray, at the maximal distance to be traversed along the ray. These two points define a 2D ray in screen space. The depth values at these two points are also needed in order to interpolate a depth value between these points. Thus, the depths p_{Z0} and p_{Z1} are defined as the corresponding depth values for p_{SS0} and p_{SS1}. The depth p_{Z0} at point p_{SS0} is known since it is available in the depth buffer at $D(u, v)$.

Given two screen space points p_{SS0} and p_{SS1}, a screen space direction vector can be computed as $\overrightarrow{d_{SS}} = p_{SS1} - p_{SS0}$. Thus, the screen space ray is simply $r_{SS}(s) = p_{SS0} + \overrightarrow{d_{SS}} \times s$ (shown in Figure 2-6a). Here, the parameter $s \in [0,1]$ represents the screen space distance along the ray. Setting s to 1 yields the maximal distance that the ray traverses.

The ray depth is computed differently depending on whether the depth components are hyperbolic or linear. If they are hyperbolic, ray depth can be interpolated between p_{Z0} and p_{Z1} as $r_Z(s) = p_{Z0} + \overrightarrow{d_Z} \times s$, where $\overrightarrow{d_Z} = p_{Z1} - p_{Z0}$. Note that $r_Z(s)$ can only be computed with this equation when the depth components are hyperbolic because $\frac{1}{Z}$

interpolates linearly in screen space. Linear depth values do not interpolate linearly in screen space. Instead, perspective correct interpolation [17] must be applied, which can be performed using the equation:

$$r_Z(s) = \frac{1}{\frac{1}{p_{Z1}} + s \times \left(\frac{1}{p_{Z1}} - \frac{1}{p_{Z0}}\right)}$$

If the ray end point p_{SS1} is out of bounds with respect to the screen, p_{SS1} and p_{Z1} can be clipped to the screen borders. However, clipping is not necessary since it can be assumed that a point along the ray is out of bounds when a depth buffer sample has a value of zero. This condition can be used to terminate the traversal process discussed in Section 2.2.2. [8]. This assumption works well with Direct3D and other Graphics APIs since loading a value from a texture with an out of bounds index is defined to return a value of zero.

The point p_{SS1} can be determined by finding a point along the view space reflection ray and performing a projection to screen space. Using the parametric ray equation, the view space reflection ray can be defined as $r_{VS}(t) = p_{VS0} + \overrightarrow{d_{VS}} \times t$ (shown in Figure 2-6b). The view space point p_{VS0} is the hit point of the primary ray through pixel p_{SS0} and $\overrightarrow{d_{VS}}$ is the view space direction vector.

The view space point p_{VS0} that corresponds to the screen space point p_{SS0} can be reconstructed from (u, v), $D(u, v)$, and the clipping planes z_n and z_f. If D contains non-normalized linear depth values, $p_{VS0}.z$ is assigned $D(u, v)$. Otherwise, $p_{VS0}.z$ is assigned to $L(D(u, v))$, where $L(x)$ converts hyperbolic depth to linear depth.

Since, the view space point p_{VS0} is the hit point of the primary ray through pixel p_{SS0}, the primary ray direction or *view vector* is simply $\overrightarrow{v_{VS}} = \frac{p_{VS0}}{|p_{VS0}|}$. The reflection ray direction

$\overrightarrow{d_{VS}}$ can be determined by reflecting $\overrightarrow{v_{VS}}$ about the surface normal $\vec{N}(u,v)$, so $\overrightarrow{d_{VS}} = reflect\left(\overrightarrow{v_{VS}}, \vec{N}(u,v)\right)$, where $reflect$ is a function that calculates a reflection vector according to the law of reflection. Recall that the normal buffer can store either world space of view space normal vectors. If world space normal vectors are stored, then a conversion to view space must be applied at this stage using a view matrix.

To calculate p_{SS1} and the corresponding depth p_{Z1}, another point on the view space reflection ray can be calculated using the parametric ray equation. Point p_{VS1} on the ray r_{VS} can be generated using a parametric t value that is sufficiently large or set to the maximum distance to be traversed. To project p_{VS1} onto the screen, a projection matrix P is required.

Transformation by the projection matrix P requires that the 3D point p_{VS1} be converted to the homogeneous 4D point $(p_{VS1}, 1)$. In general, a homogeneous coordinate is of the form (x, y, z, w), where $w \neq 0$. Transforming $(p_{VS1}, 1)$ by multiplying by the matrix P yields a transformed homogeneous coordinate $p_{HS1} = (p_{VS1}, 1) \times P$. Before the transformation, $(p_{VS1}, 1).w = 1$, but after the transformation, w is not necessarily equal to 1. Specifically, $p_{HS1}.w$ represents the linear depth under the perspective transformation P. The projection matrix is defined as the matrix $P = P_p \times V$, where P_p is a perspective projection matrix and V is a viewport matrix. The viewport matrix converts from *normalized device coordinates*, a normalized representation of the screen where $x \in [-1,1]$, $y \in [-1,1]$, and $z \in [0,1]$ with homogeneous coordinate w, to screen space coordinates where $x \in [0, W]$, $y \in [0, H]$, and z and w are unchanged. The viewport matrix is applied before perspective division, which performs a projection onto the $w = 1$ plane, to avoid an extra matrix multiplication [8]. Via perspective division, the screen space point

is $p_{SS1} = \frac{p_{HS1}.xy}{p_{HS1}.w}$. If hyperbolic depth is used, the depth of the screen space point resulting from perspective division is $p_{Z1} = \frac{p_{HS1}.z}{p_{HS1}.w}$. If linear depth is used, then $p_{Z1} = p_{HS1}.w$. To avoid a division by zero in case of perspective division, the ray end point should be clipped against the near plane. However, clipping against the near plane is only required for rays that point towards the camera, since rays that point away from the camera cannot be behind the near plane.

2.2.2 Ray Traversal

Once the screen space ray, represented by a pair $(r_{SS}(s), r_Z(s))$, is calculated for a given pixel (u, v), the ray traversal process can begin. The ray is represented as a pair here instead of as a single function, because linear depth does not interpolate linearly in screen space. The two parts are kept separate to differentiate linear and hyperbolic depth. Both depth representations can be used with SSR. In order to perform the ray traversal process, a method for determining which points to sample along the ray is required. Since several techniques can be used to perform this task, the ray traversal process is modeled in terms of an algorithm schema. A *traversal scheme* is a method for determining which points or pixels to sample along the path of a ray. The output of a traversal scheme is a screen space hit point or a special value indicating that there is no intersection, which is appropriate if there is not enough screen space information to determine an intersection. A traversal scheme is simply a function which takes a ray r and a depth buffer D and determines an intersection point q_{SS}:

$$q_{SS} = traversalScheme(r, D)$$

The ray $(r_{SS}(s), r_Z(s))$ is often used for the r parameter of the traversal scheme, but some approaches use r_{VS} and project $r_{VS}(t)$ to screen space at each step during the traversal.

```
float2 traversalScheme(r, D)
{
    while (still points on ray r to check)
    {
        p = next point on ray r
        (r_ZMin, r_ZMax) = getRayIntervalInPixel(p.uv)

        if (intersectionScheme(r_ZMin, r_ZMax, p.uv, D))
            return p.uv
    }
    return NO_INTERSECTION
}
```

Figure 2-7: Pseudocode for traversal scheme.

Several types of traversal schemes are outlined in Section 2.4. The outline of a general traversal scheme is given in the algorithm in Figure 2-7.

A traversal scheme iteratively compares the ray to the depth buffer until an intersection is found or the ray misses. To be able to perform the traversal, there needs to be a way to determine if the ray has intersected the depth buffer. In this thesis, such a method is referred to as an *intersection scheme*. An intersection scheme is a boolean function that often appears as follows:

$$intersectionScheme(r_{ZMin}, r_{ZMax}, uv, D)$$

where r_{ZMin} and r_{ZMax} define the depth interval that the ray covers in a pixel. The intersection scheme simply returns whether there is an intersection or not. Often, r_{ZMin} is omitted and the intersection scheme is defined as whether r_{ZMax} is greater than or equal to $D(u, v)$. Several intersection schemes can be used; they are discussed in more detail in Section 2.3.

2.2.3 Ray Shading

After the ray traversal process has finished, a screen space hit point q_{SS} is generated for every pixel. The hit point q_{SS} can then be used to look up a colour value stored in the

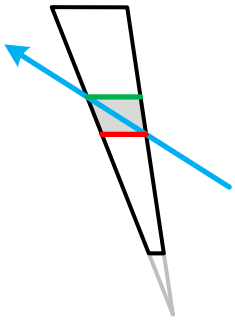


Figure 2-8: Skewed frustum of a pixel. Maximal ray depth r_{ZMax} is shown in green. Minimal ray depth r_{ZMin} is shown in red. (original in colour)

colour buffer, which can be used to approximate the reflection colour. Ray shading is discussed in more detail in Sections 2.6 and 2.7.

2.3 Intersection Schemes

The SSR algorithm uses an iterative process to find an intersection with the depth buffer by using a traversal scheme where either one pixel or one sample point is considered in each iteration. To determine whether a ray intersects the depth buffer in a pixel, the depth of the ray must be compared with the depth buffer sample for the same pixel. As a screen space ray travels across a pixel, it occupies a depth interval within the skewed frustum volume of the pixel. The minimum and maximum depth values of the ray's depth interval in a pixel are shown in Figure 2-8. A *depth buffer sample* is equivalent to a z-axis aligned plane inside of the skewed pixel frustum. Thus, the ray intersects the depth buffer if the depth plane is intersected by the depth interval of the ray in the pixel. The depth buffer samples are represented in Figure 2-9a. The maximum ray depth in one traversal iteration can also be stored and used as the minimum ray depth of the next iteration to avoid recomputing it, as noted by McGuire and Mara [8].

However, simply comparing the depth of the ray with the depth sample has issues where the ray can pass between samples. A thickness parameter is often used with SSR to

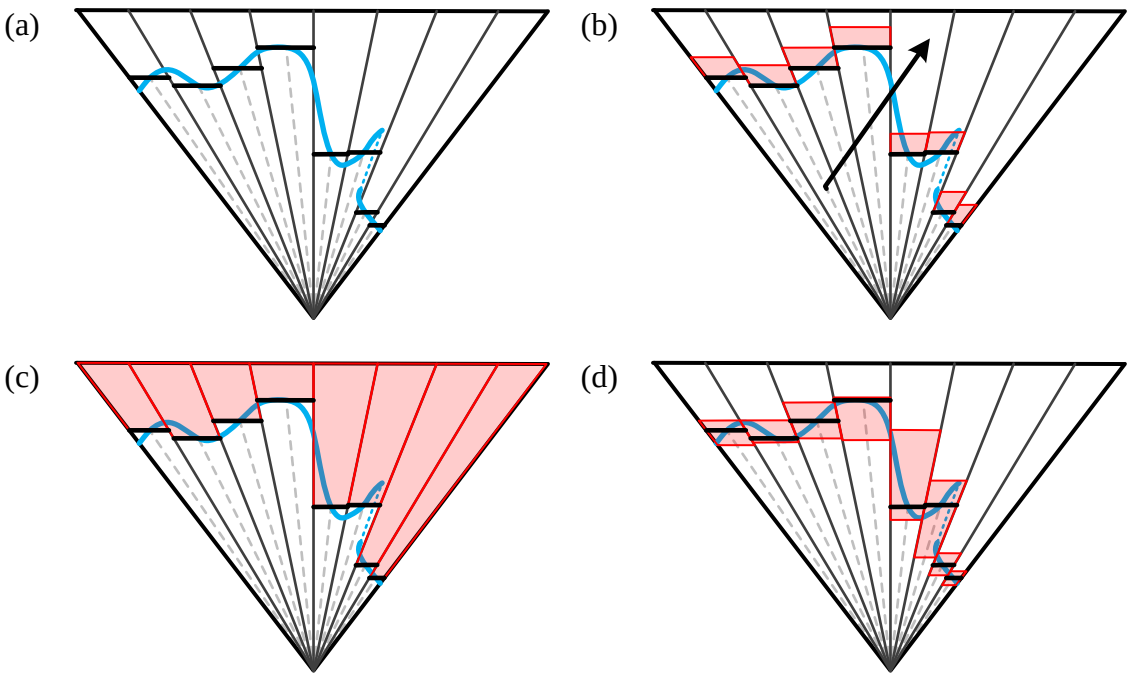


Figure 2-9: Depth thickness schemes. The blue surface represents the surface of the depth buffer. (a) The depth values of the pixel center-points. (b) Constant depth thickness. (c) Infinite thickness (d) Conservative clipped depth. (original in colour)

give each depth sample a depth interval as shown in Figure 2-9b. A constant depth thickness value can be applied to each depth sample. This changes the intersection scheme to the intersection of two depth intervals: the depth interval of the ray in the pixel and the depth interval of the depth thickness. With a constant depth thickness, the ray can still pass through a surface if the distance between neighbouring depth samples is greater than the constant depth thickness as shown by the ray in Figure 2-9b. This becomes more apparent at glancing angles and can cause artifacts. A per-material thickness parameter [18] can also be used, but this requires adding another parameter and can still cause ray skipping in certain cases.

The thickness parameter must be applied to linear depth values. It is incorrect to apply a thickness to a standard hyperbolic depth value. Either the linear depth value must be generated and used directly or a conversion between linear and hyperbolic depth values

```
bool infiniteThickness(r_ZMin, r_ZMax, uv, D)
{
    sceneZMin = D(uv)
    return (sceneZMin <= r_ZMax)
}
```

Figure 2-10: Infinite thickness intersection scheme. (r_{ZMin} is not used here)

```
bool constantThickness(r_ZMin, r_ZMax, uv, D)
{
    sceneZMin = D(uv)
    sceneZMax = sceneZMin + CONSTANT_THICKNESS
    return ((sceneZMin <= r_ZMax) && (r_ZMin <= sceneZMax))
}
```

Figure 2-11: Constant thickness intersection scheme.

must occur. Conversion may lead to some loss of depth precision. Performing the conversion before the traversal process speeds up the traversal process, since depth values do not need to be converted each time the depth buffer is sampled.

An infinite depth thickness (Figure 2-9c) removes the possibility of the ray passing between depth samples but causes in areas where the ray becomes occluded. An infinite thickness intersection scheme simply checks if the maximum depth of the ray is greater than or equal to the depth buffer value. In practice, this is a special case of the constant thickness parameter when $t_Z \geq z_f - z_n$, where z_f and z_n are the far and near clipping distances, respectively. In such a case, the thickness covers the entire depth range of the view frustum. Pseudocode for an infinite thickness and constant thickness intersection scheme are shown in Figure 2-10 and Figure 2-11, respectively.

Thomas et al. note that the depth differential in screen space becomes lower with increasing resolution or decreasing field of view [19]. Less thickness is required in these cases and they instead define a variable thickness as

$$t_Z = \frac{dtan\left(\frac{\alpha_{FOV}}{2}\right)}{WH}$$

where α_{FOV} is the field of view of the camera, d is the distance to the camera, W is the width of the image and H is the height of the image.

Each of these intersection schemes has limitations and they all fail to capture the entire depth range of the surface within the pixel. Since the depth is only determined by the sample point at the pixel center, the entire visible depth range is not calculated. To capture the entire depth range, a conservative depth interval of the surface inside the skewed pixel frustum would need to be calculated. This idea is shown in Figure 2-9d. However, calculating this accurate depth interval is quite difficult. Vardis et al. [20] perform per pixel primitive clipping to compute an accurate depth interval for each triangle and analytically compute triangle intersection for rays that intersect the depth intervals.

Another technique is to render the front and back faces to determine their respective depths and then to calculate a depth interval as their difference [21]. This depth interval can more closely match the actual thickness of objects.

An intersection scheme often compares depth intervals to determine if there is an intersection. Instead of comparing the depth interval of a ray with the depth interval of the scene, parametric ray distance intervals can be compared instead. A parametric ray distance interval is a range along a ray bounded by t_{Min} and t_{Max}. Two parametric ray distance intervals intersect when the equivalent depth intervals intersect. Using this fact, some traversals schemes can avoid computing the depth along the ray by using the parametric distance instead.

2.4 Traversal Schemes

A traversal scheme describes the method for traversing the path of the ray against the depth buffer. The traversal scheme determines which pixels of the depth buffer to sample.

Table 2-1: Traversal scheme comparison chart.

	Linear				Hierarchical	
	Ray Marching	**NC-DDA**	**NC-DDA (Strided)**	**C-DDA**	**Min Hi-Z**	**Min-Max Hi-Z**
No additional data structures	✓	✓	✓	✓	✗	✗
Skips empty space	✓	✗	✓	✗	✓	✓
Quality not reduced	✗	✗	✗	✓	✓	✓
Fast traversal time	✗	✗	✓	✗	✓	✓
Efficient traversal for occluded rays	✗	✗	✗	✗	✗	✓

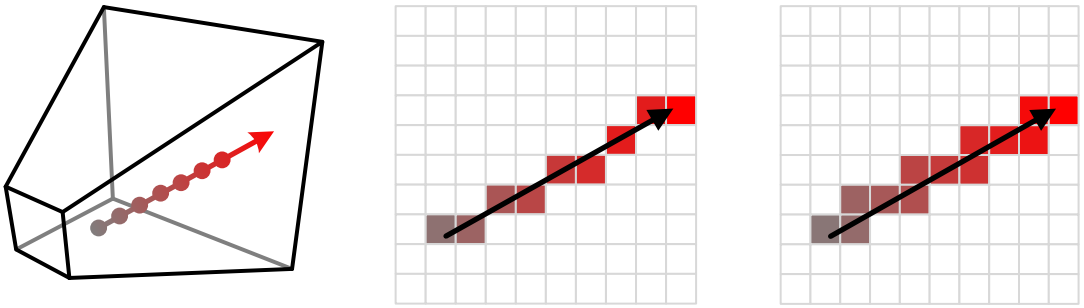
Several different traversal schemes can be used with for SSR ray traversal. Simple linear traversal schemes (covered in Section 2.4.1) use a simple process to determine the next point along the line. Hierarchical schemes (covered in Section 2.4.2) are intended to skip empty space during the traversal process. Table 2-1 shows a chart, which compares several different traversal schemes against different criteria. The traversal schemes in this chart are defined and discussed in the remainder of this section.

Algorithm pseudo-code for the various traversal schemes is given in this section. However, the pseudo-code given in this section is simplified and may not cover every case. Refer to Appendix A through Appendix G for more detailed code that is used in the implementation described in Section 3.2. The code is written in High Level Shading Language (HLSL).

2.4.1 Linear Schemes

A linear traversal scheme is a method that regularly samples points along the ray. Two main types of linear traversal schemes are considered: ray marching and Digital Differential Analyzer (DDA) algorithms.

The simplest SSR traversal scheme is to use simple 3D ray marching. One of the earliest forms of SSR, by Souza et al. [3], uses this approach. Ray marching in 3D works



(a) 3D Ray Marching (b) Non-Conservative DDA (c) Conservative DDA

Figure 2-12: Linear traversal schemes. Red colour indicates the number of depth texture samples required. (original in colour)

```
float2 rayMarch3D(p_VS0, d_VS, D)
{
    deltaT = MAX_RAY_DISTANCE / MAX_STEPS
    for (i = 0; i < MAX_STEPS; i++)
    {
        t += deltaT
        p = p_VS0 + d_VS × t
        p_HS = (p_VS, 1) × P
        p_SS = p_HS.xy / p_HS.w
        p_Z = p_HS.w
        if (intersectionScheme(p_Z, p_Z, p_SS, D))
            return p_SS
    }
    return NO_INTERSECTION
}
```

Figure 2-13: Algorithm for the 3D ray march traversal scheme. Full code sample in Appendix B.

by sampling points along the ray and projecting them into screen space. To sample the points along the ray, a constant parametric distance interval Δt can be iteratively added to the t parameter of the view space ray $r_{VS}(t) = p_{VS} + d_{VS} \times t$ to give $r_{VS}(t + \Delta t) = p_{VS} + d_{VS} \times (t + \Delta t)$. This process is shown in Figure 2-12a and an algorithm is given in Figure 2-13.

However, as McGuire and Mara [8] observe, if these 3D sample points are projected into screen space, problems with over-sampling and under-sampling occur. Due to under-sampling, gaps may occur in the screen space line when the ray is close to the camera and

this can cause a ray intersection to be missed. With over-sampling, some sample points project onto the same pixel as the ray gets farther from the camera. Over-sampling causes wasted computation since the same depth buffer texel may potentially be sampled multiple times.

Performing ray marching in 2D along the screen space ray $r_{SS}(s)$ can help solve the problems of over-sampling and under-sampling. Performing the ray march directly in screen space avoids the conversion from view space to screen space. Another approach that avoids over-sampling and under-sampling is a Digital Differential Analyzer (DDA) or a line rasterization algorithm [8].

A DDA traversal scheme works by traversing the pixels of a ray as if it were rasterized onto a 2D grid. DDA traversal can be either conservative (Figure 2-12c) or non-conservative (Figure 2-12b). A *conservative DDA* (C-DDA) traverses each pixel that a line segment overlaps, while a *non-conservative DDA* (NC-DDA) only checks a single pixel at each iterative step along the main iteration axis. The slope of the 2D line affects how the line is rasterized. For slopes less than or equal to 1, iteration is performed along the x-axis (as shown in in Figure 2-12c), otherwise it is performed along the y-axis. An optimized, NC-DDA for SSR is available [8].

The DDA algorithm must not only determine the screen space pixels of the ray, but it must also be able to determine the depth of the ray in each pixel. As noted by McGuire and Mara [8], the depth of the ray can be interpolated linearly in screen space if it is in homogeneous space. Any property in homogeneous space can be linearly interpolated. Given a view space point p_{VS} along the ray, the point $q = p_{VS} \times k$ and k can be linearly interpolated in 2D, where $k = \frac{1}{p_{HS}.w}$ and p_{HS} is the homogeneous space point

corresponding to the perspective projection of p_{VS}. The original view space point can be obtained by calculating $\frac{q}{k}$. McGuire and Mara [8] perform perspective correct interpolation across the view space ray by linearly interpolating the point q between $q_0 = p_{VS0} \times k_0$ and $q_1 = p_{VS1} \times k_1$, where $k_0 = \frac{1}{p_{HS0}.w}$ and $k_1 = \frac{1}{p_{HS1}.w}$. The interpolation is performed by iteratively adding screen space differentials along the main iteration axis. Iteration is only performed along the x axis to limit dynamic branches in the traversal loop. When the line defined by the ray has a slope greater than 1, the ray is converted to a more horizontal line by swapping the xy components via register swizzling of the screen space points and swapping the data back when required. Differentials of the screen space point p_{SS}, q and k computed as the following, respectively:

$$dP_{SS} = \left(1, \frac{dy}{dx}\right) \times sign(\mathrm{d}x)$$

$$dQ = \frac{q_1 - q_0}{dx} \times sign(\mathrm{d}x)$$

$$dk = \frac{k_1 - k_0}{dx} \times sign(\mathrm{d}x)$$

The direction to step along the x-axis is represented by $sign(\mathrm{d}x)$. Interpolation is performed on p_{SS} between p_{SS0} and p_{SS1}, q between q_0 and q_1, and k between k_0 and k_1. A view space point along the ray can be recovered as $\frac{q}{k}$. During the traversal process, only the z component needs to be interpolated. The interpolated depth is then compared with the depth buffer. Pseudocode for a NC-DDA using this approach is shown in Figure 2-14.

Since a NC-DDA does not traverse every pixel that the ray intersects, it can potentially miss an intersection with the depth buffer. A C-DDA does not miss these types of intersections because it checks every pixel that the ray intersects. However, a C-DDA can

```
float2 nonConservativeDDA(p_SS0, p_SS1, p_HS0, p_HS1, p_VS0, p_VS1, D)
{
    (k_0, k_1) = setupK(p_HS0, p_HS1)
    (q_0, q_1) = setupQ(p_VS0, p_S1, k_0, k_1)

    //Permute ray here.

    (dP_SS, dQ, dk) = setupDifferentials(p_SS0, p_SS1, q_0, q_1, k_0, k_1)

    (p_SS, q, k) = (p_SS0 + dP_SS, q_0 + dq, k_0 + dk)
    r_ZMax = p_VS0.z
    while (notAtEndOfRay && !outOfBounds)
    {
        pixel = (permute) ? p_SS.yx : p_SS
        r_ZMin = r_ZMax

        if (intersectionScheme(r_ZMin, r_ZMax, pixel, D))
            return p_SS

        p_SS += dP_SS
        q.z += dQ.z
        k += dk
    }
    return NO_INTERSECTION
}
```

Figure 2-14: Non-Conservative DDA algorithm. Full code sample in Appendix C. (Adapted from [8])

be more expensive to perform on a GPU because it requires a conditional branch instruction to choose to step along the x direction or the y direction. If every thread in a GPU warp or wavefront does not take the same branch, branch divergence occurs. The GPU must then execute both sides of the branch for all threads in the warp or wavefront since they are executed in lockstep on a GPU Streaming Multiprocessor core. A fully C-DDA approach can make use of the Amanatides and Woo algorithm [22].

The Amanatides and Woo C-DDA algorithm [22] keeps track of the ray distance t_{Max} to the next pixel boundaries along both the x and y axis. The next point along the ray is defined by $t_{Current}$ and is chosen by finding which component of t_{Max} has the minimum parametric distance value. A Δt equal to a component movement of one is added to the maximum parametric distance t_{Max} for the given component. Special care needs to be

```
float2 conservativeDDATracing(r_SS, r_Z, D)
{
    (t_Max, Δt, stepSign, t_Current ) = setupForRay(r_SS)
    r_ZMax = r_Z(0)
    while (t_Current  < 1.0f && !outOfBounds)
    {
        r_ZMin = r_ZMax
        if (t_Max.x < t_Max.y)
            t_Current  = t_Max.x
            t_Max.x += Δt.x
            p_SS.x += stepSign.x
        else
            t_Current = t_Max.y
            t_Max.y += Δt.y
            p_SS.y += stepSign.y

        r_ZMax = getRayDepth(r_SS, r_Z, t_Current )
        if (intersectionScheme(r_ZMin, r_ZMax, p_SS, D))
            return p_SS
    }
    return NO_INTERSECTION
}
```

Figure 2-15: Conservative DDA traversal scheme algorithm. Full code sample in Appendix D. (Adapted to use SSR with the algorithm from [22])

taken for horizontal and vertical rays. The C-DDA traversal scheme algorithm is shown in Figure 2-15.

For all of these linear traversal schemes, long screen space rays are a problem. A long screen space ray is a ray that travels a large distance across the screen. Long screen space rays require that the depth buffer be sampled multiple times to traverse its path across the screen. For example, if a screen space ray travels across the entire viewport, many samples are required. One approach to this problem is to increase the stride at which pixels are sampled [3] [8].

The *stride* controls how much screen space distance is traversed before sampling another pixel. An example with a stride of two is shown in Figure 2-16. Increasing the stride results in fewer depth buffer samples, but it can also cause the traversal to skip an intersection with a surface in between the strided sample points. Increasing the stride can

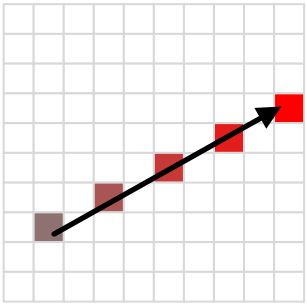


Figure 2-16: A non-conservative DDA with a stride of 2. (original in colour)

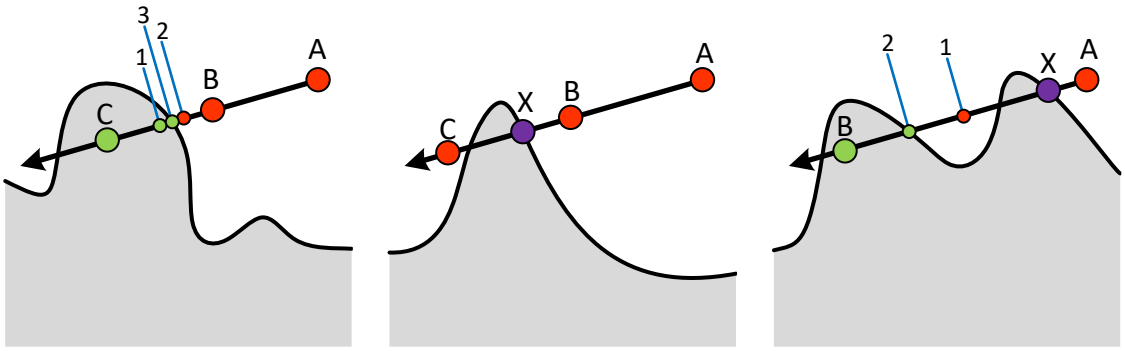


Figure 2-17: (a) Binary Search Refinement. Red dots are misses; green dots are intersections. Numbered blue lines show binary search steps. (b) Large stride causes missed intersection. Purple shows the missed intersection point. (c) Incorrect intersection point. The correct point is shown in purple. (original in colour)

also estimate the depth of the intersection inaccurately. This is especially of concern for long strides. High stride values can also cause banding artifacts in the reflection to occur.

After finding an intersection using strided sampling, binary search refinement [8] can be used to determine a better estimate of the accurate intersection. When a depth buffer intersection is found, the halfway point between the intersection point and the previous sample point can be sampled. A new halfway point can be sampled between the current min and max sample points until an intersection transition is found. This allows a strided sample point that is inside of an object to slide to the edge of the object. This process is shown in Figure 2-17a. Suppose the consecutive sample points A, B and C are generated, where C intersects the depth buffer. Point 1 is created halfway between B and C but this still intersects the depth buffer, so point 2 is generated halfway between B and 1. Point 2

does not intersect the depth buffer, so point 3, between points 2 and 1 is generated. In this case point 3 is found to be on the depth buffer surface.

However, binary search refinement can still fail to find the correct intersection. If the stride is large, consecutive sample points can occur on two sides of an object. Since no intersection is detected, no binary search is performed between the consecutive points and the intersection is missed. This is shown in Figure 2-17b where sample points B and C fail to find the accurate intersection point X. Binary search refinement can also produce an incorrect intersection point when there are multiple surfaces features within the stride. If the binary search looks in the wrong direction, another surface may be missed. This case is shown as shown in Figure 2-17c. The halfway point 1 between A and B does not intersect the depth buffer so the search continues between 1 and B, missing the actual intersection point X. Another technique to solve the problem of long screen space rays is to use a hierarchical depth buffer, which is covered in Section 2.4.2.

2.4.2 Hierarchical Schemes

A *hierarchical traversal scheme* is a traversal scheme that constructs a hierarchical data structure in order to skip large amounts of empty space by traversing the space of the ray in a hierarchical manner. Hierarchical traversal schemes greatly reduce the number of depth buffer samples required to reach an intersection. Thus, hierarchical traversal schemes suffer much less from long screen space rays.

A Hierarchical Depth Buffer or Hi-Z Buffer [23] [24] is the standard hierarchical approach used with SSR. A *Hi-Z Buffer* forms a quad tree in screen space where each level of the hierarchy stores the minimum of the four depth values of the level below it. The size

(a) (b)

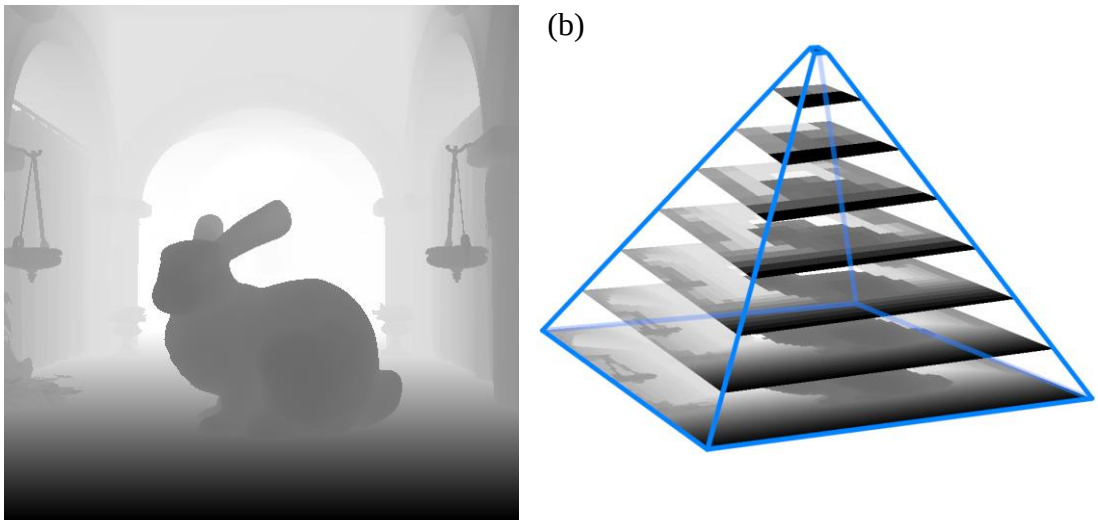


Figure 2-18: (a) Depth buffer for mip-level 0. (b) Minimum Hi-Z Pyramid.

of each level is half that of the previous level. This forms a hierarchical pyramid of depth values (shown in Figure 2-18).

Using a Hi-Z Buffer for hierarchical traversal requires a preprocessing step to construct the Hi-Z Buffer from a standard depth buffer, but the savings that occur during ray traversal can outweigh the precomputation cost. The contents of the pyramid can conveniently be represented in GPU memory as a mip-mapped texture where each mip-level stores a level in the pyramid. The mip-mapped texture allows any texel in any mip-level to be easily queried. The minimum depth values for the succeeding levels of the pyramid must be calculated manually with multiple shader passes that compute the minimum of four texels in the previous level.

Using a Hi-Z Buffer, a screen space ray can be traversed at a fine-grained resolution corresponding to mip level 0 and increasingly coarser-grained resolutions at higher mip-levels. The texels in more coarse-grained levels cover a larger area in the viewport than the texels in the finer grained ones. If the maximal depth of the ray in a coarse-grained texel is less than the depth value in that texel, then the ray does not intersect with the depth value

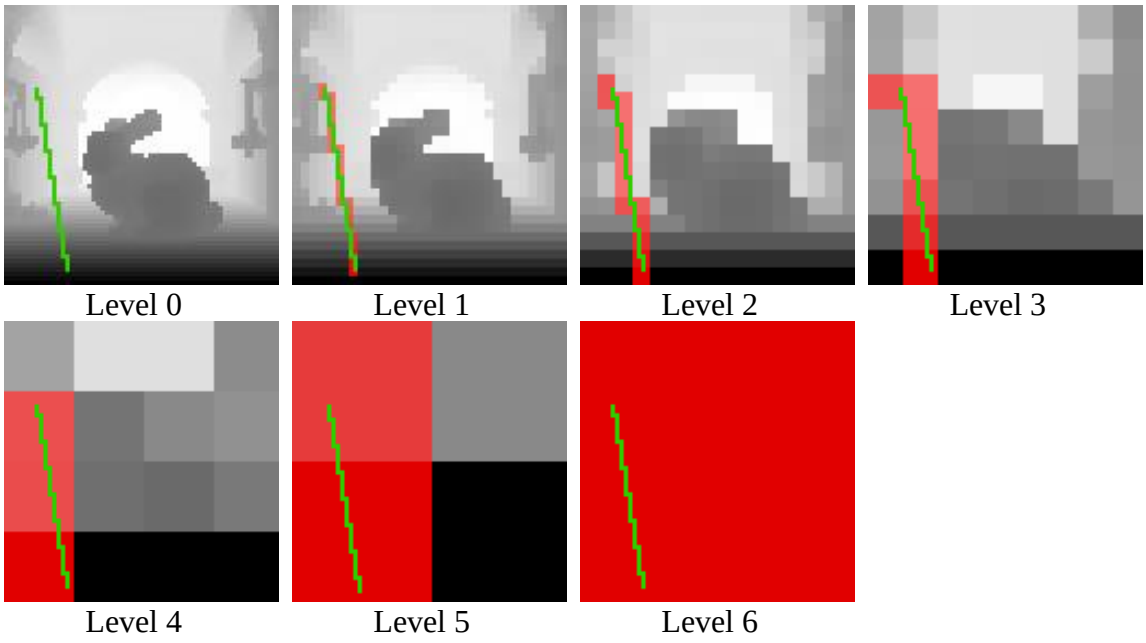


Figure 2-19: Hierarchical depth buffer traversal. The green line is the reflection ray at level 0. Red shows the ray at each hierarchical level. (original in colour)

in any of the finer grained resolutions in the texel area. This property allows screen space ray traversal to operate at higher or lower levels in the hierarchical pyramid levels to traverse at path of the ray at different resolutions. This allows rays to skip large portions of empty space to lower the number of depth buffer samples required to get a potential intersection point. The Hi-Z traversal process is shown in Figure 2-19, where the green lines represents the pixels along the ray at mip level 0 and the red pixels represent the same ray at a different resolution.

Uludag [23] presents a Hi-Z algorithm where at each iteration, a finer grained resolution is examined if the ray intersects with the Hi-Z depth value and a coarser-grained resolution is entered otherwise. During traversal, the ray is clipped to either the planes defined by the texel boundaries or the minimum depth plane stored in the Hi-Z texels. A hit occurs if an attempt is made to decrease the mip level while already at the finest resolution. The Hi-Z traversal scheme is inherently conservative, because the ray traversal

never skips too far ahead along the ray. The conservative property is important, because otherwise intersections could be missed when stepping through higher mip levels.

A Hi-Z Buffer that uses minimum depth values (Min Hi-Z Buffer) works quite well for forward facing rays, but there is difficulty for rays that face towards the camera. To allow for both forward-facing and camera-facing rays, a minimum and maximum Hi-Z Buffer (Min-Max Hi-Z Buffer) can be used [23]. A Min-Max Hi-Z Buffer stores both the minimum and maximum depth values of the 4 texels in the previous hierarchy level. To calculate an intersection for a camera facing ray, the min and max depth values of the Hi-Z interval can be swapped. However, for many camera-facing rays, an intersection may not be found due to rays going outside of the view volume towards the camera. A Min-Max Hi-Z Buffer also allows rays to efficiently pass behind an object.

Pseudocode for a Min Hi-Z traversal based on the Uludag [23] method is given in Figure 2-20. The pseudocode uses a constant thickness intersection scheme. The Min-Max version is similar, but with the difference that both a minimum and maximum depth plane are read from the Hi-Z buffer. The depth plane comparisons can be modified to the code in Figure 2-21. The constant thickness parameter is added to the minimum depth plane and stored in the y component of the Hi-Z Buffer in mip level 0.

Widmer et al. [24] propose a Min-Max Hi-Z algorithm with a Min-Max interval, but they can also store a plane normal in their hierarchical structure if the 2 by 2 nodes can be approximated by a plane. The addition of plane normal vectors in the hierarchical structure allows for skipping additional space by skipping to the plane intersection point instead of the depth interval planes. It also allows for a more accurate intersection point to be obtained versus a simple Min-Max interval.

```
float2 hiZTracing(p_SS0, v_SS, d_Z, D)
{
    mipLevel = 0
    while (mipLevel > 0)
    {
        p_SS = p_SS0 + v_SS × t

        sceneZMin = getSceneDepthPlane(D, p_SS)
        tSceneZMin = (sceneZMin – p_SS0) / d_Z//Scene depth plane

        tPixelXY = intersectPixelEdges(pixel, levelSize, r_SS, r_Z)
        tPixelEdge = min(tPixelXY.x, tPixelXY.y)//Pixel edge depth plane

       mipLevel-- //Descend Level.
       if (intersectsSceneDepthPlane(tSceneZMin, tPixelEdge))
            tParameter = max(tParameter, tSceneZMin)//Set ray to depth plane
            if (atLowestMipLevel(mipLevel))
                if (withinThickness(tParameter, sceneZMin, CONSTANT_THICKNESS)
                    mipLevel = min(MAX_MIP_LEVEL, mipLevel + 2)//Ascend Level
                    tParameter = tPixelEdge//Set ray to pixel edge
        else
            tParameter = tPixelEdge//Set ray to pixel edge
            mipLevel = min(MAX_MIP_LEVEL, mipLevel + 2)//Ascend Level
    }

    if (intersected)
        return p_SS
    else
        return NO_INTERSECTION
}
```

Figure 2-20: Min Hi-Z traversal scheme pseudocode with constant thickness intersection scheme. Full code sample in Appendix E and Appendix E. (Adapted from [23])

```
//...
mipLevel-- //Descend Level
if (tSceneZMinMax.x <= tPixelEdge && tParameter <= tSceneZMinMax.y)
    tParameter = max(tParameter, tSceneZMin)//Set ray to depth plane
    //Descend Level. Happens above.
else
    tParameter = tPixelEdge//Set ray to pixel edge
    mipLevel = min(MAX_MIP_LEVEL, mipLevel + 2)//Ascend Level
//...
```

Figure 2-21: Min Hi-Z traversal scheme depth plane comparison. Full code sample in Appendix G. (Adapted from [23])

2.4.3 Traversal Optimizations

The ray traversal process requires multiple traversal steps in order to find an intersection point. Optimizations that either limit the number of traversal steps or that reduce the number of rays that need to be traversed, should improve the traversal time.

Simple techniques to reduce the number of traversal steps are to terminate traversal after traversing a maximum number of iterations or a maximum ray distance [8]. Adding a stride to the traversal scheme is also an optimization as mentioned in Section 2.4.1. A common technique to reduce the number of rays to traverse is to perform the traversal at a lower resolution such as half resolution. This is employed by several video game SSR implementations [23] [25]. This requires the depth buffer to be down-sampled and the traversal results to be up-sampled. Tracing at a lower resolution reduces the quality of the results but speeds up the traversal. Interleaved sampling can be used to only trace half of the rays during each frame by only tracing every second ray and alternating which rays are traced in a frame [23] [25].

Some optimization techniques aim to avoid traversing rays all. Such techniques can avoid sampling the depth buffer multiple times. Cichocki [26] proposes a simplified SSR technique that reverses the problem by directly taking visible surface points and determining the corresponding points that they reflected off. This works by reflecting the visible surface points about some defined planar surface and testing if the ray from the camera to the reflected point intersects the surface. For valid intersections, offsets are scattered into a buffer so that the reflection can be directly determined in a separate reflection pass. Drobot [27] avoids traversing rays with SSR by reusing the intersections of box projected reflection probes to generate approximate reflections.

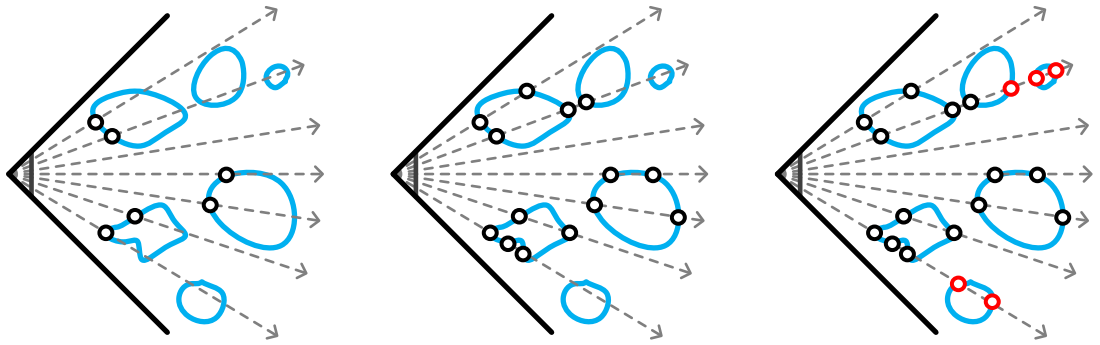
Another optimization is to handle texture samples in a group called a batch instead of individually [28]. With this approach, multiple depth buffer samples along the ray can be read at once. Linear traversal schemes can easily take advantage of batched samples since it is easy to determine each subsequent sample point along the ray. However, choosing a

batch of samples is more difficult with hierarchical traversal schemes because of the varying mip levels that need to be sampled.

2.5 Multi-Layer Techniques

Occlusion within the camera view is a major source of artifacts for SSR. Since the depth buffer only stores the Z distances to visible surfaces, there is no information about the scene behind these visible surfaces. If a ray travels behind an object and becomes occluded from the camera view, it is unknown which surface it intersects. The ray traversal algorithm could be designed to stop when it encounters an occlusion, but this would leave a hole in the reflection corresponding to the occluded rays. SSR could also be designed to allow a ray to pass behind objects and potentially find an intersection point after the ray becomes unoccluded again. This last approach can work in many cases, but it is technically incorrect. It does not account for a ray hitting an occluded surface that is behind the surface that is occluding the ray.

A solution to the occlusion problem is to use multi-layer SSR. *Multi-layer SSR* stores multiple layers of scene information in each pixel and performs traversal across multiple layers. With multiple layers, the visible surfaces as well as subsequent surfaces are captured. Figure 2-22 compares the standard Z-Buffer approach (Figure 2-22a) to multi-layer depth buffer approaches. Figure 2-22b shows a K-Buffer approach. A K-Buffer stores at most k layers for each pixel where each layer stores a pixel fragment. A *fragment* is a sample point of rasterized geometry, which includes data, such as depth, that is required to render a pixel for a surface along a primary view ray. There can be multiple fragments along the primary ray and the closest fragment defines the visible surface in the pixel. A K-Buffer captures more scene information than the Z-Buffer shown in Figure 2-22a. In



(a) Z-Buffer (b) K-Buffer (c) A-Buffer

Figure 2-22: Depth Buffering Techniques. Dots represent the stored samples along each view ray. (a) Z Buffer stores information about only visible surfaces to be captured. (b) K-Buffer stores at most K fragments (K = 3 shown). (c) A-Buffer stores all fragments along a view ray. Fragments not captured by K-Buffer are shown in red. Both (b) and (c) include back faces. (original in colour)

fact, the Z-Buffer approach is equivalent to the K-Buffer approach with $k = 1$. To capture of the all fragments in each pixel, an A-Buffer (shown in Figure 2-22c) must be used. However, generating these multiple layers is quite expensive to do, especially in real-time applications such as games.

If deferred shading is used with a multi-layer technique, the G-Buffer properties must be stored for the fragments in every layer. These properties include the depth, normal, and material properties. The resulting multi-layer structure is called a *Deep G-Buffer*. Figure 2-23 shows the surface normal vectors stored in a K-Buffer. The other G-Buffer properties still need to be stored in the K-Buffer in order to both determine shading and to perform the SSR ray traversal process. Mara et al. [29] note that most of the scene information is available in the first two layers and they provide an efficient 2-layer Deep G-Buffer algorithm with a minimum constraint on the separation between fragments. This constraint ensures that fragments in different layers are at least a fixed distance away. The advantage of using the minimum separation constraint is that it can avoid storing information about surfaces that are close behind the visible surface and thus unlikely to be hit by a ray. Nonetheless, using this constraint can also remove important scene information.

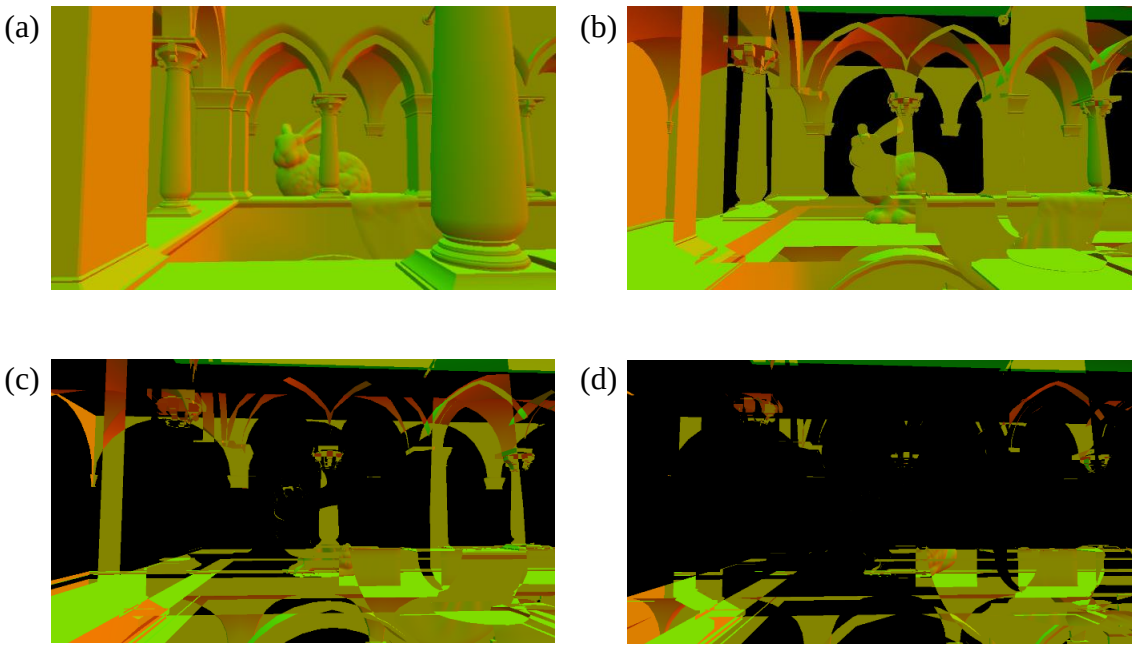
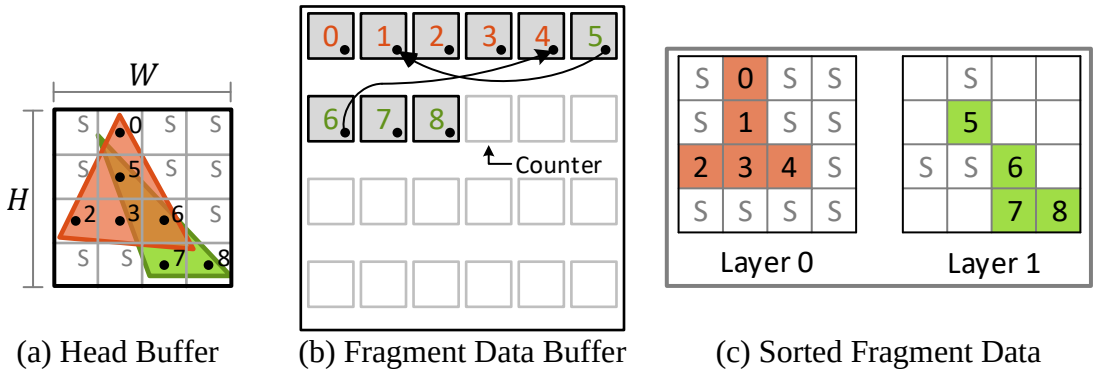


Figure 2-23: Encoded surface normal vectors for K-Buffer approach (K = 4) for layers 0, 1, 2 and 3 (a, b, c and d, respectively). Black indicates that that there are no fragments at that layer. The normals are encoded with Lambert azimuthal equal-area projection [11]. (original in colour)

The K-Buffer approach is only concerned about the *k* nearest fragments in each pixel. Thus, it can have a fixed memory cost by allocating enough memory for each pixel to have *k* fragments. A K-Buffer can be generated by depth peeling [30]. *Depth peeling* is a technique to generate multiple layers by using two depth buffers that act as lower and upper bounds on which pixel fragments are accepted for rendering in the current layer. Fragment depth must be greater than the lower bound and less than the upper bound. Each pass clears the upper bound depth buffer and uses the upper bound buffer of the previous pass as its lower bound buffer. The first pass does not need a lower bound buffer. Each depth peeling pass *i* produces the *i'th* close fragments for each pixel. This involves *k* passes over the scene geometry, which can be prohibitive for some applications.

Another approach to create a K-Buffer is to generate a linked list for each pixel that stores all fragments relevant to that pixel and apply a sorting pass to the linked list to select the closest *k* fragments in the linked list [31]. Two GPU buffers are required for this



(a) Head Buffer (b) Fragment Data Buffer (c) Sorted Fragment Data

Figure 2-24: K-Buffer generation with linked list and sorting pass. The approach from [31] but with a separate sorted fragment buffer. S represents the end of list sentinel value. The red triangle is drawn first (fragments 0-4) and is in front of the green triangle (fragments 5-8). (original in colour)

approach: a head pointer buffer (shown in Figure 2-24a) and a fragment data buffer (shown in Figure 2-24b). The fragment data buffer is large block of memory that stores the G-Buffer properties of all pixel fragments generated by the graphics pipeline. This requires allocating enough memory to store all the fragments in the fragment data buffer. This size is neither fixed nor known in advance, so a large amount of memory must be allocated to fit all fragments. Each fragment also stores an offset or "pointer" to the next fragment in the list. As each fragment is generated, it is added to the fragment data buffer using an atomic counter and the head buffer is set to point to the newly generated fragment. The linked lists can then be sorted by depth using a simple sorting algorithm such as insertion or selection sort. The sorted fragments can be sorted either in the unused portions of the fragment data buffer or stored in a separate fixed size buffer with k levels with a sentinel to denote the end of the list (shown in Figure 2-24c). If the fragments are stored in a structured order in the sorted list, the need for a sorted head buffer is removed and the "pointer" component of fragment data can be omitted in the sorted list. Hofmann et al. [32] perform truncated sorting using shared memory to increase efficiency.

K-Buffers can also be generated by using pixel synchronization [33], which can be implemented in Direct3D with Rasterizer Ordered Views [34]. Pixel synchronization and

Rasterizer Ordered Views process geometry in the order that it was submitted. The use of this ordering removes the need for atomic instructions, which are required by the linked list approach. The K-Buffer can be generated in a single geometry pass where a pixel shader generates the current G-Buffer fragment, loads the previous fragment, sorts the fragments, and writes the sorted fragments for each pixel. The need for fragment memory buffers is also eliminated since the fragments are inserted in sorted order.

An A-Buffer is a special case of a K-Buffer where $k = \infty$. Using an A-Buffer assumes that enough memory is available to store all the fragments. An A-Buffer can be created by using the same linked list approach mentioned above, but the list must be fully sorted for each pixel. The linked list approach makes it more difficult to store the sorted fragments in an ordered format since there is no bound on the number of fragments relevant to a pixel. Vardis et al. present two multi-layer SSR techniques that use A-Buffers, one that stores pixel fragments [35] and another that stores per-pixel linked lists of primitive IDs to compute the ray triangle intersection analytically [20].

If multi-layer SSR is used, the traversal scheme must be modified since there is no longer a single layer of fragments per pixel. When testing a ray against a depth buffer texel, the ray may need to be compared with the depth value of all layers for that texel. A loop over each layer is performed during each traversal step. At each loop iteration, the depth interval of the ray is compared to the depth interval stored at the current layer. This process is repeated until an intersection is found or the layers are exhausted.

The loop can exit early if the maximal ray depth is less than the minimal depth buffer value at a layer. In this case, which is shown in Figure 2-25, is it impossible for the ray to intersect any deeper layers in the pixel. However, checking this condition does add branch

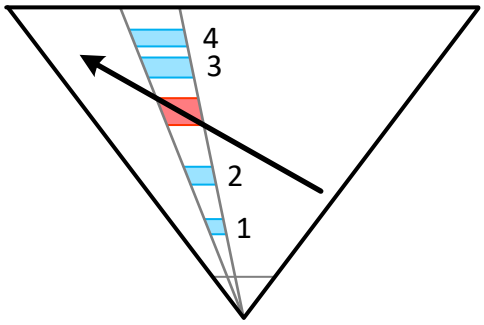


Figure 2-25: Multi-layer SSR traversal step. The depth interval of the ray (red) in the pixel is compared with the depth interval of each layer (blue). Note that the last depth layer does not need to be checked in this case (layer 4). (original in colour)

divergence to the traversal process because the number of layers that need to be checked for each texel can differ between threads in a GPU warp or wavefront. McGuire and Mara load four layers at once by storing each layer in the RGBA channels of a texture [8].

Multiple layers can be incorporated in any of the traversal schemes discussed in Section 2.4 once the multi-layer data structure has been generated. DDA based traversal schemes simply require a loop over each layer. Hierarchical traversal schemes must also loop over each layer, but additional techniques can be performed to improve efficiency. Widmer et al. [24] construct a quadtree where each node is either a depth interval or a plane normal. They use a K-Buffer to store information about k layers and they generate a separate quadtree structure for each layer. The quadtrees of deeper layers are only traversed when the occlusion volume of a layer is intersected.

Hofmann et al. [32] present a different hierarchical multi-layer approach that merges the fragments in higher level Hi-Z layers together based on a varying distance from the first fragment in the interval. They capture the Min-Max interval of each fragment for each layer in the first mip level (shown in red in Figure 2-26a). Each successive Hi-Z mip level stores a set of fragment depth intervals for each texel. A Hi-Z mip level texel stores an interval for all fragments that are within a certain distance T of the closest fragment, as

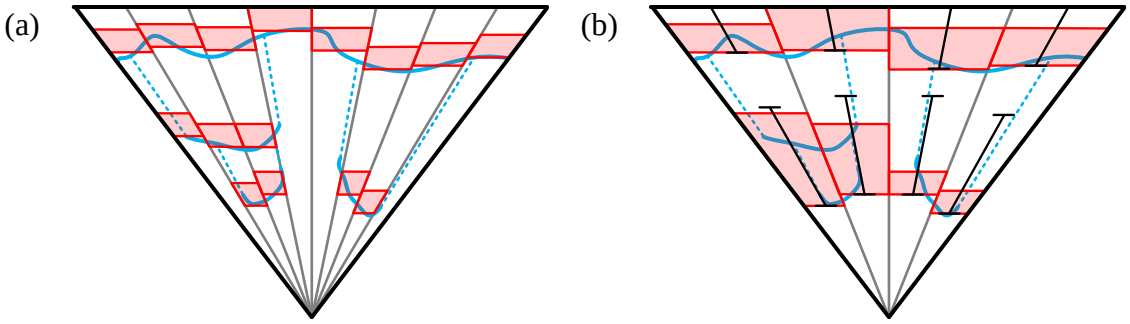
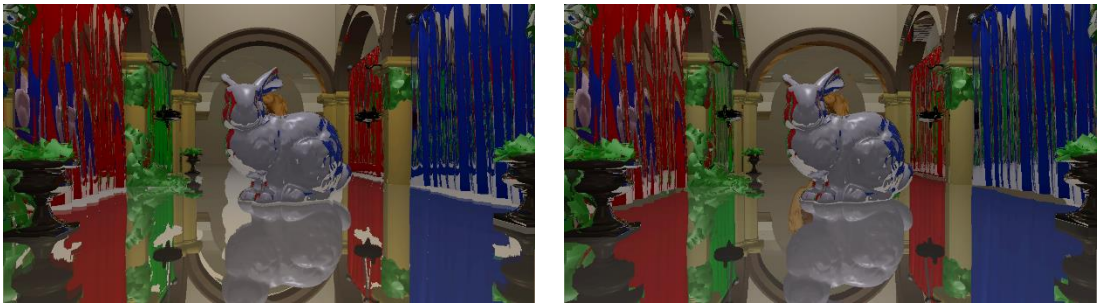


Figure 2-26: Multi-layer Min-Max Hi-Z structure [32]. The red blocks represent the depth intervals of fragments. (a) Mip level 0. (b) Mip Level 1. The black bars represent the threshold T. (original in colour)



1 Layer 4 Layer

Figure 2-27: Comparison between single and multi-layer SSR.
Note the removal of the holes behind the bunny. (original in colour)

well as subsequent intervals within distance T for the remaining fragments (mip level 1 shown in Figure 2-26b). For a texel at a given mip level, fragments from the corresponding four texels of the previous mip level are placed into fragment intervals until all fragments have been considered. Using fragment intervals allows multiple nearby fragments to be skipped at once during the Hi-Z traversal. After performing multi-layer ray traversal, the resulting image should have fewer holes compared to single layer traversal as seen in Figure 2-27.

Hofmann et al. [32] define the T parameter to determine the size of the fragment interval as $T = \tau \times w \times 2^{M-1}$, where w is the width of the current closest fragment of the current layer L, τ is a user controllable parameter, and M is the current Hi-Z mip level. To generate the fragment intervals for a mip level M, each layer of fragment intervals in the

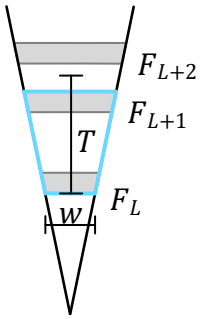


Figure 2-28: Multi-layer Hi-Z threshold [32]. Fragments F_L and F_{L+1} are included in the blue interval, but fragment F_{L+2} is not. F_{L+2} is the next closest fragment for the next layer at mip level M.(original in colour)

(a) (b)

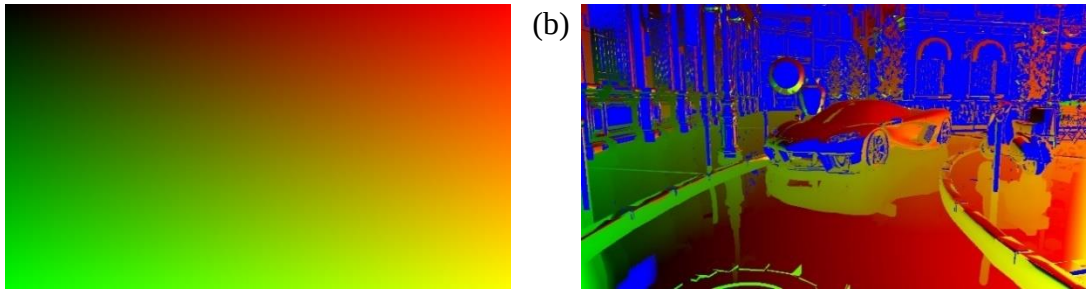


Figure 2-29: (a) Screen Space Coordinate Visualization. The red and green channels show the u and v coordinate, respectively. (b) Screen Space hit point coordinate. Blue represents ray misses. (original in colour)

previous mip level $M - 1$ is searched through and grouped into fragment intervals for mip level M. Figure 2-28 shows the first iteration of this process.

2.6 Shading

The output of the traversal process is a screen space hit point. In other words, for a given pixel, the traversal process determines the screen location where the reflection ray intersected the depth buffer. A hit point is simply a screen space coordinate. Figure 2-29a shows a visualization of the screen space coordinate for each pixel on the screen, typically denoted (u, v). After performing the traversal process using a traversal scheme, each ray that intersects with the depth buffer has a hit point. Figure 2-29b shows the hit points generated by the traversal process. Rays that missed are shown in blue. In this example,

rays that face towards the camera are rejected, so reflection rays from the walls in the back all miss.

Once the traversal process has determined the hit point for a ray, the shading for the reflection value can be determined. Since deferred shading stores the shading properties for each pixel on the screen, the screen space hit point can be used to read from the G-Buffer to calculate the shading at this point. However, several hit points could map to the same pixel which would result in calculating the exact same colour value for hit points in the pixel. Instead, the colour value from the lighting buffer could be used, since it must already be calculated for the visible surfaces. This avoids the cost of potentially evaluating the shading multiple times, but it is also technically incorrect since the colour value is evaluated along the primary ray direction [36]. This means that anisotropic materials cannot be accurately rendered.

Many SSR implementations instead choose to reproject the reflection hit points into the previous frame to obtain a reflection value from the previous frame [3]. Reprojection causes the reflections to lag, but temporal filtering is applied to achieve a stable result. Reprojection into the previous frame allows multiple bounces to be accumulated without having to traverse all the bounces [23].

Tracing a single reflection ray produces sharp specular reflections, but there may be artifacts in the image. However, most surfaces in the real world are not perfect mirrors and instead have some roughness. A rough surface produces a blurry or glossy reflection. Often, SSR implementations take a filtering approach to rendering glossy reflections.

Valient [25] creates mip chains with successively blurred versions of the reflection buffer and hit mask. The mip chains are sampled based on the material roughness to obtain

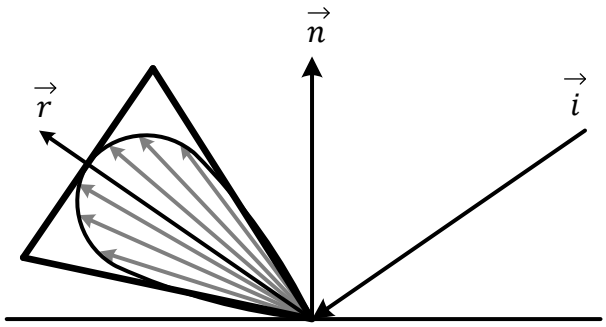


Figure 2-30: A rough surface reflects incident ray $\vec{\imath}$ about the normal $\vec{n}$ light in specular cosine lobe in direction $\vec{r}$. The lobe can be approximated by a cone (Cone shown in 2D). (Adapted from [1])

a blurred reflection colour and a blurred mask value to weight the reflection. This approach is not physically accurate because it is not depth aware. The lack of depth awareness results in reflections leaking into pixels that should not be affected and the method fails to capture contact hardening in the reflections.

Uludag [23] presents a similar approach inspired by voxel cone tracing, which allows reflections of distant objects to be blurrier. Uludag generates a convolved mip chain of the colour buffer as well as a visibility buffer mip chain. The mip chains are sampled at points along the at points along the axis of the cone of influence of the ray. This cone is shown in Figure 2-30. The mip level to sample is based on the radius of cone at a point along its axis. In screen space, the cone can be approximated as an isosceles triangle. This means that the radius of an inscribed circle in the triangle, which is centered along the axis of the cone is the screen space equivalent to the cone radius. The radius value is used to determine which to points along the cone axis to sample the convolved colour and visibility buffer mip chains. Tri-linear interpolation is used for the sampling. Hermanns and Franke present a similar cone tracing approach [37].

Instead of using simple filtering, Stachowiak and Uludag [4] take an approach that uses Monte Carlo integration. They generate random rays with importance sampling and

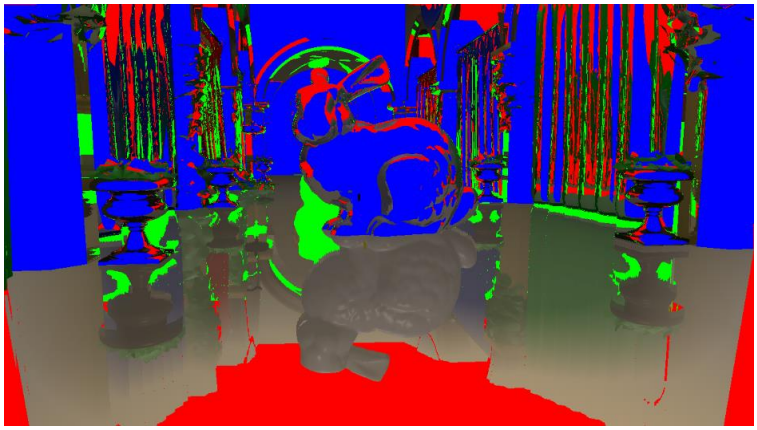


Figure 2-31: Holes in the reflection. Red represents areas where the ray went out of bounds of the image. Green represents areas where the ray misses due to occlusion or reaching a maximum number of iterations. Blue represents areas where the ray reflects towards the camera and is ignored in this example. (original in colour)

accumulate the contributions with neighbouring pixels by properly weighting them. Thus, they assume that the hit points of neighbouring pixels are visible from the reflection surface point, which is not necessarily true.

2.7 Artifact Resolution

Due to the incomplete representation of the scene available to SSR, SSR suffers from several visual artifacts. Single layer SSR approaches, which are typically used, have access to only the visible surfaces in the scene. Multi-layer approaches can capture additional layers of scene information, but the information is still incomplete. There is only information concerning the field of view of the camera. It is impossible to determine hit points outside of this view. The lack of information results in artifacts called *holes* in the reflections produced by SSR. Holes can also occur when a ray passes into the occlusion volume behind a surface. Some of these hole artifacts are visible in Figure 2-31. Flickering can also occur as a reflection hit point suddenly becomes visible or occluded due to movements of the camera or objects in the scene.

Uludag [23] describes several techniques to hide several of the artifacts associated with SSR. Uludag fades out the effect of ray hit points that are near the edges of the screen, rays that have traveled a long distance, and rays that point towards the camera. These techniques are not perfect, but they are often preferable to having holes or jagged edges in the ray traced portions of the image. Despite using these simple techniques, a miss can still occur.

Another common approach is to have a fallback reflection technique for ray misses. On a ray miss, a local cube map or reflection probe can be used as a fallback technique. A *cube map* is a representation of the scene from a point that encodes a colour value for each direction mapped onto a cube. There are six faces on the cube map and a reflection direction vector can be mapped onto a point on one of the 6 faces to approximate a reflection. If there is no suitable local cube map usable near an SSR hit point, a global cube map can be used instead [8] [38]. Cupisz and Kasper [39] use a confidence value based on the number of traversals steps required to get a ray intersection, such that the confidence value is lower if many traversal steps are required. The confidence value is used to linearly interpolate between the value produced by an SSR technique and the value produced by the fallback technique.

Depth precision can also be a source of artifacts. With a standard hyperbolic depth buffer, most of the range of depth values is reserved for values close to the near plane. Swapping the near and far plane in the projection matrix can be used to produce a reversed depth buffer, which can offer more precision [23].

2.8 Multi-View Approaches

Typically, SSR has information about only the visible surfaces of the camera view. Multi-layer approaches go a step further and provide information about the scene for

occluded areas, but the resulting representation of the scene is still incomplete. There are approaches which provide multiple views of the scene. In a *multi-view SSR* approach, multiple views of the scene are generated, and the ray traversal process is performed by switching between the multiple views. This approach provides information about the scene that is outside of the camera view, although it still does not create a complete representation of the scene unless a view of every surface is created. Using a cube map for a fallback technique, as discussed in Section 2.7, can be thought of as a multi-view approach, but here, the discussion is limited to techniques that perform some ray traversal process.

A multi-view approach generates the same data structures required for the main camera, such as the G-Buffer, but from several different viewpoints. To perform ray traversal against these data structures, the traversal process operates in the image space of the viewpoint instead of the screen space of the camera view. Either of these two spaces can be used equally well by SSR traversal schemes. Multi-view approaches are computationally expensive since they require the scene to be rendered multiple times to generate the different views. One way to decrease the expense is to generate these views at a lower resolution, which reduces the number of pixels to be rendered and traversed.

An early multi-view approach based on image space ray traversal across a depth or distance buffer was proposed by Szirmay-Kalos et al. [9] and Umenhoffer et al. [10]. They create cube maps at the center of reflective or refractive objects and performed two traversal schemes, one that operates linearly and one that uses a secant search. This method allows for objects to be reflective or refractive and it is an improvement over standard cube mapping. This is because standard cube mapping or environment mapping assumes that surfaces are infinitely distant. The method performs a similar traversal process to modern

SSR, except that the traversal is performed by using the cube map distance buffers instead of the camera depth buffer.

Instead of placing cube maps at the center of objects, several multi-view methods place cube maps at the camera location. The traversal of a ray across a cube map is performed by traversing the path of the ray in the image space of the cube map face that it is in. When the ray traversal process reaches the edge of a face, the search can continue in the next adjacent face. By using a camera cube map, scenarios such as looking straight at a mirror become possible. Such a scenario is impossible for single view SSR to resolve since there is no information on what is behind the camera.

Several efforts have been made to apply the camera centered cube map approach. Ganestam and Doggett [18] propose a hybrid approach that uses both a spatial acceleration structure in combination with a camera centered cube map. They use a *Bounding Volume Hierarchy* (BVH), which is a spatial acceleration structure that defines a hierarchy of volumes encompassing the volumes of child nodes. The BVH is centered on the camera and is used for nearby objects, while the camera centered cube map is used for distant objects. Ray traversal operates by switching between traversing the BVH and ray marching against the camera cube map. Widmer et al. [24] also traverse rays across a camera cube map, but they use a hierarchical traversal scheme that accounts for surface normals. Vardis et al. [35] create an A-Buffer data structure for each of the six faces of the camera cube map as well as for the camera view itself. A hierarchical traversal is performed against the A-Buffer layers in the main view and the cube map views. Vardis et al. [20] also provide a variant of their algorithm that stores triangle IDs instead of fragments. For each fragment they store a triangle ID and compute the depth interval that the triangle occupies within the

texel. The interval is calculated by clipping the triangle against the skewed frustum of the acceleration structure texels. The triangles are further organized into buckets. During the traversal process, ray triangle intersection is only performed if the ray intersects the depth interval.

Some approaches place cube maps in the empty space of the scene and rays are traced against the depth maps of the cube maps faces. When a ray goes into the occlusion volume from the perspective of one cube map, another cube map can be selected, and the trace can continue in that cube map. McGuire et al. [40] create a field or 3D grid of light probes. Each light probe captures a representation of the light in the scene at the grid point by storing radiance, normal and depth per pixel. These light probes use an octahedral mapping instead of a cube map. Traversal begins in one of the light probes and continues until an intersection is found or an occlusion volume is found. If an occlusion volume is found, the traversal begins in another light probe. Sobek [41] places cube maps at designer selected points, such as the center of rooms or hallways. Sobek first perform an SSR technique to get either a ray hit point or a point when the ray becomes occluded. If the ray did not intersect anything, the traversal can continue by ray marching inside one of the cube maps. If the ray becomes occluded in this cube map, then a new cube map is selected, and traversal continues in it.

Chapter 3 Evaluation

An evaluation was performed to evaluate the performance of several SSR techniques. Several traversal scheme algorithms were implemented in a custom engine using Direct3D 11. Section 3.1 outlines the measurements gathered in the experiment such as the rendering time. User acceptance for different SSR techniques is strongly influenced by how fast they can be performed. Therefore, measuring the runtime is highly appropriate. Various factors that affect the performance were varied such as the resolution. The implementation used in the evaluation is described in Section 3.2. Section 3.3 details the test scenes used in the experiment. A statistical hypothesis test is performed to show that the traversal time measurements are statistically significant. The statistical tests are described in Section 3.4, while the results of the evaluation are given in Chapter 4.

3.1 Measurements

The main performance metric that was measured for SSR was the average GPU time to perform the SSR rendering process and the average time to build any acceleration structure such as a Hi-Z Buffer. The average times to perform other rendering tasks such as the geometry pass and the lighting pass were also calculated. The evaluation was performed on a computer with an NVIDIA GTX 1080 Ti for the GPU.

GPU timing measurements were obtained using NVIDIA Nsight Graphics. NVIDIA Nsight Graphics is a GPU profiling and debugging tool for NVIDIA GPUs. Figure 3-1 shows a screen shot of the tool showing a timeline of event ranges with deferred shading and SSR. Performance markers were placed around the relevant sections of code for each task in the rendering process. The performance markers denote the start and end of a range of graphics API events on the GPU. The profiling tool keeps track of the GPU time for

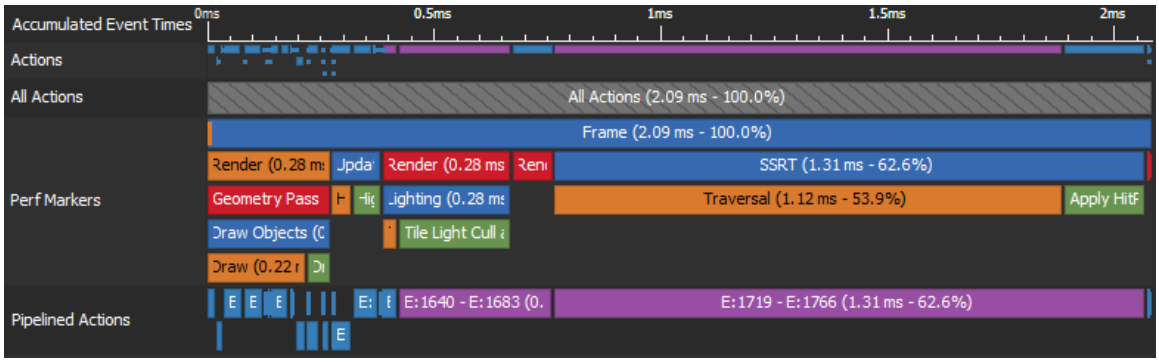


Figure 3-1: Screen shot of NVIDIA Nsight Graphics profiling tool showing the GPU time ranges. (original in colour)

these event ranges. The timings for the event ranges were recorded by capturing different frames using the profiler and generating an average for each event range.

The evaluation examines five traversal schemes: 3D ray marching, a Non-Conservative DDA (NC-DDA), a Conservative DDA (C-DDA), a Min Hi-Z and a Min-Max Hi-Z. The 3D ray march is based on the original SSR method presented by Souza et al. [3]. Though little details were given in their work, their method has been inferred to be a 3D ray march [8]. Ray marching in 3D was chosen for evaluation to provide a comparison of the different traversal schemes with the original SSR method. The DDA algorithms were chosen since they are another type of linear traversal scheme. The NC-DDA is based on the algorithm by McGuire and Mara [8], while the C-DDA uses the Amanatides and Woo algorithm [22]. Since linear traversal schemes perform poorly with long rays, it is natural to evaluate hierarchical schemes. Both Min Hi-Z and Min-Max Hi-Z were evaluated to compare how both traversal schemes perform when the ray becomes occluded. The single-layer Min Hi-Z and Min-Max Hi-Z are based on the Uludag algorithm [23]. However, the traversal schemes used for the Hi-Z algorithms, compares parametric ray distances instead of depth values.

Multiple layers are considered for the linear traversal schemes and the Min-Max Hi-Z traversal scheme. These traversal schemes are evaluated with a different number of layers: single-layer, 2-layer, 4-layer, and 8-layer. For the multi-layer Min-Max Hi-Z traversal scheme, the Hofmann et al. [32] approach is used generate a multi-layer Hi-Z structure. A multi-layer Min Hi-Z scheme was not considered, as it would be inefficient.

The image resolution affects the number of rays that need to be traversed. To test how SSR scales with resolution, the evaluation uses both 1920x1080 and 1280x720 resolutions. The common optimization of performing the traversal at a lower resolution and upscaling the results was not tested in this evaluation.

Two intersection schemes are considered in the evaluation. Constant thickness and infinite thickness intersection schemes are considered. The constant thickness intersection scheme uses a thickness value of 0.2. The thickness value depends on the scale of the scene. An infinite thickness intersection scheme is simulated by simply using a sufficiently large thickness value with respect to the far clipping distance. Performance could possibly improve slightly by having a hardcoded version of the shaders with infinite thickness.

A maximum number of 400 iterations is used for all tests. Each iteration involves a single sample point or pixel. Each ray has a maximum traversal distance of 80. The maximum ray distance parameter depends on the scale of the scene. The distance was selected to cover the extent of the scenes used.

The evaluation also measures the average number of traversal iterations in each pixel for each traversal scheme. A visualization of the number of traversal steps for each traversal scheme tested is generated.

3.2 Implementation

The implementation uses a custom engine built using Direct3D 11. As previously mentioned in Section 2.4, the code for SSR traversal main function and the traversal schemes is given in Appendix A through Appendix G.

3.2.1 Über Shaders

Since there are various traversal schemes and varying parameters that could be applied to the traversal schemes, several different SSR shaders had to be created. These shaders all have the same basic structure which results in duplicated code. To avoid duplicated code and to ensure that each shader performs the same operations for the constant parts of the SSR algorithm, conditional compilation was used. Preprocessor directives were used to include or exclude the nonconstant parts of the SSR algorithm such as a differing traversal scheme or layer count. Multiple permutations of the shader for varying configurations were generated by compiling multiple shaders. This idea is often called an *über shader*, though this term is sometimes used to describe a complex shader which selects differing options at runtime using dynamic branching. A system to select the appropriate shader variants at runtime was built into the engine. The über shader approach was used to avoid code duplication and to allow a fair comparison between traversal schemes by ensuring the constant parts of the shader exactly are the same.

Care has been taken to try and remove parts of the shader code that the compiler could not optimize away by examining the HLSL shader disassembly. However, there are likely more improvements and optimizations that could be applied. The results could be affected by the lack a fully optimized system.

3.2.2 Deferred Shading and Build

Tiled deferred shading, as is described in Section 2.1, is used as the rendering algorithm in the implementation. The geometry pass is performed to generate a G-Buffer. Frustum culling against mesh Axis Aligned Bounding Boxes (AABB) is used to limit the amount of geometry that needs to be submitted to the graphics pipeline. Coarse depth sorting of draw calls is also to used as an optimization. No spatial acceleration structures are used.

After the geometry pass, a *Build* stage is performed to generate any data required for SSR. If the traversal scheme requires a Hi-Z Buffer, the Hi-Z Buffer is built in the *Build* stage. Eight hierarchical levels are used instead of constructing the entire mip chain. When using a single layer, the depth buffer is converted to linear depth in the *Build* stage. The conversion is done here to avoid converting during SSR traversal so that a thickness can be applied to the depth values. The conversion does lower the precision of the depth values. A standard hardware depth buffer is still used during the geometry pass in order to allow the early depth test optimization on the GPU. After the *Build* stage, the lighting pass is performed. The lighting pass uses AABBs for the tile depth bounds to cull lights in a tile.

3.2.3 Multiple Layers

For multi-layer support, the implementation creates a K-Buffer by generating fragment linked lists and performing a truncated sort to k layers using the approach by Yang et al. [31]. No minimum separation between layers is used. A K-Buffer generated in a single pass with Rasterizer Ordered Views was considered, but it was found to be outperformed by the linked list approach.

For the multi-layer hierarchical traversal schemes, mip level 0 of the multi-layer Hi-Z structure is generated for each layer with a shader that reads the compressed fragment from

the fragment buffer, decompresses the depth and converts them from a normalized linear integer depth to floating point view space depth. The view space depth is saved into mip level 0 of the Hi-Z structure as the minimum value. A constant thickness is added to the minimum value to be used as the maximum value.

The higher mip levels are generated in several passes with a separate shader which groups nearby fragments together into fragment intervals. This process requires the τ parameter which was defined in Section 2.5. Hofmann et al. [32] use the τ parameter to control the sizes of fragment intervals in the multi-layer Hi-Z structure. A value of $\tau = 25$ was found to work well for the implementation and scenes used in the evaluation. The multi-layer Hi-Z traversal uses the Uludag algorithm [23], but it is modified to support multiple layers.

In the current implementation, the lighting pass generates a shaded image for each layer when using multiple layers. This is wasteful since not all the shaded fragments are necessarily used when applying the SSR hit points. A better approach might be to only perform shading for pixels which have SSR hit points. Another approach could be to generate only the first layer, since it is required to render the scene, and to sample the G-Buffer for each SSR hit point with a layer index greater than zero. The latter approach may shade the same point twice, but it may have less overhead since most of the SSR hit points reside in the first few layers.

3.2.4 SSR

After performing tiled deferred shading and building the appropriate data structures, the SSR traversal shader is executed. The SSR traversal shader is implemented as a compute shader where each thread is responsible for a single pixel. The compute shader

implementation was found to outperform an equivalent implementation that used a pixel shader with a screen facing triangle. Hofmann et al. [32] terminate rays when the number of threads executing in a warp is low. The implementation in this thesis does not perform this optimization. Multiple versions of the traversal shader are created with an über shader as described in Section 3.2.1.

The traversal shader outputs ray hit points (or misses), the hit layer and a blending amount. The output is stored in an RGBA texture with 16 bits per component. The reflection colour is determined in a separate pass called *Apply Hits.* The *Apply Hits* shader reads the hit points as input, performs artifact resolution, and outputs the reflection colour to be blended over the image using alpha blending. The reflection colour is determined by sampling the colour buffer or lighting buffer at the screen space hit point. The implementation does not account for surface roughness. *Apply Hits* is kept separate from the traversal process to allow for a more accurate timing of the traversal scheme and to allow for any filtering methods to be applied to the output. However, no such method is performed in this implementation.

3.2.5 Assumptions

The evaluation makes a few assumptions about the reflective rays. First, every surface is assumed to be reflective, meaning that a reflection ray is generated for each pixel. The surfaces are also assumed to be specular, meaning that rough surfaces are not accounted for. To account for rough surfaces, one of the approaches in Section 2.6 could be used. Also, only forward-facing rays are traced. This is achieved by rejecting a view space reflection ray when $d_{VS}.z \leq 0$. Rays are permitted to pass behind objects, travel behind

Figure 3-2: Reference view for the Sponza scene. (original in colour)

Figure 3-3: Reference view for the Amazon Lumberyard Bistro Exterior scene. (original in colour)

and potentially find an intersection if the ray becomes unoccluded (disocclusion). This could result in a false intersection.

3.3 Test Scenes

For the evaluation, two test scenes [42] are used: the classic *Sponza* scene shown in Figure 3-2, and the *Amazon Lumberyard Bistro Exterior* scene shown in Figure 3-3. These test scenes contain additional objects placed in the scene [42] [43]. The Sponza scene contains ~350K triangles including the triangles for the additional objects. The Bistro scene is a more complex scene that might be a more realistic representation of a modern video game scene as it contains ~3.1M triangles. The scene triangle count mostly affects the

deferred shading geometry pass since the lighting pass and SSR pass are screen space effects.

The reference view of the Sponza scene contains 17 visible or potentially visible point lights within the camera frustum, while the view of the Bistro scene contains 71. The scenes are untextured and normal mapping is not used due to limitations of the custom 3D engine. Normal maps affect the reflection direction of reflection rays, so the lack of normal mapping in this evaluation is a limitation.

The view chosen for the Sponza scene contains a more constrained space, as if looking down a hallway. The Bistro scene shows a more open space that contains several objects in the space that could lead to occluded rays such as the lamp posts and string lights hung over the street. These two types of views affect how rays reflect and become occluded.

3.4 Statistical Tests

The different test configurations used in the evaluation are described in Section 3.4.1. Hypothesis tests are used to show the statistical significance of the measurements for the average *Traverse* time. The hypothesis tests are described in Section 3.4.2, while the method to remove outliers in the data is described in Section 3.4.3.

3.4.1 Test Configurations

Each traversal scheme in the evaluation is tested on various test configurations. For each test configuration, 30 frames are captured with the profiler to record the GPU event timings for the relevant rendering tasks. Each frame capture has the same camera setup with no movement in the scene, thus each frame essentially performs the same workload. Each test configuration is a combination of each possible assignment of the following variables:

$$Scene \in \{Sponza, Bistro\}$$
$$Resolution \in \{1920x1080, 1280x720\}$$
$$LayerCount \in \{1,2,4,8\}$$
$$IntersectionScheme \in \{0.2, \infty\}$$

For each test configuration, each of the following traversal schemes is tested:

$$TraversalScheme \in \{3DRayMarch, NCDDA, CDDA, MinHiZ, MinMaxHiZ\}$$

There are constraints between the traversal scheme and the test configuration that prune the number of combinations. The Min Hi-Z is only used with a single layer and the Min-Max Hi-Z is only used with a constant thickness. The average time and standard deviation of each task is calculated based on the test configuration and traversal scheme.

3.4.2 Statistical Significance

To test how each traversal scheme performs relative to the other traversal schemes for a given test configuration, the *Traverse* times are compared. To verify which *Traverse* timing measurements have statistically significant differences, a two-sample z-test [44] is performed for each combination of traversal scheme pairs on the various test configurations. The two-sample z-test compares the following equation with a critical value z_α with a significance level α to test a hypothesis:

$$z = \frac{(\bar{x}_1 - \bar{x}_2) - (\mu_1 - \mu_2)}{\sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}}}$$

where $\bar{x}_i$ is the sample average, n_i is the sample size, μ_i is the unknown population average, σ_i is the unknown population standard deviation for the *Traverse* time of traversal scheme i.

Given two distinct traversal schemes T_1 and T_2, with corresponding traversal time sample averages $\bar{x}_i$ and sample standard deviations s_i where $\bar{x}_1 < \bar{x}_2$, it is hypothesized

that T_1 performs better than T_2 on a given test configuration. A lower one-tailed z-test is performed to test the alternative hypothesis $H_A : \mu_1 < \mu_2$, with a null hypothesis $H_0 : \mu_1 = \mu_2$, where μ_i is the unknown average population of all frames. The significance level chosen for the test is $\alpha = 0.05$, which results in a critical value of $z_\alpha = -1.645$ for a left one tailed z-test. The null hypothesis is rejected if $z \leq z_\alpha$. If the null hypothesis is rejected, then the alternative hypothesis that T_1 performs better than T_2 is accepted. In this case, the difference between the mean traversal times is said to be statistically significant.

Since the population standard deviations are unknown and the sample sizes are sufficiently high, the sample deviations s_i are used to approximate the population variance σ_i. In statistics, a sample size of $n \geq 30$ is considered sufficient to justify the use of a z-test [44]. In this experiment, averages are taken over 30 frames ($n_1 = n_2 = 30$).

The results of the z-tests for the varying traversal schemes on each test configuration are given in Appendix I. The z-test results show the test configuration used, which two traversal schemes are compared, the z value, and whether the null hypothesis H_0 is rejected. Most of the z-tests result in the null hypothesis being rejected, but these results are discussed further in Chapter 4.

3.4.3 Outlier Handling

Upon manual analysis of the data, there appeared to be some outliers in the data causing higher averages and standard deviations for some setups. Often some setups had a majority of data points that had very similar timing data and a few data points (around 0-3 out of 30 for a setup), with abnormally high values. Sometimes, these high data points caused the frame time to double or triple. Such data points can potentially be attributed to the computer system performing some other task during the time period that the frame was processed. It

is difficult to determine the exact cause, but to provide a fairer comparison between the traversal schemes, it is beneficial to remove such outliers. Outlier removal allows for a better representation of the time spent performing the processing relevant to the frame.

Outlier detection is performed on the data to find frames where the frame time is 2.5 or more standard deviations away from the average frame time for a setup. In a normal distribution, the data that is not within 2.5 standard deviations of the mean accounts for about 1.24% of the data. In each setup, about 4.4% of the frames where determined to be outliers. Initially, this outlier detection scheme was performed for each of the recorded rendering tasks such as *Traverse* and *Build*, but this was found to aggressively discard data for small rendering tasks that were relatively constant. So, the outlier detection is only performed with respect to the average frame time. A limitation of this approach is that frames that were rendered relatively fast with respect to the mean are also discarded.

When an outlier is detected, the timing data for the frame is discarded and new timing data for the frame is obtained. Outlier detection is performed again with the new data using the original mean and standard deviation. If the new data is also an outlier, the process repeats until no new outliers are detected. A new average and standard deviation are calculated with the resolved data.

For the detected outliers, 10.9% of the frames replaced by new frames were also detected as outliers. Of all the frames considered, including those rejected as outliers, 4.6% of them were detected as outliers.

Chapter 4 Results

This chapter presents the results of the evaluation described in Chapter 3. Section 4.1 shows the output image generated using each traversal scheme. In Section 4.2, the average rendering time to perform different rendering tasks is presented and discussed. Varying traversal schemes are used along with different parameters. Section 4.3 evaluates the number of traversal iterations used by each traversal scheme. In Section 4.4, the results are discussed.

4.1 Output Images

The final rendered images produced by applying five SSR traversal schemes are shown in Figure 4-1 and Figure 4-2 for the Sponza and Bistro scenes, respectively. Reflections are shown in exaggerated form to clearly see them. As well, only a limited amount of artifact resolution is performed to fade the edges of the reflection. The fading is limited here to accentuate the pixels that have hit points.

Among the five traversal schemes, 3D ray marching is noteworthy for producing several artifacts due to the larger sampling interval and thickness value used. The two DDA

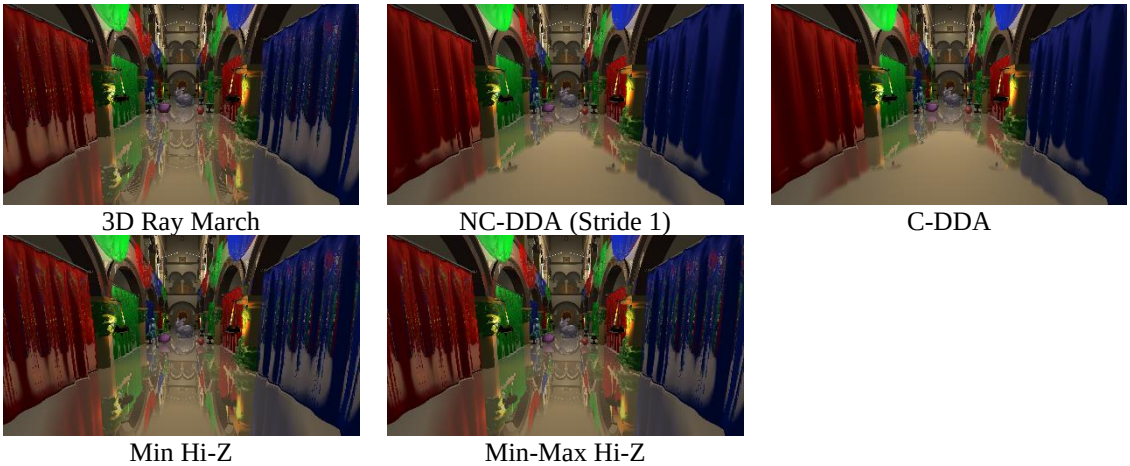


Figure 4-1: Sponza scene rendered with different SSR techniques using a single layer. (original in colour)

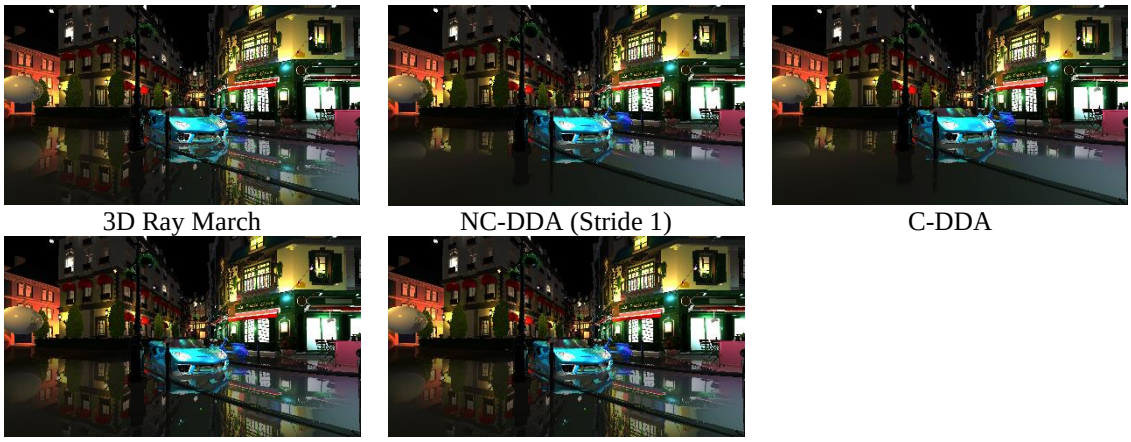


Figure 4-2: Bistro scene rendered with different SSR techniques using a single layer. (original in colour)

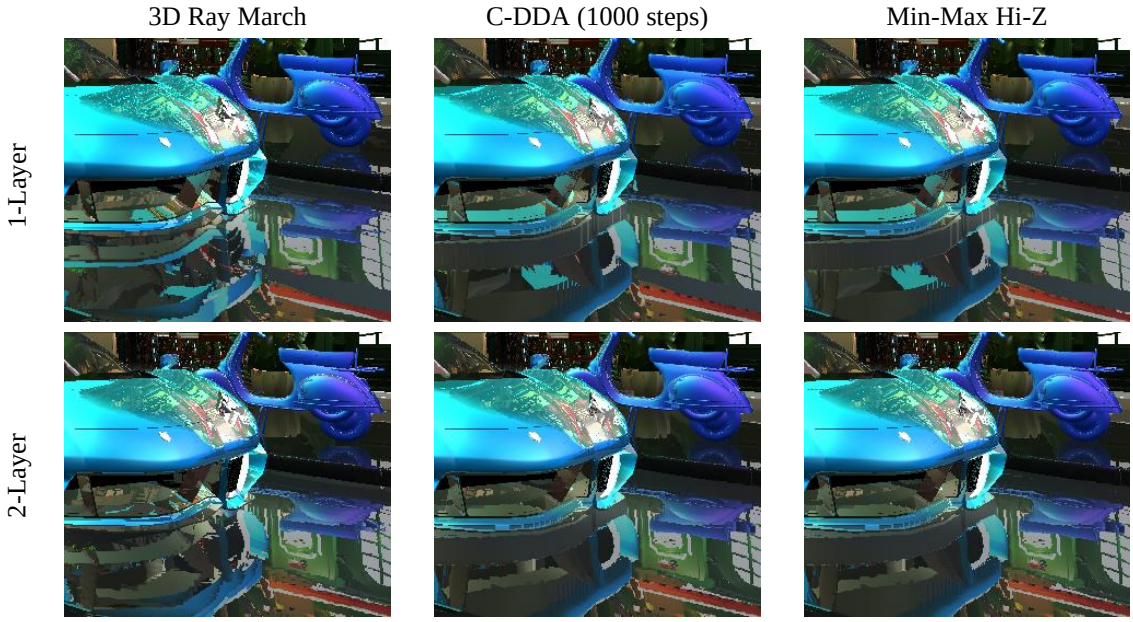


Figure 4-3: A close-up view of the reflection in the Bistro scene for 1 and 2 layers. (original in colour)

traversal schemes provide similar results. C-DDA has slightly fewer reflection hit points due to the extra sample points required to conservatively check the ray. Though difficult to see in the images, the reflections are slightly longer with NC-DDA than C-DDA. With the resolution and maximum number of sample points used, both DDA schemes fail to create reflections that cover the screen. The Min Hi-Z and Min-Max Hi-Z traversal schemes give nearly identical results.

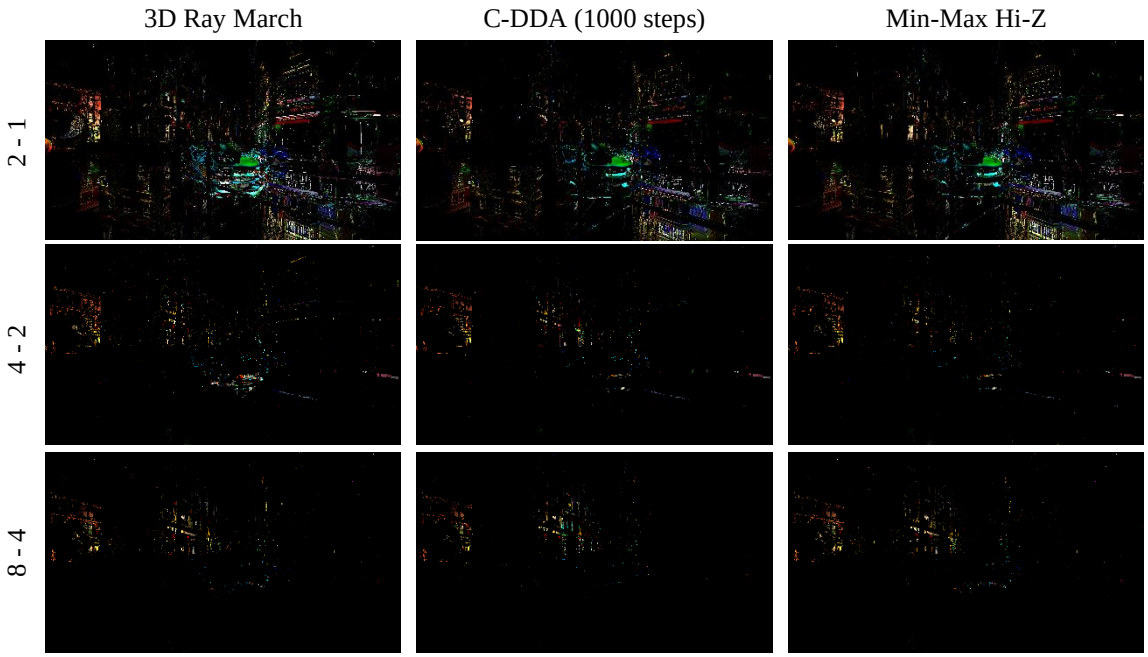


Figure 4-4: Colour difference of multi layer SSR on Bistro. Brightness has been increased by 50%. (original in colour)

A close-up of the bistro scene rendered with both single and 2-layer SSR is shown in Figure 4-3. The ray marching scheme produces banding artifacts visible in the reflection of the car. C-DDA and Min-Max Hi-Z produce similar images, but C-DDA requires many steps to generate a full reflection. A maximum step count of 1000 is used with the C-DDA in Figure 4-3. The use of two layers resolves some of the artifacts caused by occlusion, such as the reflection of the car on the ground and the reflection of the red awning (a red line at the bottom right of the image). The red awning occludes part of the building for a few pixels above the red line when using only a single layer, but the pixels are resolved when two layers are used. Figure 2-27 on page 45 shows a different view of the Sponza scene with single and multiple layers.

Each additional layer used with SSR further resolves holes caused by occlusion artifacts. Figure 4-4 shows the colour difference between the final rendered results with different layer counts used. Each row shows the difference between the image using a

certain layer count and the image using the next smaller layer count that was tested. For example, the 2-1 image in the first column shows the colour difference between the image generated with 3D ray marching with two layers and the image generated with one layer. The brightness is increased by 50% to more clearly illustrate the differences between layers. There is a greater pixel difference in the 2-1 images than the 4-2 and 8-4 images. However, there are two problems that inflate this result.

First, the reflection images generated with single-layer SSR and with multi-layer SSR are offset from each other by a few pixels using the current implementation. This is potentially due to depth precision issues. The implementation processes depth values slightly differently when using a single layer than when using multiple layers. The single layer implementation uses hyperbolic depth converted to linear depth, while the multi-layer one uses the raw linear depth value obtained from projection, which is normalized to an unsigned integer when storing the fragments into packed data. The depth is then converted back to floating point in the *Build* stage. These two mathematically equivalent methods may produce different results due to precision error. The slightly shifted reflection causes edges to be highlighted in the 2-1 difference images.

The other problem related to the 2-1 difference images is that the multi-layer implementation of the geometry pass only stores one floating value for the specular component of each fragment, whereas single-layer implementation stores each RGB component. This was done to pack the fragment data to 16 bytes for efficient access. When using multiple layers, the average of each specular RGB component is stored, as is performed by Hofmann et al. [32]. This simplified scheme changes the specular colour in the output images, which affects the difference images.

Table 4-1: Average GPU time (ms). (Spona scene) (1920x1080 resolution) (1 Layer)

	Geometry Pass			Lighting Pass		SSR		
Traversal Scheme	**Draw**	**Sort**	**Build**	**Tile Bounds**	**Cull + Shade**	**Traverse**	**Apply Hits**	**Total**
3D Ray March	0.2520	---	0.0300	0.0377	0.2500	4.1230	0.1627	4.9787
NC-DDA	0.2510	---	0.0300	0.0367	0.2500	3.1453	0.1003	3.9420
C-DDA	0.2530	---	0.0300	0.0390	0.2560	4.2137	0.1003	5.0230
Min Hi-Z	0.2577	---	0.0697	0.0367	0.2543	3.7517	0.1883	4.6740
Min-Max Hi-Z	0.2710	---	0.1123	0.0373	0.2650	1.1160	0.1790	2.0917

Table 4-2: Average GPU time (ms). (Sponza scene) (1920x1080 resolution) (2 Layer)
(The bracket connection in the Traverse column denotes a difference that is not statistically significant.)

	Geometry Pass			Lighting Pass		SSR		
Traversal Scheme	**Draw**	**Sort**	**Build**	**Tile Bounds**	**Cull + Shade**	**Traverse**	**Apply Hits**	**Total**
3D Ray March	1.1303	0.9720	0.2503	0.0700	0.4490	6.5783	0.1717	9.7573
NC-DDA	1.1590	0.9570	0.2603	0.0700	0.4717	7.6897	0.1010	10.8290
C-DDA	1.1650	0.9837	0.2607	0.0700	0.4550	7.7833	0.0973	10.9323
Min-Max Hi-Z	1.1507	0.9540	0.6700	0.1100	0.4497	2.2550	0.1630	5.8610

Table 4-3: Average GPU time (ms). (Sponza scene) (1920x1080 resolution) (4 Layer)

	Geometry Pass			Lighting Pass		SSR		
Traversal Scheme	**Draw**	**Sort**	**Build**	**Tile Bounds**	**Cull + Shade**	**Traverse**	**Apply Hits**	**Total**
3D Ray March	1.1107	0.9927	0.5440	0.1400	0.6763	11.5173	0.1903	15.2663
NC-DDA	1.1997	1.0020	0.5810	0.1400	0.6817	12.8307	0.1050	16.6557
C-DDA	1.1637	1.0040	0.5697	0.1403	0.6993	13.2757	0.1003	17.0557
Min-Max Hi-Z	1.1573	0.9800	1.3097	0.2190	0.6773	2.6170	0.1747	7.2243

Table 4-4: Average GPU time (ms). (Sponza scene) (1920x1080 resolution) (8 Layer)

	Geometry Pass			Lighting Pass		SSR		
Traversal Scheme	**Draw**	**Sort**	**Build**	**Tile Bounds**	**Cull + Shade**	**Traverse**	**Apply Hits**	**Total**
3D Ray March	1.1143	1.3573	1.1870	0.2873	0.8943	16.5553	0.1970	21.6693
NC-DDA	1.1730	1.3787	1.2493	0.2863	0.8870	14.7857	0.1043	19.9400
C-DDA	1.1637	1.3653	1.2300	0.3037	0.9020	15.2437	0.1000	20.3970
Min-Max Hi-Z	1.1533	1.3460	2.4563	0.4313	0.8933	2.4170	0.1777	8.9417

4.2 Rendering Time

The rendering performance of SSR using various traversal schemes was recorded. The screen resolution used in the performance tests is 1920x1080 unless otherwise specified. The GPU times for the various rendering tasks, including deferred shading and SSR, are shown in Table 4-1 through Table 4-4 for the Sponza scene. Each column in the tables is the average GPU time over 30 frames to perform a rendering task for different traversal schemes, where a rendering task is the range of relevant sections of the code to

perform a rendering process. All times are measured in milliseconds. Each table shows the average GPU times for a different number of layers. The different layer counts tested are 1, 2, 4 and 8.

The total column shows the average GPU time to render the entire frame. The times for the different rendering tasks may not sum to the total, because only the relevant rendering tasks are included here and because of rounding. Standard deviations for the averages in Table 4-1 through Table 4-4 are given in Appendix H-1 through Appendix H-4. The standard deviations were calculated using the sample variance formula.

The geometry pass contains the *Draw* and *Sort* tasks. *Draw* involves generating the fragments for the G-Buffer. For single layer rendering, *Draw* is just the G-Buffer generation, but for multi-layer rendering, *Draw* involves generating the unsorted linked list of fragments for each pixel as well as sorting the lists. The *Sort* task is only relevant when using multiple layers and it involves performing a truncated sort for k layers.

The *Build* task involves converting the depth buffer to linear depth when using a linear traversal scheme. When using a hierarchical traversal scheme, the *Build* range also involves constructing the Hi-Z Buffer. When using multi-layer Hi-Z schemes, the build process is more complex and uses the approach discussed in Section 2.5 and Section 3.2.3.

The lighting pass contains the *Tile Bounds* and the *Cull+Shade* tasks. The *Tile Bounds* task computes the minimum and maximum depth in each tile of the screen, while the *Cull+Shade* culls the lights for the tiles and performs the shading calculations to determine the lighting in the pixel.

The *SSR* task involves the *Traverse* and *Apply Hits* tasks. The *Traverse* task involves performing the SSR traversal shader using the corresponding traversal scheme. *Traverse*

outputs the screen space hit points. The *Apply Hits* task takes the SSR hit points as input, reads the corresponding colour value, performs artifact resolution, and outputs the reflection colour to be blended over the image.

In Table 4-2 and in some subsequent tables in this chapter, there are brackets which connect two traversal schemes together in the $Traverse$ column. The connections indicate that the differences between the two measurements were not found to be statistically significant using a two-sample z-test. For example, the NC-DDA and C-DDA in Table 4-2 have a connection, indicating that there was not enough evidence to say that C-DDA performs better than the NC-DDA for the test configuration. In the cases with brackets, the *Traverse* times are very similar. A bracket corresponds to a failure to reject the null hypothesis, which is indicated in Appendix I where the "*Reject?*" column is zero. Thus, in this table and in subsequent tables, if there are no brackets connecting any two traversal schemes, then according to the method outlined in Section 3.4, the difference between the average *Traverse* times is statistically significant.

Table 4-5 through Table 4-8 shows the timing measurements using the same experimental setup as for Table 4-1 through Table 4-4, except applied to the Bistro scene. Standard deviations for the averages in Table 4-5 through Table 4-8 are given in Appendix H-5 through Appendix H-8.

The average GPU time for the geometry pass increases with the number of layers. The time for the multi-layer *Draw* task remains about the same for each layer count because all fragments are written to the fragment buffer regardless of the layer count of the K-Buffer. The time for the *Sort* stage increases with the layer count since the fragment lists are truncated to k layers.

Table 4-5: Average GPU time (ms). (Bistro scene) (1920x1080 resolution) (1 Layer)

	Geometry Pass			Lighting Pass		SSR		
Traversal Scheme	**Draw**	**Sort**	**Build**	**Tile Bounds**	**Cull + Shade**	**Traverse**	**Apply Hits**	**Total**
3D Ray March	0.8303	---	0.0300	0.0393	0.6300	4.2180	0.1423	5.9643
NC-DDA	0.7563	---	0.0300	0.0390	0.6300	2.7120	0.1000	4.3293
C-DDA	0.7653	---	0.0300	0.0387	0.6300	3.6020	0.0900	5.2243
Min Hi-Z	0.7483	---	0.0700	0.0387	0.6300	2.7457	0.1267	4.4337
Min-Max Hi-Z	0.7507	---	0.1363	0.0383	0.6523	1.2343	0.1300	3.0053

Table 4-6: Average GPU time (ms). (Bistro scene) (1920x1080 resolution) (2 Layer)

	Geometry Pass			Lighting Pass		SSR		
Traversal Scheme	**Draw**	**Sort**	**Build**	**Tile Bounds**	**Cull + Shade**	**Traverse**	**Apply Hits**	**Total**
3D Ray March	4.6347	2.3987	0.2560	0.0700	1.0000	9.9877	0.1200	18.5193
NC-DDA	4.5997	2.4163	0.2537	0.0700	0.9977	6.5553	0.0910	15.0527
C-DDA	4.6037	2.4257	0.2517	0.0703	1.0287	6.2843	0.0903	14.8170
Min-Max Hi-Z	4.6133	2.4167	0.6970	0.1100	0.9960	1.5967	0.1200	10.5973

Table 4-7: Average GPU time (ms). (Bistro scene) (1920x1080 resolution) (4 Layer)

	Geometry Pass			Lighting Pass		SSR		
Traversal Scheme	**Draw**	**Sort**	**Build**	**Tile Bounds**	**Cull + Shade**	**Traverse**	**Apply Hits**	**Total**
3D Ray March	4.5853	2.4183	0.5267	0.1400	1.6407	19.2880	0.1300	28.7643
NC-DDA	4.6103	2.4600	0.5277	0.1407	1.6400	11.9667	0.0997	21.4907
C-DDA	4.6610	2.4420	0.5293	0.1400	1.6403	10.3703	0.0903	19.9193
Min-Max Hi-Z	4.6370	2.9523	1.3313	0.2193	1.6400	1.8757	0.1200	12.8083

Table 4-8: Average GPU time (ms). (Bistro scene) (1920x1080 resolution) (8 Layer)

	Geometry Pass			Lighting Pass		SSR		
Traversal Scheme	**Draw**	**Sort**	**Build**	**Tile Bounds**	**Cull + Shade**	**Traverse**	**Apply Hits**	**Total**
3D Ray March	4.6307	2.5293	1.1103	0.2800	2.4470	28.2200	0.1380	39.3657
NC-DDA	4.6347	2.5283	1.1123	0.2783	2.4387	26.0113	0.1000	37.1277
C-DDA	4.7613	2.5230	1.1110	0.2783	2.4317	24.0683	0.0907	35.2857
Min-Max Hi-Z	4.5813	2.5247	2.5457	0.4327	2.4333	1.9923	0.1287	14.6583

The average time for the lighting pass scales linearly with the layer count. The time for the *Tile Bounds* stage increases with the layer count since the tile bounds must be generated for each layer. The time for the *Cull+Shade* stage increases with the layer count, but the time for the multi-layer version is less than $k \times t_S$, where t_S is the time to shade a single layer and k is the layer count. This occurs because some tiles are culled when they have empty tile bounds. For each layer, a minimum and maximum depth value is

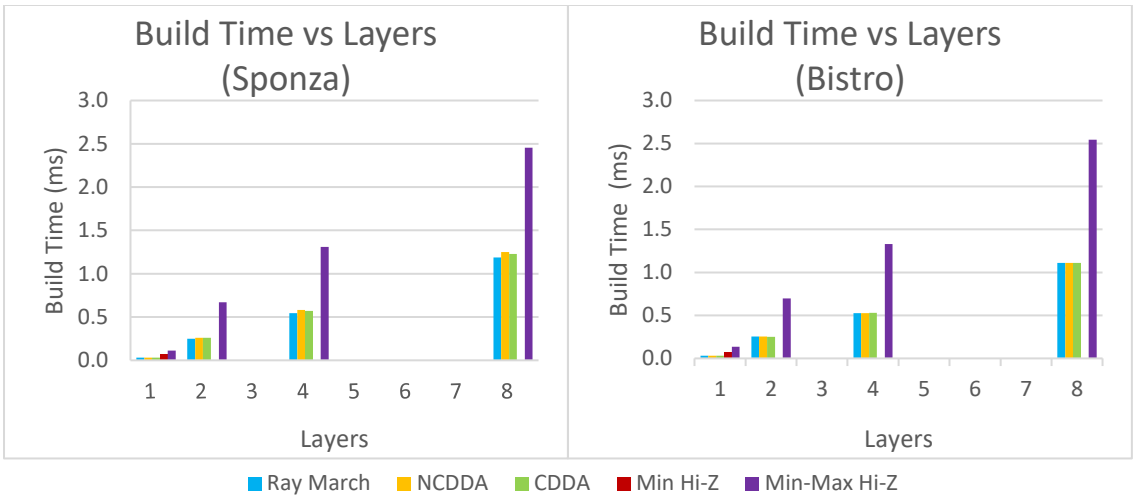


Figure 4-5: Build Time vs Layer Count. (1920x1080 resolution) (original in colour)

determined for each tile. For tiles that contain no fragments, an empty tile bounds is generated, allowing shading calculations to be skipped for the tile.

Although the lighting pass scales linearly with the layer count, the geometry pass is still quite expensive when using multiple layers due to the high constant cost of the *Draw* stage. For most of the multi-layer data, the time to perform the geometry pass is larger than $k \times t_G$, where t_G is the time to generate to the G-Buffer for a single layer. Only on the Bistro scene with 8 layers and a resolution of 1280x720, as shown later in Section 4.2.1 (Table 4-16), was the multi-layer G-Buffer generation less than or equal to $k \times t_G$. This indicates that perhaps the current implementation would be outperformed by other depth peeling methods or a more optimized implementation. Either way, generating multiple layers is an expensive process.

The *Build* stage increases with layer count because the data structure must be generated to account for each layer. Figure 4-5 shows the build time for varying layer count. With a Hi-Z traversal scheme, the Build stage is increased compared to the linear traversal schemes since a mip chain must be generated in addition to converting to linear

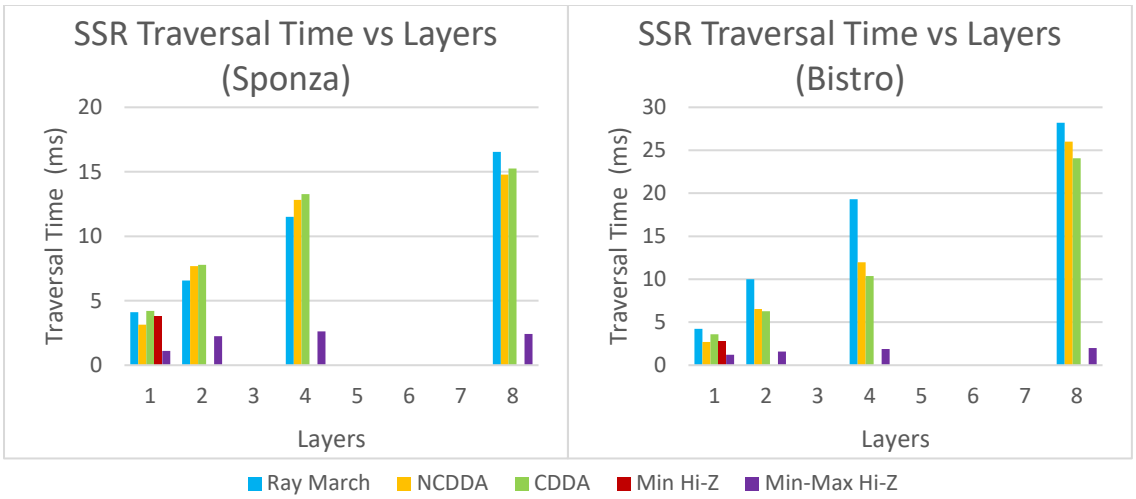


Figure 4-6: SSR Traversal Time vs Layer Count. Y axis differs. (1920x1080 resolution) (original in colour)

depth. The Min-Max Hi-Z traversal scheme has an increased cost when using multiple layers since intervals of fragments are merged together into fragment intervals.

Figure 4-6 shows the SSR *Traverse* time with differing layer counts. The linear traversal schemes scale proportionately with the layer count. Compared to the Min-Max Hi-Z, the linear traversal schemes scale quickly. The Min-Max Hi-Z scales well with the layer count but requires the increased *Build* time shown in Figure 4-5. The ray marching scheme requires more traversal time on the Bistro scene than the DDA schemes.

There is one odd aspect in the data for the Min-Max Hi-Z *Traverse* time. For the Sponza scene, the traversal time for 8 layers is less than the traversal time for 4 layers. This anomaly is shown more clearly in Table 4-1 through Table 4-4. The anomaly is likely caused by a hardware related constraint on the implementation where the 2 and 4-layer shaders are using more hardware registers than the 8-layer shader, while the 8-layer shader uses the optimal number of registers, but also uses local memory. The increase in registers is causing the GPU occupancy to be lowered for the 2- and 4-layer versions of the shader.

Occupancy is the ratio of active GPU warps over the maximum number of warps per streaming multiprocessor. The 8-layer version was found to have higher occupancy for the

hardware used. Each multi-layer Min-Max Hi-Z shader was found to have about 80% of the maximum theoretical occupancy for the shader based on a single profile run for each layer size. The 2 and 4-layer shaders had a maximum theoretical occupancy of 75% while the 8-layer version had 100%. Despite the register spills, the 8-layer shader performs faster on the hardware used.

Table 4-9: Average GPU time (ms). (Sponza scene) (1280x720 resolution) (1 Layer)
(the bracket connection in the Traverse column denotes a difference that is not statistically significant.)

	Geometry Pass			**Lighting Pass**		**SSR**		
Traversal Scheme	**Draw**	**Sort**	**Build**	**Tile Bounds**	**Cull + Shade**	**Traverse**	**Apply Hits**	**Total**
3D Ray March	0.1540	---	0.0100	0.0200	0.1200	1.7630	0.0703	2.1860
NC-DDA	0.1603	---	0.0100	0.0200	0.1200	1.2790	0.0587	1.6977
C-DDA	0.1607	---	0.0100	0.0200	0.1200	1.7983	0.0500	2.2097
Min Hi-Z	0.1777	---	0.0300	0.0200	0.1200	1.7100	0.0833	2.1730
Min-Max Hi-Z	0.1787	---	0.0500	0.0000	0.1200	0.4900	0.0803	0.9537

Table 4-10: Average GPU time (ms). (Sponza scene) (1280x720 resolution) (2 Layer)

	Geometry Pass			**Lighting Pass**		**SSR**		
Traversal Scheme	**Draw**	**Sort**	**Build**	**Tile Bounds**	**Cull + Shade**	**Traverse**	**Apply Hits**	**Total**
3D Ray March	0.7397	0.4103	0.1200	0.0463	0.2190	2.7307	0.0777	4.3877
NC-DDA	0.7577	0.4117	0.1197	0.0483	0.2190	3.1230	0.0547	4.7787
C-DDA	0.7563	0.4083	0.1200	0.0487	0.2180	3.2783	0.0500	4.9257
Min-Max Hi-Z	0.7510	0.4113	0.3603	0.0607	0.2190	0.9613	0.0720	2.8760

Table 4-11: Average GPU time (ms). (Sponza scene) (1280x720 resolution) (4 Layer)

	Geometry Pass			**Lighting Pass**		**SSR**		
Traversal Scheme	**Draw**	**Sort**	**Build**	**Tile Bounds**	**Cull + Shade**	**Traverse**	**Apply Hits**	**Total**
3D Ray March	0.7440	0.4300	0.2510	0.0900	0.3377	3.8557	0.0867	5.8387
NC-DDA	0.7583	0.4317	0.2510	0.0903	0.3393	5.4483	0.0590	7.4217
C-DDA	0.7757	0.4320	0.2517	0.0907	0.3373	5.9343	0.0510	7.9170
Min-Max Hi-Z	0.7517	0.4357	0.6607	0.1300	0.3403	1.1287	0.0800	3.5513

Table 4-12: Average GPU time (ms). (Sponza scene) (1280x720 resolution) (8 Layer)

	Geometry Pass			**Lighting Pass**		**SSR**		
Traversal Scheme	**Draw**	**Sort**	**Build**	**Tile Bounds**	**Cull + Shade**	**Traverse**	**Apply Hits**	**Total**
3D Ray March	0.7373	0.6010	0.5497	0.1810	0.4557	4.6463	0.0873	7.2757
NC-DDA	0.7803	0.6010	0.5473	0.1827	0.4547	6.6730	0.0563	9.3077
C-DDA	0.7823	0.6003	0.5487	0.1830	0.4570	6.9207	0.0547	9.5610
Min-Max Hi-Z	0.7517	0.6000	1.1560	0.2507	0.4593	1.0393	0.0800	4.3477

Table 4-13: Average GPU time (ms). (Bistro scene) (1280x720 resolution) (1 Layer)
(The bracket connection in the Traverse column denotes a difference that is not statistically significant.)

	Geometry Pass			Lighting Pass		SSR		
Traversal Scheme	**Draw**	**Sort**	**Build**	**Tile Bounds**	**Cull + Shade**	**Traverse**	**Apply Hits**	**Total**
3D Ray March	0.7597	---	0.0100	0.0200	0.3153	1.3747	0.0503	2.5293
NC-DDA	0.7000	---	0.0100	0.0200	0.3113	1.1277	0.0500	2.2407
C-DDA	0.6793	---	0.0100	0.0200	0.2997	1.4910	0.0500	2.5700
Min Hi-Z	0.7063	---	0.0367	0.0200	0.2997	1.3650	0.0600	2.5017
Min-Max Hi-Z	0.6807	---	0.0500	0.0000	0.3053	0.5563	0.0603	1.6640

Table 4-14: Average GPU time (ms). (Bistro scene) (1280x720 resolution) (2 Layer)

	Geometry Pass			Lighting Pass		SSR		
Traversal Scheme	**Draw**	**Sort**	**Build**	**Tile Bounds**	**Cull + Shade**	**Traverse**	**Apply Hits**	**Total**
3D Ray March	3.9583	1.0617	0.1133	0.0480	0.4943	2.9460	0.0600	8.6867
NC-DDA	3.9810	1.0760	0.1167	0.0497	0.4800	2.7247	0.0500	8.4840
C-DDA	3.9587	1.0640	0.1173	0.0453	0.4800	2.6917	0.0500	8.4133
Min-Max Hi-Z	3.9400	1.0633	0.3700	0.0613	0.4800	0.7103	0.0600	6.6780

Table 4-15: Average GPU time (ms). (Bistro scene) (1280x720 resolution) (4 Layer)

	Geometry Pass			Lighting Pass		SSR		
Traversal Scheme	**Draw**	**Sort**	**Build**	**Tile Bounds**	**Cull + Shade**	**Traverse**	**Apply Hits**	**Total**
3D Ray March	3.9467	1.1400	0.2410	0.0900	0.8280	5.7133	0.0600	12.0127
NC-DDA	4.0223	1.0973	0.2413	0.0900	0.8113	4.6663	0.0500	10.9780
C-DDA	4.0017	1.1063	0.2400	0.0903	0.8117	4.4453	0.0500	10.7613
Min-Max Hi-Z	3.9527	1.1010	0.6927	0.1437	0.8120	0.8897	0.0600	7.6273

Table 4-16: Average GPU time (ms). (Bistro scene) (1280x720 resolution) (8 Layer)

	Geometry Pass			Lighting Pass		SSR		
Traversal Scheme	**Draw**	**Sort**	**Build**	**Tile Bounds**	**Cull + Shade**	**Traverse**	**Apply Hits**	**Total**
3D Ray March	3.9427	1.1380	0.5193	0.1823	1.2510	10.0457	0.0663	17.1043
NC-DDA	3.9740	1.1347	0.5187	0.1833	1.2523	9.5893	0.0503	16.6670
C-DDA	4.0173	1.1347	0.5203	0.1817	1.2720	9.2550	0.0500	16.3970
Min-Max Hi-Z	4.0240	1.1640	1.3107	0.3333	1.2500	1.0903	0.0600	9.1907

4.2.1 Resolution

Screen resolution directly affects the number of rays that need to be traversed for SSR. Table 4-9 through Table 4-12 show the average GPU task times for the Sponza scene with the same settings as Table 4-1 through Table 4-4, except that a lower resolution of 1280x720 is used instead 1920x1080. Standard deviations for the averages in Table 4-9 through Table 4-12 are given in Appendix H-9 through Appendix H-12. In Table 4-9, there is bracket connecting two traversal schemes indicating that the two traversal schemes failed

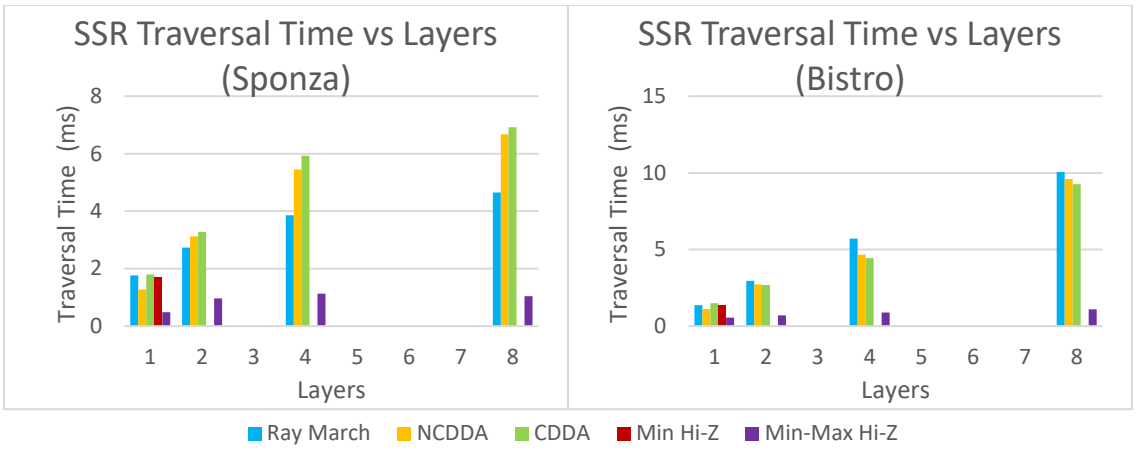


Figure 4-7: SSR Traversal Time vs Layer Count. Y axis differs. (1280x720 resolution) (original in colour)

the z-test. It is worth noting that the *Traverse* times for the connected schemes, the 3D ray march and the Min Hi-Z in this case, have very similar values.

The GPU timing data for same experimental setup, but with the Bistro scene are given in Table 4-13 through Table 4-16. Standard deviations for the averages in Table 4-13 through Table 4-16 are given in Appendix H-13 through Appendix H-16. When the resolution is decreased, the time to perform each rendering task is also decreased since there are fewer pixels to process. The lower resolution also affects the DDA traversal schemes for a different reason. As the resolution is lowered, the DDA may produce a higher fraction of hit pixels since the maximum number of iterations is not scaled with the resolution.

Figure 4-7 graphs the SSR traversal time for the 1280x720 resolution. The trends are similar to those with the 1920x1080 resolution. Table 4-17 and Table 4-18 show summaries of only the SSR *Traverse* tasks from the average times from Table 4-1 through Table 4-16.

To compare how the traversal schemes scale with resolution, the relative traversal times using the two resolutions are shown in Figure 4-8. The *relative traversal time* is defined as the ratio of the traversal times for the two resolutions. If the traversal time scales

Table 4-17: Average SSR Traversal time (ms). Summary of Table 4-1 through Table 4-8. (1920x1080 resolution)

Traversal Scheme	Sponza Scene				Bistro Scene			
	1 Layer	2 Layer	4 Layer	8 Layer	1 Layer	2 Layer	4 Layer	8 Layer
3D Ray March	4.1230	6.5783	11.5173	16.5553	4.2180	9.9877	19.2880	28.2200
NC-DDA	3.1453	7.6897	12.8307	14.7857	2.7120	6.5553	11.9667	26.0113
C-DDA	4.2137	7.7833	13.2757	15.2437	3.6020	6.2843	10.3703	24.0683
Min Hi-Z	3.7517	---	---	---	2.7457	---	---	---
Min-Max Hi-Z	1.1160	2.2550	2.6170	2.4170	1.2343	1.5967	1.8757	1.9923

Table 4-18: Average SSR Traversal time (ms). Summary of Table 4-9 through Table 4-16. (1280x720 resolution)

Traversal Scheme	Sponza Scene				Bistro Scene			
	1 Layer	2 Layer	4 Layer	8 Layer	1 Layer	2 Layer	4 Layer	8 Layer
3D Ray March	1.7630	2.7307	3.8557	4.6463	1.3747	2.9460	5.7133	10.0457
NC-DDA	1.2790	3.1230	5.4483	6.6730	1.1277	2.7247	4.6663	9.5893
C-DDA	1.7983	3.2783	5.9343	6.9207	1.4910	2.6917	4.4453	9.2550
Min Hi-Z	1.7100	---	---	---	1.3650	---	---	---
Min-Max Hi-Z	0.4900	0.9613	1.1287	1.0393	0.5563	0.7103	0.8897	1.0903

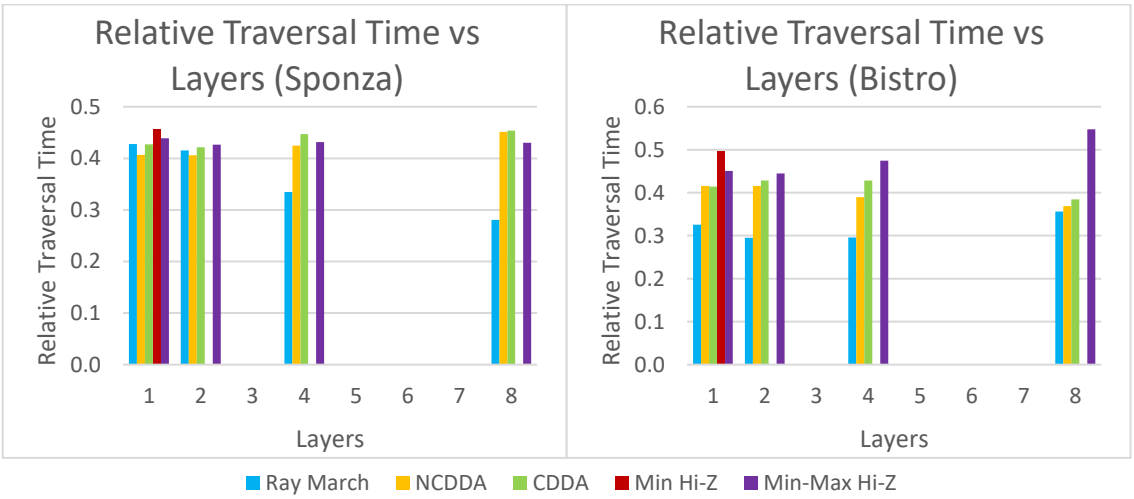


Figure 4-8: Relative SSR Traversal Time vs Layer Count as the ratio of the traversal times for the 1280x720 and 1920x1080 resolutions. Relative Traversal Time is a unitless quantity. (original in colour)

proportionally to the resolution size, the relative traversal time should be equal to the ratio the number of pixels processed. The ratio of the number of pixels for the two resolutions is $\frac{1280\times720}{1920\times1080} = 0.44$ and most of the relative traversal times are clustered near this value. The relative traversal time changes with the layer count differently on the two scene views and no common trends were observed.

4.2.2 Infinite Thickness

To test the performance of the traversal schemes without the impact of occlusion, an intersection scheme with infinite thickness was used. With infinite thickness, the SSR traversal process terminates when a ray becomes occluded, whereas the traversal may continue with a constant thickness. Using infinite thickness results in holes or artifacts in the reflection. The timing data using infinite thickness is shown for 1920x1080 resolution

Table 4-19: Average GPU time (ms). (Sponza scene) (1920x1080 resolution) (1 Layer) (Uses infinite thickness intersection scheme)

	Geometry Pass			**Lighting Pass**		**SSR**		
Traversal Scheme	**Draw**	**Sort**	**Build**	**Tile Bounds**	**Cull + Shade**	**Traverse**	**Apply Hits**	**Total**
3D Ray March	0.2543	---	0.0300	0.0377	0.2503	1.6070	0.1703	2.4810
NC-DDA	0.2533	---	0.0300	0.0390	0.2550	2.8037	0.1080	3.6047
C-DDA	0.2567	---	0.0300	0.0370	0.2553	3.8607	0.1000	4.6700
Min Hi-Z	0.2577	---	0.0690	0.0377	0.2567	0.9690	0.1803	1.8767

Table 4-20: Average GPU time (ms). (Bistro scene) (1920x1080 resolution) (1 Layer) (Uses infinite thickness intersection scheme)

	Geometry Pass			**Lighting Pass**		**SSR**		
Traversal Scheme	**Draw**	**Sort**	**Build**	**Tile Bounds**	**Cull + Shade**	**Traverse**	**Apply Hits**	**Total**
3D Ray March	0.7520	---	0.0300	0.0390	0.6300	0.9687	0.1207	2.6070
NC-DDA	0.7623	---	0.0300	0.0390	0.6300	2.1003	0.0937	3.7250
C-DDA	0.7480	---	0.0300	0.0397	0.6300	2.7783	0.0900	4.4627
Min Hi-Z	0.7503	---	0.0687	0.0387	0.6300	0.8107	0.1203	2.4823

Table 4-21: Average GPU time (ms). (Sponza scene) (1280x720 resolution) (1 Layer) (Uses infinite thickness intersection scheme)

	Geometry Pass			**Lighting Pass**		**SSR**		
Traversal Scheme	**Draw**	**Sort**	**Build**	**Tile Bounds**	**Cull + Shade**	**Traverse**	**Apply Hits**	**Total**
3D Ray March	0.1550	---	0.0100	0.0200	0.1200	0.6217	0.0713	1.0503
NC-DDA	0.1607	---	0.0100	0.0200	0.1200	1.1887	0.0543	1.6030
C-DDA	0.1607	---	0.0100	0.0203	0.1200	1.5853	0.0503	1.9987
Min Hi-Z	0.1793	---	0.0300	0.0200	0.1200	0.4113	0.0800	0.8703

Table 4-22: Average GPU time (ms). (Bistro scene) (1280x720 resolution) (1 Layer) (Uses infinite thickness intersection scheme)

	Geometry Pass			**Lighting Pass**		**SSR**		
Traversal Scheme	**Draw**	**Sort**	**Build**	**Tile Bounds**	**Cull + Shade**	**Traverse**	**Apply Hits**	**Total**
3D Ray March	0.6770	---	0.0100	0.0200	0.3000	0.4620	0.0600	1.5547
NC-DDA	0.6767	---	0.0100	0.0200	0.2993	0.8503	0.0500	1.9253
C-DDA	0.6837	---	0.0100	0.0200	0.2997	1.1803	0.0500	2.2633
Min Hi-Z	0.6723	---	0.0393	0.0200	0.3000	0.3820	0.0600	1.4813

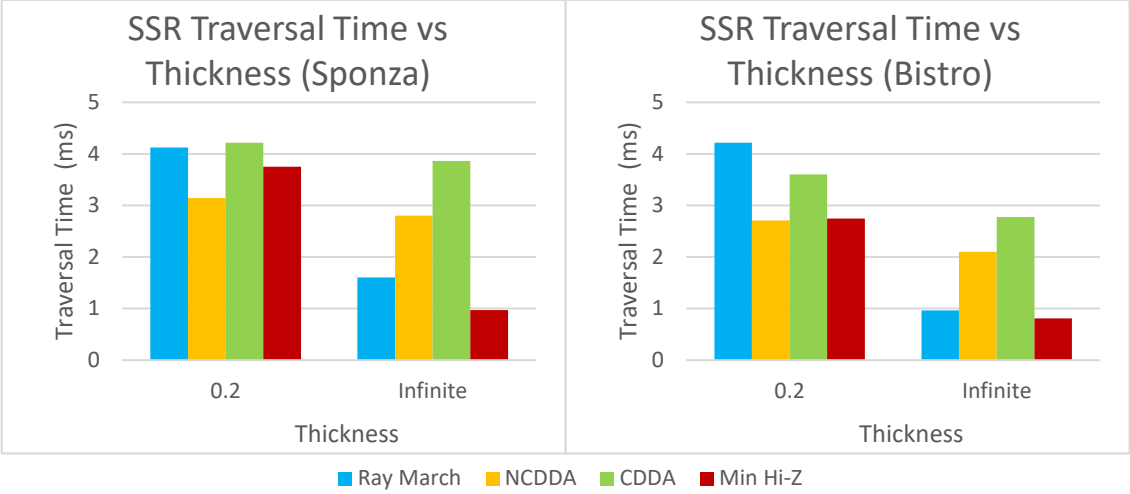


Figure 4-9: SSR Traversal Time vs Thickness. (1920x1080 resolution) (original in colour)

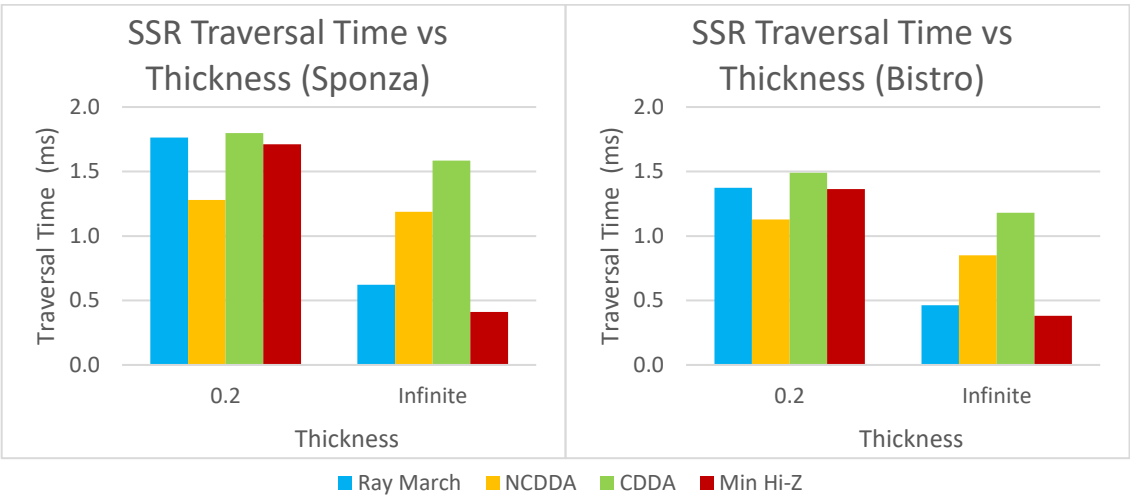


Figure 4-10: SSR Traversal Time vs Thickness. (1280x720 resolution) (original in colour)

for the Sponza and Bistro scenes in Table 4-19 and Table 4-20, respectively. Similar data is shown in Table 4-21 and Table 4-22 for the 1280x720 resolution.

The use of infinite thickness lowers the time in the *Traverse* stage since the traversal process terminates earlier in cases of occlusion. Figure 4-9 compares the traversal time using infinite thickness to the traversal time with a thickness of 0.2. The DDA schemes are not affected much by the infinite thickness because they still inefficiently traverse the empty space before intersecting the depth buffer. The 3D ray marching scheme and the Min Hi-Z have improved traversal times. With infinite thickness, the Min Hi-Z traversal

Table 4-23: Average Traverse time (ms) for varying batch size with 3D ray marching. (1920x1080 resolution) (1 Layer)

	No Batch	Batch-2	Batch-4	Batch-6	Batch-8
Sponza	4.1230	3.0930	2.4533	2.2507	2.1323
Bistro	4.2180	3.2370	2.7860	2.6360	2.5960

scheme does not have to traverse at a fine-grained resolution while a ray is occluded. Figure 4-10 shows similar results using the 1280x720 resolution.

4.2.3 Sample Batching

Several batch sizes were tested with the 3D ray marching traversal scheme. The implementation uses unrolled loops that repeat each step of the traversal process b times, where b is the batch size. Special care was taken to ensure that an unrolled loop used to process an array of sample points in a batch, did not compile using indexable temporary registers in the HLSL assembly. The shader performed faster with standard non-indexable temporary registers. Sample batching could potentially also benefit other traversal schemes, such as the DDA schemes, but they were not tested due to time constraints.

Table 4-23 shows the *Traverse* times with various batch sizes for the Sponza scene (standard deviations in Appendix H-21). Only a single layer was used for this test. Increasing the batch size decreases the traversal time. Increasing the batch size also increases the number of hardware registers required by the traversal shader. At higher batch sizes, less gains are obtained and eventually performance can be expected to degrade.

4.2.4 DDA Stride

The NC-DDA traversal scheme was tested with several stride sizes. Increasing the stride was not tested with the C-DDA scheme, since it would no longer be conservative. With NC-DDA, no binary search refinement was performed on the hit point to obtain a more accurate hit point. As a result, the reflections have banding artifacts that are left

Table 4-24: Average Traverse time (ms) using NC-DDA with different stride lengths. (1920x1080 resolution) (1 Layer)

	Stride-1	Stride-2	Stride-4	Stride-8
Sponza	3.1453	2.9460	2.3273	1.4317
Bistro	2.7120	3.0677	2.4560	1.6043

unresolved. Traversal times for varying stride sizes are shown in Table 4-24 (standard deviations in Appendix H-22). A decrease in traversal time is observed except in the case of stride size 2 on the Bistro scene. It is unknown why this difference occurs in this case. Using larger stride sizes lowers the quality of the reflections.

4.3 Traversal Iterations

The number of iterations in the traversal process depends on the view, the traversal scheme and the values of its parameters, to obtain a result. the Each traversal scheme requires a different amount of iterations in the traversal process to obtain a result. A traversal iteration is one iteration of the traversal process, which involves loading a sample point in a pixel or along the ray. It does not include loading each layer. To see which parts of the image are problematic, Figure 4-11 and Figure 4-12 show visualizations of the traversal iterations for the various traversal schemes with a small constant thickness. Only a single layer is used.

The linear traversal schemes have more iterations than the hierarchical schemes, which is indicated by the increased redness in the iteration images. The 3D Ray March scheme reaches the iteration cap when rays become occluded without becoming visible again, such as when the ray goes beneath the curtains of the Sponza scene on the sides. Without any stride applied, the DDA traversal schemes are poor at traversing open space. The DDA schemes can reach the maximum iteration count when traversing large amounts of space. This effect is seen in the lower portion of the images in Figure 4-11 and Figure 4-12 for C-

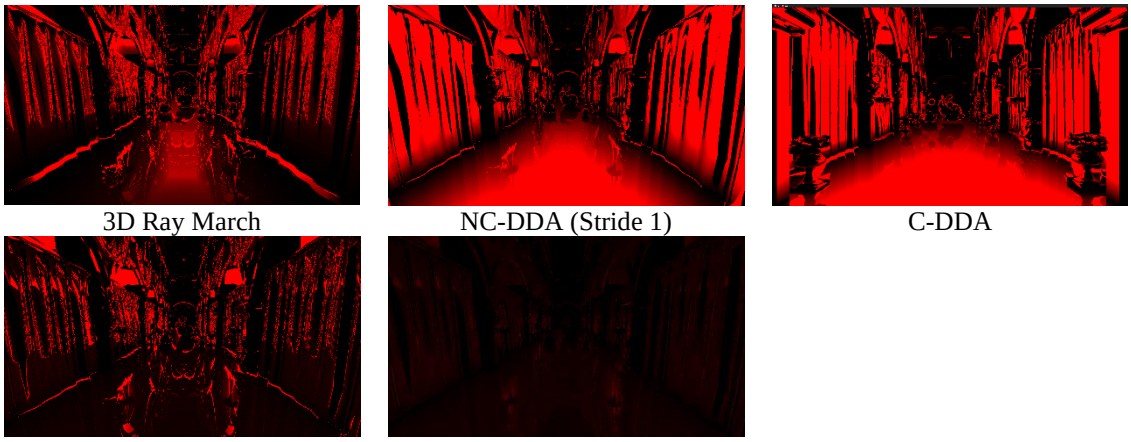


Figure 4-11: Visualization of single-layer traversal iterations for differing traversal schemes. Black is 0 iterations, Red is 400. Black is better. (Sponza) (original in colour)

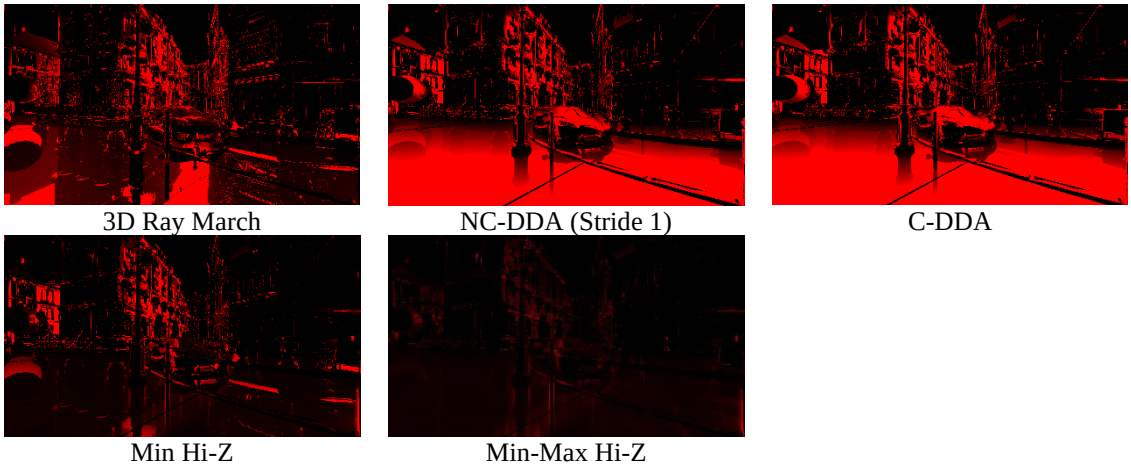


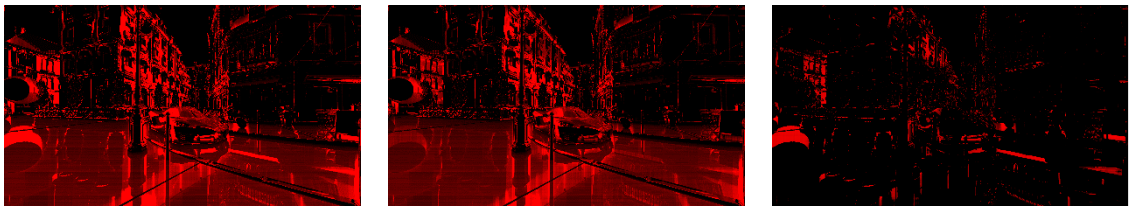
Figure 4-12: Visualization of single-layer traversal iterations for differing traversal schemes. Black is 0 iterations, Red is 400. Black is better. (Bistro) (original in colour)

DDA and NC-DDA. C-DDA requires slightly more iterations than NC-DDA, which is due to the additional pixels that are conservatively checked. Though difficult to see in the iteration images, C-DDA has more solid red pixels than NC-DDA.

The hierarchical traversal schemes have a relatively few traversal iterations except for certain areas in the image for the Min Hi-Z where many rays are becoming occluded. Min Hi-Z is efficiently skipping empty space, but its performance suffers when occlusion occurs. While the ray is occluded, traversal must operate at the finest resolution, which makes the traversal process equivalent or worse than the linear sampling schemes when the ray is occluded. Min-Max Hi-Z performs considerably fewer iterations than Min Hi-Z in

Min Hi-Z Min-Max Hi-Z Difference

Figure 4-13: Visualization of single-layer hierarchical traversal schemes. Black is 0 iterations, Red is ≥ 75. Black is better. (original in colour)



Min Hi-Z Min-Max Hi-Z Difference

Figure 4-14: Visualization of single-layer hierarchical traversal schemes. Black is 0 iterations, Red is ≥ 75. Black is better. (original in colour)

the areas where the ray becomes occluded. Figure 4-13 shows the number of traversal iterations for the hierarchical traversal schemes for the Sponza scene in Figure 4-11, but visualized with a maximum iteration count of 75. The difference between the visualization for Min Hi-Z and Min-Max Hi-Z highlights the areas where rays become occluded with the Min Hi-Z scheme. Min-Max Hi-Z efficiently skips empty space, while also efficiently traversing rays across occluded areas. The same effect is shown for the Bistro scene in Figure 4-14.

The average number of traversal iterations with varying numbers of layers for each image in Figure 4-11 and Figure 4-12 are shown in Table 4-25 and Table 4-26. Pixels that

Table 4-25: Average traversal iterations in Figure 4-11 with the addition of multiple layers. (Sponza)

Layers	3D Ray March	NC-DDA	C-DDA	Min Hi-Z	Min-Max Hi-Z
1	122.70	281.92	288.03	76.21	34.37
2	91.67	277.85	284.64	---	35.32
4	79.53	276.64	283.74	---	32.67
8	76.95	276.36	283.52	---	32.13

Table 4-26: Average traversal iterations in Figure 4-12 with the addition of multiple layers. (Bistro)

Layers	3D Ray March	NC-DDA	C-DDA	Min Hi-Z	Min-Max Hi-Z
1	129.10	250.37	255.52	68.68	33.95
2	104.99	246.05	251.31	---	27.64
4	92.47	244.29	249.77	---	25.82
8	84.98	242.83	248.81	---	25.47

Table 4-27: Standard deviations for the average traversal iterations in Table 4-25. (Sponza)

Layers	3D Ray March	NC-DDA	C-DDA	Min Hi-Z	Min-Max Hi-Z
1	139.21	153.93	153.07	116.88	20.38
2	107.23	154.16	153.54	---	24.55
4	86.05	154.05	153.55	---	17.52
8	80.23	154.00	153.51	---	16.57

Table 4-28: Standard deviations for the average traversal iterations in Table 4-26. (Bistro)

Layers	3D Ray March	NC-DDA	C-DDA	Min Hi-Z	Min-Max Hi-Z
1	145.28	167.39	167.32	111.10	33.00
2	128.03	168.21	168.15	---	21.64
4	117.45	168.54	168.55	---	17.89
8	109.02	168.09	168.32	---	16.68

have zero traversal steps, such as when rays face towards the camera, are not included in the averages. For the linear traversal schemes, the number of traversal iterations seems to decrease with the number of layers. The decrease may occur because there is a higher chance for an intersection to occur in a pixel when there are multiple layers. The average number of iterations for C-DDA and NC-DDA traversal schemes are similar and they do not change much with the layer count since these schemes are poor at traversing empty space. Due to its conservative nature, C-DDA requires slightly more iterations than NC-DDA. The hierarchical traversal schemes require fewer iterations because they efficiently skip empty space. Min-Max Hi-Z requires the smallest number of iterations.

The standard deviations corresponding to these averages are given in Table 4-27 and Table 4-28. The standard deviations were calculated with the population variance formula since the population of all pixels for the current frame is known. The standard deviations for the DDA schemes do not change much, while the standard deviations for the 3D ray march and the Min-Max Hi-Z become lower with the layer count.

Using an intersection scheme with infinite thickness removes the problem of occlusion during the traversal process but using an infinite thickness can result in holes in the reflection. Figure 4-15 and Figure 4-16 show the average number of traversal iterations

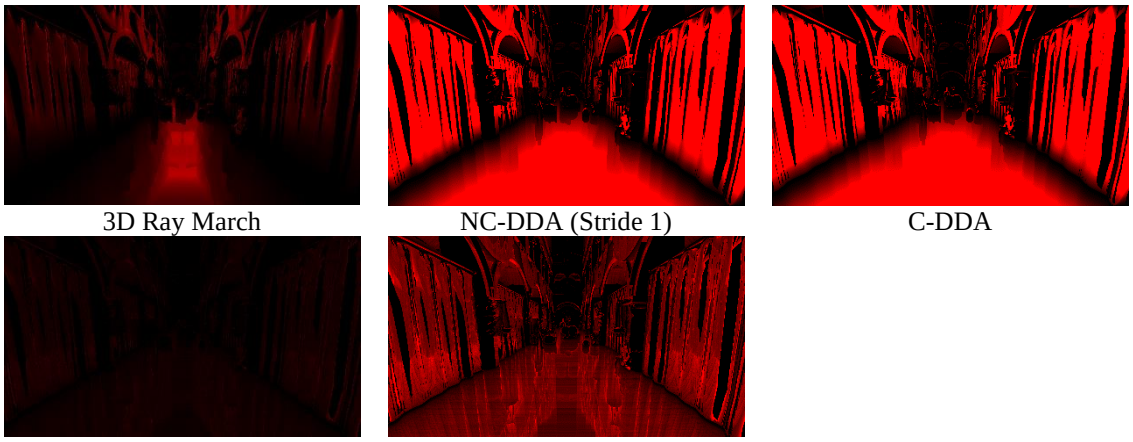


Figure 4-15: Visualization of single-layer traversal iterations for differing traversal schemes. Black is 0 iterations; Red is 400 unless specified. Black is better. (Infinite thickness) (original in colour)

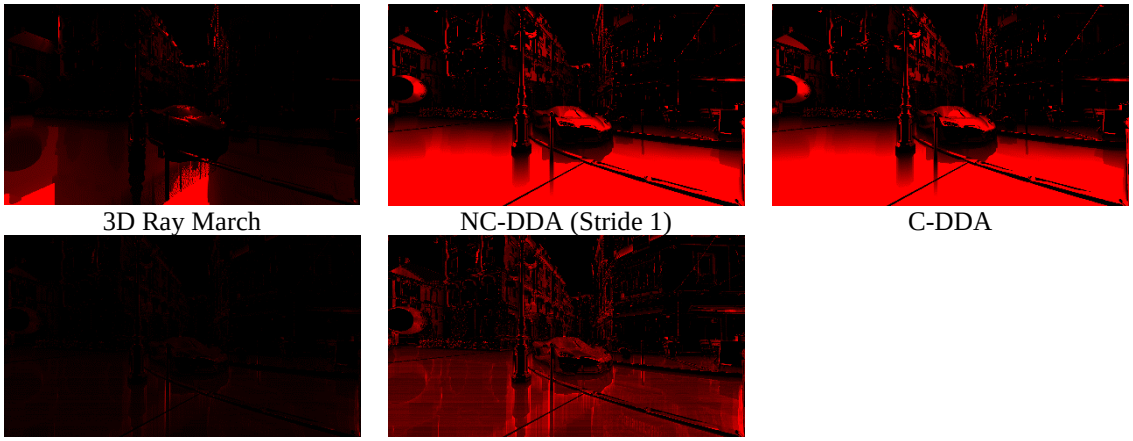


Figure 4-16: Visualization of single-layer traversal iterations for differing traversal schemes. Black is 0 iterations; Red is 400 unless specified. Black is better. (Infinite thickness) (original in colour)

when infinite thickness is used. Infinite thickness is not considered with multi-layer approaches, because traversal would become equivalent to using a single layer. With infinite thickness, occlusion regions with high iteration counts are removed. However, the DDA schemes still have high iteration counts from traversing open space.

The average number of iterations and standard deviations when using infinite thickness are shown in Table 4-29 and Table 4-30. Ray marching and Min Hi-Z have much lower numbers of iterations compared to when finite thickness is used, while the DDA

Table 4-29: Statistical analysis of traversal iterations in Figure 4-15. Uses infinite thickness intersection scheme. (Sponza) (1 Layer)

	3D Ray March	NC-DDA	C-DDA	Min Hi-Z	Min-Max Hi-Z
Average	59.59	262.84	269.80	29.74	---
Standard Dev	62.73	160.89	159.87	15.24	---

Table 4-30: Statistical analysis of traversal iterations in Figure 4-16. Uses infinite thickness intersection scheme. (Bistro) (1 Layer)

	3D Ray March	NC-DDA	C-DDA	Min Hi-Z	Min-Max Hi-Z
Average	57.46	217.87	223.94	24.23	---
Standard Dev	83.72	170.18	170.38	15.98	---

traversal schemes are not changed much. The results show trends similar to those shown in Figure 4-9 on page 81.

4.4 Discussion

Overall, Min-Max Hi-Z performs the best of the traversal schemes tested. Min-Max Hi-Z has the lowest traversal time in each of the test configurations, even when accounting for the additional time required in the *Build* stage. The sum of the average time for the *Build* and *Traverse* stages for Min-Max Hi-Z is less than the corresponding sum for any other traversal scheme in the same test configuration.

The linear traversal schemes scale poorly and have infeasible *Traverse* times with higher layer counts. The *Geometry Pass* and *Lighting Pass* increase by a considerable amount when multiple layers are used due to the added cost of generating and processing the linked lists of fragments. Thus, the use of multiple layers is very expensive. Min-Max Hi-Z scales well with the layer count since it efficiently skips empty space and handles occlusion.

The *Traverse* time for Min Hi-Z is close to the *Traverse* time of the linear schemes for a single layer. However, the Min Hi-Z traversal schemes performs poorly when rays become occluded. If the problem of occlusion is removed by using an infinite thickness during the traversal, Min Hi-Z performs better than the other traversal schemes. Simple 3D

ray marching also has a large improvement to the *Traverse* time with infinite thickness, but the quality of the reflection is less accurate than the Min Hi-Z.

The maximum iteration count for this evaluation was chosen with all of the traversal schemes in mind. The number was chosen to be high enough so the linear traversal schemes would produce an image with enough hit points to obtain a decent reflection. With a lower maximum iteration count, Min Hi-Z with a finite thickness would likely perform much better than with a higher maximum iteration count. Since Min Hi-Z reaches the maximum number of iterations in a large number of situations where the ray becomes occluded, a large amount of time is spent trying to obtain an unlikely intersection. If the maximum number of iterations was lowered, Min Hi-Z would likely outperform all of the linear traversal schemes in this evaluation.

The z-tests described in Section 3.4, with results in Appendix I, show that differences between the average *Traverse* times are statistically significant, except for a few cases. Of all the tests performed, 98% were statistically significant. Only three of the null hypotheses were not rejected in 136 z-tests. These three cases are displayed with bracket connections between their *Traverse* times in Table 4-2, Table 4-6, and Table 4-13. For reference, without outlier reduction, there are nine null hypotheses that were not able to be rejected (93% of tests show statistical significance). With or without outlier resolution, each rejected null hypothesis is related to traversal schemes that had nearly identical *Traverse* times such as NC-DDA and C-DDA. In cases where statistically significant differences were not found, the *Traverse* timing measurements differed by less than about 0.25ms. Min-Max Hi-Z was found to have a *Traverse* time that was consistently less than that of every other traversal scheme.

Chapter 5 Conclusions and Future Research

Section 5.1 provides a summary of the evaluation performed in this thesis. The conclusions of the evaluation are presented in Section 5.2 and remarks about future research are given in Section 5.3.

5.1 Summary

Screen Space Reflection(s) (SSR) is a group of techniques to render specular global illumination in the form of reflections using screen space data. Several algorithms exist to traverse rays in screen space, so a performance evaluation was conducted to compare the traversal time and iteration counts for each traversal scheme. Linear traversal schemes, such as 3D ray marching and DDAs, were considered as well as hierarchical traversal schemes, such as Min Hi-Z and Min-Max Hi-Z.

The traversal schemes were compared on two scene views with varying resolution, layer count and thickness parameters. A statistical two-sample z-test was performed for each pair of traversal schemes on each test configuration.

5.2 Conclusions

Despite the increased build time, the Min-Max Hi-Z traversal schemes consistently outperformed the other traversal schemes for every test configuration where it was evaluated. Min-Max Hi-Z has good scalability with respect to the number of depth layers used since it efficiently skips empty space and handles occlusion. The linear traversal schemes perform poorly when traversing empty space when no stride is used. Both the linear schemes and Min Hi-Z perform poorly when the ray becomes occluded. With these

schemes, occluded rays often cause the traversal to reach the maximum iteration count without finding an intersection.

When the intersection scheme uses an infinite thickness, Min Hi-Z outperforms the linear traversal schemes due to its efficient skipping of empty space. Although linear traversal schemes with large enough stride sizes could perhaps perform faster than Min Hi-Z, the quality of the reflections suffers. Min Hi-Z performs a conservative traversal that skips along the ray to at most the minimum depth at the current hierarchical level, so the quality is not reduced.

Ninety-eight percent of the statistical tests show that the differences in average traversal times between two traversal schemes on a given test configuration are statistically significant. In the remaining 2% of the cases, the traversal times were nearly identical.

5.3 Future Research

Several SSR techniques were covered in the evaluation of this thesis, but there are still some techniques that could be tested and parameters that could be varied. The evaluation also contains several limitations that could be removed in a future study. One topic that is not covered in this thesis is hardware accelerated ray tracing, which can be used with SSR. The remainder of this section expands on these ideas.

There are several techniques or parameters that could have been varied for the evaluation. One such parameter is the maximum number of traversal iterations. Varying this parameter could allow for Min Hi-Z to perform better using a constant thickness, while still producing the same output. A traversal scheme that was not tested was 2D screen space ray marching. A 2D ray marching scheme is similar to a 3D one, except the ray is sampled in screen space rather than in view space with a conversion to screen space. By using a

screen space stride, 2D ray marching is expected to be faster than the other linear traversal schemes, but similar to NC-DDA when striding is used. Hybrid techniques that use both a hierarchical component and a linear one could also be evaluated. McGuire et al. [40] describe a method similar to this. The lighting performed in this evaluation is limited and the lighting techniques used are perhaps outdated. Incorporating a more modern shading model would improve the quality of the results. The use of a more modern shading model would likely only affect the lighting pass and not the SSR traversal time, so this possibility was not explored in this thesis.

The evaluation in this thesis assumes that each surface is perfectly specular. Thus, rough surfaces cannot be properly rendered. Measuring the performance and quality of rendering glossy reflections using the various traversal schemes could be added to the evaluation. The approaches described in Section 2.6 could be used for this purpose. Each ray is also assumed to be reflective in the evaluation. Other visual effects such as refraction can be rendered using the multi-layer techniques presented in this thesis.

Recently, hardware accelerated support for ray tracing has been implemented into consumer level GPUs. Hardware such as the NVIDIA RTX can accelerate ray triangle intersection and Bounding Volume Hierarchy (BVH) traversal, while Graphics APIs such as Direct3D 12 and Vulcan have incorporated support for ray tracing directly into the APIs. Video games have begun to use this technology to improve visual effects with ray tracing.

The use of hardware accelerated ray tracing could potentially produce more accurate reflections than SSR can provide since more scene information is available in acceleration structures that are efficiently traversed. Despite the recent introduction of hardware accelerated ray tracing on consumer level GPUs, SSR will likely remain an important

group of techniques for rendering reflections in real time applications, such as video games, for some time. Ray tracing is still an expensive process and SSR techniques can still be used as a cheaper way for rendering reflections. SSR can also be used in conjunction with hardware accelerated ray tracing, since it can be used to increase the sample count of a raytraced image or to accelerate the traversal process [36]. Thomas et al. reduce the number of rays to be traced in a BVH by only traversing the BVH if an intersection is not quickly found in screen space [19]. Incorporating SSR with hardware accelerated ray tracing is an interesting topic for future research.

References

[1] T. Möller, E. Haines, N. Hoffman, A. Pesce, M. Iwanicki and S. Hillaire, Real-Time Rendering, 4th ed., A K Peters/CRC Press, 2018.

[2] C. Wyman, "Interactive Image-Space Refraction of Nearby Geometry," *Proc. 3rd Int'l Conf. Computer Graphics and Interactive Techniques in Australasia and South East Asia (GRAPHITE 05),* pp. 205-211, 2005.

[3] T. Sousa, N. Kasyan and N. Schulz, "Secrets of CryENGINE 3 Graphics Technology," *Advances in Real-Time Rendering in 3D Graphics and Games SIGGRAPH Course (SIGGRAPH 11),* 2011.

[4] T. Stachowiak and Y. Uludag, "Stochastic Screen-Space Reflections," *Advances in Real-Time Rendering in Games SIGGRAPH Course (SIGGRAPH 15),* 2015.

[5] B. Wronski, "'Assassin's Creed 4: Black Flag': Road to Next-Gen Graphics," *Proc. Game Developers Conf. (GDC 14),* 2014.

[6] M. Giacalone, "Screen Space Reflections in 'The Surge'," 20 June 2016; https://www.slideshare.net/MicheleGiacalone1/screen-space-reflections-in-the-surge.

[7] J. Mansouri, "Rendering 'Rainbow Six Siege'," *Proc. Game Developers Conf. (GDC 16),* 2016.

[8] M. McGuire and M. Mara, "Efficient GPU Screen-Space Ray Tracing," *J. of Computer Graphics Techniques,* vol. 3, no. 4, pp. 73-85, 2014.

[9] L. Szirmay-Kalos, B. Aszódi, I. Lazányi and M. Premecz, "Approximate Ray-Tracing on the GPU with Distance Impostors," *Computer Graphics Forum,* vol. 24, no. 3, pp. 695-704, 2005.

[10] T. Umenhoffer, G. Patow and L. Szirmay-Kalos, "Robust Multiple Specular Reflections and Refractions," *GPU Gems 3,* Addison-Wesley, 2007, pp. 387-407.

[11] A. Pranckevičius, "Compact Normal Storage for Small G-Buffers," 25 Mar. 2010; https://aras-p.info/texts/CompactNormalStorage.html.

[12] Z. H. Cigolle, S. Donow, D. Evangelakos, M. Mara, M. McGuire and Q. Meyer, "A Survey of Efficient Representations for Independent Unit Vectors," *J. of Computer Graphics Techniques,* vol. 3, no. 2, pp. 1-30, 2014.

[13] S. Hargreaves and M. Harris, "Deferred Shading. 6800 Leagues Under the Sea.," NVIDIA Corp., 2004; https://developer.nvidia.com/presentations-6800-leagues-under-sea.

[14] A. Lauritzen, “Deferred Rendering for Current and Future Rendering Pipelines,” *Beyond Programmable Shading SIGGRAPH Course (SIGGRAPH 10)*, 2010.

[15] G. Thomas, “Advancements in Tiled-Based Compute Rendering,” *Proc. Game Developers Conf. (GDC 15)*, 2015.

[16] E. Young, “Direct Compute Optimizations and Best Practices,” *GPU Technology Conf. (GTC 10)*, 2010.

[17] K.-L. Low, “Perspective-Correct Interpolation,” 2 Mar. 2002; https://www.comp.nus.edu.sg/~lowkl/publications/lowk_persp_interp_techrep.pdf.

[18] P. Ganestam and M. Doggett, “Real-time Multiply Recursive Reflections and Refractions Using Hybrid Rendering,” *The Visual Computer*, vol. 31, no. 10, pp. 1395-1403, 2015.

[19] T. Willberger, C. Musterle and S. Bergmann, “Deferred Hybrid Path Tracing,” *Ray Tracing Gems*, Apress, 2019, pp. 475-492.

[20] K. Vardis, A. A. Vasilakis and G. Papaioannou, “DIRT: Deferred Image-Based Ray Tracing,” *Proc. High Performance Graphics (HPG 16)*, pp. 63-73, 2016.

[21] B. Hopkins, “Screen Space Reflections in Unity 5,” 2 Feb. 2015; http://www.kode80.com/blog/2015/03/11/screen-space-reflections-in-unity-5/.

[22] J. Amanatides and A. Woo, “A Fast Voxel Traversal Algorithm for Ray Tracing,” *Proc. European Computer Graphics Conf. and Exhibition (EG 87)*, vol. 87, no. 3, pp. 3-10, 1987.

[23] Y. Uludag, “Hi-Z Screen-Space Cone-Traced Reflections,” *GPU Pro 5: Advanced Rendering Techniques*, A K Peters/CRC Press, 2014, pp. 149-192.

[24] S. Widmer, D. Pajak, A. Schulz, K. Pulli, J. Kautz, M. Goesele and D.Luebke, “An Adaptive Acceleration Structure for Screen-Space Ray Tracing,” *Proc. 7th Conf. on High-Performance Graphics (HPG 15)*, pp. 76-76, 2015.

[25] M. Valient, “Reflections and Volumetrics of 'Killzone Shadow Fall',” *Advances in Real-Time Rendering in Games SIGGRAPH Course (SIGGRAPH 14)*, 2014.

[26] A. Cichocki, “Optimized Pixel-Projected Reflections for Planar Reflectors,” *Advances in Real-Time Rendering in Games SIGGRAPH Course (SIGGRAPH 17)*, 2017.

[27] M. Drobot, “Rendering of 'Call of Duty: Infinite Warfare',” *Digital Dragons*, 2017.

[28] L. Bavoil, “Appendix: Optimizing Ray-Marching Loops. The Peak-Performance-Percentage Analysis Method for Optimizing Any GPU Workload,” 21 June 2019; https://devblogs.nvidia.com/the-peak-performance-analysis-method-for-optimizing-any-gpu-workload/.

[29] M. Mara, M. McGuire and D. Luebke, “Lighting With Deep G-Buffers: Single-Pass, Layered Depth Images with Minimum Separation Applied to Indirect Illumination,” *NVIDIA Technical Report,* 2013.

[30] C. Everitt, “Interactive Order-Independent Transparency,” NVIDIA Corp., 15 May 2001; https://www.nvidia.com/object/Interactive_Order_Transparency.html.

[31] J. Yang, J. Hensley, H. Grün and N. Thibieroz, “Real-Time Concurrent Linked List Construction on the GPU,” *Proc. 21st Eurographics Conf. on Rendering (EGSR 10),* vol. 29, no. 4, pp. 1297-1304, 2010.

[32] N. Hofmann, P. Bogendörfer, M. Stamminger and K. Selgrad, “Hierarchical Multi-Layer Screen-Space Ray Tracing,” *Proc. High-Performance Graphics (HPG 17),* no. 18, pp. 1-10, 2017.

[33] M. Salvi, “Pixel Synchronization: Solving Old Graphics Problems with New Data Structures,” *Advances in Real-Time Rendering in Games SIGGRAPH Course (SIGGRAPH 13),* 2013.

[34] L. Davies, “Rasterizer Order Views 101: a Primer,” Intel Corp., 5 Aug. 2015; https://software.intel.com/en-us/gamedev/articles/rasterizer-order-views-101-a-primer.

[35] K. Vardis, A. A. Vasilakis and G. Papaioannou, “A Multiview and Multilayer Approach for Interactive Ray Tracing,” *Proc. 20th ACM SIGGRAPH Symp. Interactive 3D Graphics and Games (I3D 16),* pp. 171-178, 2016.

[36] I. Llamas and E. Liu, “Ray Tracing in Games with NVIDIA RTX,” *Proc. Game Developers Conf. (GDC 18),* 2018.

[37] L. Hermanns and T. A. Franke, “Screen Space Cone Tracing for Glossy Reflections,” *ACM SIGGRAPH 2014 Posters,* no. 102, 2014.

[38] P. Sikachev and N. Longchamps, “Reflection System in 'Thief',” *Advances in Real-Time Rendering in Games SIGGRAPH Course (SIGGRAPH 14),* 2014.

[39] K. Cupisz and K. Engelstoft, “Lighting in Unity,” *Proc. Game Devlopers Conf. (GDC 15),* 2015.

[40] M. McGuire, M. Mara, D. Nowrouzezahrai and D. Luebke, “Real-Time Global Illumination using Precomputed Light Field Probes,” *Proc. 21st ACM SIGGRAPH Symp. Interactive 3D Graphics and Games (I3D 17),* no. 2, pp. 1-11, Feb. 2017.

[41] M. Sobek, “Real-Time Reflections in 'Mafia III' and Beyond,” *Proc. Game Developers Conf. (GDC 18),* 2018.

[42] M. McQuire, “Computer Graphics Archive,” July 2017; https://casual-effects.com/data.

[43] nickolasnm, “Mannequin figure,” TurboSquid, 12 Feb 2016; https://www.turbosquid.com/3d-models/free-mannequin-male-3d-model/1005602.

[44] W. Mendenhall, R. Beaver, B. Beaver and S. E. Ahmed, “Estimating the Difference Between Two Population Means,” *Introduction to Probability and Statistics*, 3rd Canadian ed., Nelson Education, 2013, pp. 330-333.

Appendices

The following contains HLSL code samples of the SSR traversal shader. Several traversal scheme variants are included. Note that some parts of the code are excluded for simplicity and that some constants (prefixed with "*cb*") are defined in constant buffers (not shown). Note that conditional compilation is used, and that the shader is meant to be compiled with different constants to generate different permutations.

Appendix A SSR Shader Main

```
/*Main Screen Space Reflections shader.*/
[numthreads(NUM_THREADS_X, NUM_THREADS_Y, 1)]
void main(uint3 dispatchThreadId : SV_DispatchThreadID)
{
    //Stop threads from reading out of bounds.
    if (any(dispatchThreadId.xy >= cbFrustumData.resolutionUint.xy))
        return;

    float3 pointSS0;
    pointSS0.xy = dispatchThreadId.xy + float2(0.5f, 0.5f);

    //Load Depth and Normals from the G-Buffer
    pointSS0.z = readDepthBuffer(depthSRV, pointSS0.xy).x;
    float3 normalVS = readAndDecodeNormal(dispatchThreadId.xy);

    //Convert depth to linear depth to allow for position reconstruction.
    float linearDepth = pointSS0.z;
#if defined(Z_BUFFER_IS_HYPERBOLIC)
    linearDepth = linearizeDepth(linearDepth, cbFrustumData.linearDepthConversion);
#endif

    //Calculate viewspace intersection point
    float3 originVS = screenSpaceToViewSpace(pointSS0.xy, linearDepth);

    //Calculate view direction.
    float3 viewRayVS = normalize(originVS);

    float3 directionVS = reflect(viewRayVS, normalVS);
    float3 endPointVS = originVS + directionVS * cbSsrData.maxRayDistance;

    //Calculate the Screen space points
    float4 pointHS1 = mul(float4(endPointVS, 1), cbFrustumData.vsToHSProjMatrix);
    float3 pointSS1;
```

Appendix A-1: SSR Compute Shader main HLSL code (Part 1)

```
    //Perspective divide. Now in range [0-resolutionPixels]
#if defined(Z_BUFFER_IS_HYPERBOLIC)
    pointSS1 = pointHS1.xyz / pointHS1.w;
#else
    pointSS1.xy = pointHS1.xy / pointHS1.w;
    pointSS1.z = pointHS1.w;//Linear Z.
#endif

    float3 hitPointSS = float3(-1.0f, -1.0f, 0.0f);
    float hitLayer = -1.0f;
    bool missed = true;
    float iterations = 0.0f;

    //Perform the Ray marching traversal using the Traversal sceheme that is defined.
    //Skip camera facing rays
    if (directionVS.z > 0)
    {
#if defined(TRAVERSAL_SCHEME_RAY_MARCH_3D)
        missed = rayMarch3D(originVS, directionVS, iterations, hitPointSS, hitLayer);
#elif defined(TRAVERSAL_SCHEME_NON_CONSERVATIVE)
        missed = nonConservativeDDATracing(pointSS0.xy, pointSS1.xy, pointHS1.w,
            originVS, endPointVS, iterations, hitPointSS, hitLayer);
#elif defined(TRAVERSAL_SCHEME_CONSERVATIVE)
        missed = conservativeDDATracing(pointSS0, pointSS1, iterations, hitPointSS,
            hitLayer);
#elif defined(TRAVERSAL_SCHEME_MIN_HI_Z) || defined(TRAVERSAL_SCHEME_MIN_MAX_HI_Z)
        missed = hiZTracing(pointSS0, pointSS1,  iterations, hitPointSS, hitLayer);
#else
    #error "A TRAVERSAL_SCHEME variant is not defined"
#endif
    }

    float alphaBlend = 0.0f;
    if (missed)
    {
        //Move hitpoint outside of view if it did not intersect.
        hitPointSS.xy = float2(-1.0f, -1.0f);
    }
    else
    {
        //Perform artifact resolution using data that wont be available in Apply Hits
        //Fade on ray distance, Z direction value and itertaions
        alphaBlend = calculateFadeAmount(originVS, directionVS.z, hitPointSS,
            iterations);
    }

    //Return the hitpoint
    hitPointUAV[dispatchThreadId.xy] = float4(hitPointSS.xy, hitLayer, alphaBlend);

    //Output hit point
#if defined(OUTPUT_DEBUG_DATA)
    debugIterationsUAV[dispatchThreadId.xy] = iterations;
#endif
}
```

Appendix A-2: SSR Compute Shader main HLSL code (Part 2)

Appendix B 3D Ray March

```
bool rayMarch3DNoBatch(float3 originVS, float3 dirVS,
    inout float iterations, out float3 hitPointSS, out float hitLayer)
{
    //Prevent immediate self intersection.
    originVS = originVS + dirVS * RAY_MARCH_ORIGIN_OFFSET_EPSILON;

    float sceneZMin = cbFrustumData.farZ;
    hitLayer = -1.0f;

    float deltaT = cbSsrData.maxRayDistance / cbSsrData.maxSteps;
    float t = 0.0f;
    float3 raySS = float3(-1, -1, cbFrustumData.farZ);
    bool outOfBounds = false;

    //Ray Marching loop
    [loop]
    for (float i = 0.0f; i < cbSsrData.maxSteps && !outOfBounds; i++)
    {
        iterations++;

        t += deltaT;
        float3 rayVS = originVS + dirVS * t;

        //Convert to homogeneous clip space.
        float4 rayHS = mul(float4(rayVS, 1), cbFrustumData.vsToHSProjMatrix);
        //Perform perspective divide to convert to pixel coordinates.
        raySS.xy = rayHS.xy / rayHS.w;
        raySS.z = rayHS.w;//Linear depth

        for (float layer = 0; layer < SSRT_LAYERS; layer++)
        {
            sceneZMin = readDepthBuffer(depthSRV, raySS.xy, layer).x;
            outOfBounds = (sceneZMin == 0.0f);

#if SSRT_LAYERS > 1
            //Multilayer early exit
            if (outOfBounds || sceneZMin >= getFarZDepth())
                break;
#endif

            //Get the linear depth.
#if defined(Z_BUFFER_IS_HYPERBOLIC)
            sceneZMin = linearizeDepth(sceneZMin, cbFrustumData.linearDepthConversion);
#endif

            float sceneZMax = sceneZMin + cbSsrData.zThickness;
```

Appendix B-1: 3D ray marching traversal scheme HLSL code. (Part 1)

```
            //If the depth buffer is intersected, break
            if (intersectsDepthBuffer(sceneZMin, sceneZMax, raySS.z, raySS.z))
            {
                hitLayer = layer;
                break;
            }
        }
        if (hitLayer >= 0)
            break;
    }

    //Build hitpoint
    hitPointSS = float3(raySS.xy, sceneZMin);

    return (raySS.z < sceneZMin) || (sceneZMin + cbSsrData.zThickness < raySS.z);
}
```

Appendix B-2: 3D ray marching traversal scheme HLSL code. (Part 2)

Appendix C Non-Conservative DDA

```
// Copyright (c) 2014, Morgan McGuire and Michael Mara
// All rights reserved.
//
// From McGuire and Mara, Efficient GPU Screen-Space Ray Tracing,
// Journal of Computer Graphics Techniques, 2014
//
// This software is open source under the "BSD 2-clause license":
//
//    Redistribution and use in source and binary forms, with or
//    without modification, are permitted provided that the following
//    conditions are met:
//
//    1. Redistributions of source code must retain the above
//    copyright notice, this list of conditions and the following
//    disclaimer.
//
//    2. Redistributions in binary form must reproduce the above
//    copyright notice, this list of conditions and the following
//    disclaimer in the documentation and/or other materials provided
//    with the distribution.
//
//    THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
//    CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES,
//    INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
//    MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
//    DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR
//    CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
//    SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
//    LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF
//    USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED
//    AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
//    LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING
//    IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF
//    THE POSSIBILITY OF SUCH DAMAGE.

/*This code has been modified from Efficient GPU Screen - Space Ray Tracing"
By Morgan McQuire and Michael Mara
It has been translated to HLSL and modified.*/

bool ncDDATraversalLoopCondition(float pSSx, float stepDirection, float endSS,
    float iterations, float maxSteps, bool outOfBounds, float sceneZMin, float rayZMin,
float rayZMax)
{
#if (SSRT_LAYERS > 1)
    return (pSSx * stepDirection <= endSS)//Break if traveled to end point distance.
        && (iterations < maxSteps)//Break if max steps reached.
        && !outOfBounds);//Break if sampling outside of the depth buffer.
#else
    float sceneZMax = sceneZMin + cbSsrData.zThickness;
    return ((pSSx * stepDirection) <= endSS)
        && (iterations < maxSteps)
        && !outOfBounds
        && !intersectsDepthBuffer(sceneZMin, sceneZMax,
            rayZMin, rayZMax); //Break if intersection found.
#endif
}
```

Appendix C-1: Non-Conservative DDA traversal scheme HLSL code. (Adapted from [8]) (Part 1)

```
bool nonConservativeDDATracing(float2 pSS0, float2 pSS1, float pHS1w, float3 originVS,
    float3 endPointVS, inout float iterations, out float3 hitPointSS, out float hitLayer)
{
    //Project the origin into screen-space pixel coodinates.
    //pHS1 is already given in input.
    float pHS0w = mul(float4(originVS, 1), cbFrustumData.vsToHSProjMatrix).w;

    //Perspective Division terms.
    float k0 = 1.0f / pHS0w;
    float k1 = 1.0f / pHS1w;

    //Convert to points which can by linearly interpolated in 2D.
    float3 q0 = originVS * k0;
    float3 q1 = endPointVS * k1;

    //Initialize to off-screen
    float2 hitPixel = float2(-1.0f, -1.0f);

    //If the line is degenerate, make it cover at least one pixel
    pSS1 += (float2)((distanceSquared(pSS0, pSS1) < 0.0001f) ? 0.01f : 0.0f);

    //Permute so that the primary iteration is in x to reduce branches later.
    bool permute = false;
    float2 delta = pSS1 - pSS0;
    if (abs(delta.x) < abs(delta.y))
    {
        //This is a more vertical line.
        //Create a permutation that swaps x and y in the output.
        permute = true;
        delta = delta.yx;
        pSS0 = pSS0.yx;
        pSS1 = pSS1.yx;
    }

    //From now on, x is the primary iteration direction and y is the second one.
    float stepDirection = sign(delta.x);
    float invDx = stepDirection / delta.x;
    float2 dPSS = float2(stepDirection, invDx * delta.y);

    //Track the derivatives of q and k.
    float3 dQ = (q1 - q0) * invDx;
    float dk = (k1 - k0) * invDx;

    //Scale Derivatives by pixel stride.
    dPSS *= cbSsrData.stride;
    dQ *= cbSsrData.stride;
    dk *= cbSsrData.stride;

    //Slide pSS from p0SS to p1SS, q from q0 to q1 and k from k0 to k1
    float2 pSS = pSS0;
    float3 q = q0;
    float k = k0;

    //pSS1.x is never modified after this point
    //so pre-scale it by the step direction for a signed comparison
    float endSS = pSS1.x * stepDirection;
```

Appendix C-2: Non-Conservative DDA traversal scheme HLSL code. (Adapted from [8]) (Part 2)

```
    //Move to the next pixel to prevent immediate self intersection.
    pSS += dPSS;
    q.z += dQ.z;
    k += dk;

    //Setup the ray depth interval.
    //Keep track of previous ray depth, so only 1 value is computed per iteration
    float rayZMax = originVS.z;
    float rayZMin = originVS.z;
    float sceneZMin = rayZMax + cbFrustumData.farZ;//Sufficiently far away
    float sceneZMax = sceneZMin;
    hitLayer = -1.0f;
    bool outOfBounds = false;

    //Loop until an intersection or the end of the ray has been reached.
    //Single Layer traversal bundles intersection check here as an optimization.
    [loop]
    for (pSS; ncDDATraversalLoopCondition(pSS.x, stepDirection, endSS,
            iterations, cbSsrData.maxSteps, outOfBounds, sceneZMin, rayZMin, rayZMax);
        pSS += dPSS, q.z += dQ.z, k += dk)
    {
        Iterations++;

        hitPixel = permute ? pSS.yx : pSS;

        //Use max from previous iteration as the current min.
        rayZMin = rayZMax;

        //Compute the maximum depth of the ray in this pixel.
        rayZMax = (q.z + dQ.z * 0.5f) / (k + dk * 0.5f);

        //Check each layer for intersection.
        for (float layer = 0; layer < SSRT_LAYERS; layer++)
        {
            //Get the viewspace depth from the depth buffer.
            sceneZMin = readDepthBuffer(depthSRV, hitPixel, layer).x;
            outOfBounds = (sceneZMin == 0.0f);
#if defined(Z_BUFFER_IS_HYPERBOLIC)
            sceneZMin = linearizeDepth(sceneZMin, cbFrustumData.linearDepthConversion);
#endif

            sceneZMax = sceneZMin + cbSsrData.zThickness;

#if (SSRT_LAYERS > 1)
            if (outOfBounds || sceneZMin >= cbFrustumData.farZ)
                break;
            //Break if intersected the depth buffer or sampled outside depth buffer.
            //Single layer will perform this check in the loop condition.
            if (intersectsDepthBuffer(sceneZMin, sceneZMax, rayZMin, rayZMax))
            {
                hitLayer = layer;
                break;
            }
#endif
        }
```

Appendix C-3: Non-Conservative DDA traversal scheme HLSL code. (Adapted from [8]) (Part 3)

```
        //Multilayer double loop exit condition.
#if (SSRT_LAYERS > 1)
        if (hitLayer >= 0)
            break;
#endif
    }

    //Calculate hipoint with viewspace depth
    hitPointSS = float3(hitPixel.xy, q.z * (1.0f / k));

    //Dicard hitpoints that are outside of the view intersecting depth buffer.
    //If the ray does not intersect the depth buffer, then discard the hitpoint.
    return ((any(hitPointSS.xy < 0) || any(hitPointSS.xy >= cbFrustumData.resolution))
        || !intersectsDepthBuffer(sceneZMin, sceneZMax, rayZMin, rayZMax));
}
```

Appendix C-4: Non-Conservative DDA traversal scheme HLSL code. (Adapted from [8]) (Part 4)

Appendix D Conservative DDA

```
//Calculate the maximum ray depth with linear interpolation.
float conservativeDepthInterp(float interpPoint, float interpVec, float a)
{
#if defined(Z_BUFFER_IS_HYPERBOLIC)
    return interpPoint + interpVec * a;
#else
    //Calculate the maximum ray depth with perspective correct interpolation.
    return 1.0f / (interpPoint + interpVec * a);
#endif
}

//Conservative DDA Tracing.
bool conservativeDDATracing(float3 pSS0, float3 pSS1, inout float iterations,
    out float3 hitPointSS, out float hitLayer)
{
    //Calculate Screen space direction vector.
    //From start point to maximum distance point.
    float3 vSS = pSS1 - pSS0;

    //Get the step direction for moving to the next pixel during traversal
    float3 stepSign;
    stepSign.x = (vSS.x >= 0) ? 1.0f : -1.0f;
    stepSign.y = (vSS.y >= 0) ? 1.0f : -1.0f;
    stepSign.z = (vSS.z >= 0) ? 1.0f : -1.0f;
    float2 step = saturate(stepSign.xy);

    //Ignore zero components.
    float3 vSSAbs = abs(vSS.xyz);
    vSS.x = (vSSAbs.x < DIR_ZERO_EPSILON_X) ? DIR_ZERO_EPSILON_X * stepSign.x : vSS.x;
    vSS.y = (vSSAbs.y < DIR_ZERO_EPSILON_Y) ? DIR_ZERO_EPSILON_Y * stepSign.y : vSS.y;
    vSS.z = (vSSAbs.z < DIR_ZERO_EPSILON_Z) ? DIR_ZERO_EPSILON_Z * stepSign.z : vSS.z;

    float2 vSSInv = 1.0f / vSS.xy;

    float2 pixel = floor(pSS0.xy);

    //Amount to increment for X and Y steps. (a componenet movement of 1).
    float2 tDelta = stepSign.xy * vSSInv.xy;
    //Initialize tMax to the distance to the next pixel boundary.
    float2 tMax = ((pixel + step) - pSS0.xy) * vSSInv.xy;

    //Go to the next pixel to avoid self intersection.
    float tCurrent;
    if (tMax.x < tMax.y)
    {
        tCurrent = tMax.x;
        tMax.x += tDelta.x;
        pixel.x += stepSign.x;
    }
    else
    {
        tCurrent = tMax.y;
        tMax.y += tDelta.y;
        pixel.y += stepSign.y;
    }
```

Appendix D-1: Conservative DDA traversal scheme HLSL code. (Part 1)

```
    //Setup some initial parameters for traversal.
#if defined(Z_BUFFER_IS_HYPERBOLIC)
    //Hyperbolic depth buffer interp parameters.
    //Can interpolate hyperbolic Z in screen space.
    //Depth interpolated like so: rayZMax = pSS.z + vSS.z * tCurrent
    const float interpPoint = pSS0.z;
    const float interpVec = vSS.z;
#else
    //Perspective Correct Interp parameters. Must interpolate with 1/Z
    //Depth interpolated like so: rayZMax = 1 / (pSS0InvZ+(pSS1InvZ-pSS0InvZ)*tCurrent)
    float pSS1SignZ = (pSS1.z >= 0) ? 1.0f : -1.0f;
    pSS1.z = (pSS1.z < DIR_ZERO_EPSILON_Z) ? DIR_ZERO_EPSILON_Z * pSS1SignZ : pSS1.z;
    const float interpPoint = 1.0f / pSS0.z;
    const float interpVec = (1.0f / pSS1.z) - interpPoint;
#endif
    float sceneZMin = getFarZDepth();
    float sceneZMax = getFarZDepth();
    float2 currPixel = float2(-1, -1);;
    hitLayer = -1.0f;

    //Setup ray depth interval.
    float rayZMin = pSS0.z;
    float rayZMax = rayZMin;

    //Perform the conservative ray march process.
    //The depth buffer intersection test could be bundled with loop condition here,
    //like with the NCDDA version.
    bool outOfBounds = false;
    [loop]
    while (tCurrent < 1.0f && !outOfBounds && iterations < cbSsrData.maxSteps)
    {
        rayZMin = rayZMax;

        iterations++;
        currPixel = pixel;

        //Step to next pixel based on the closer X or Y boundary.
        if (tMax.x < tMax.y)
        {
            tCurrent = tMax.x;
            tMax.x += tDelta.x;
            pixel.x += stepSign.x;
        }
        else
        {
            tCurrent = tMax.y;
            tMax.y += tDelta.y;
            pixel.y += stepSign.y;
        }

        for (float layer = 0; layer < SSRT_LAYERS; layer++)
        {
            //Get the minimum depth plane from the depth buffer.
            sceneZMin = readDepthBuffer(depthSRV, currPixel, layer).x;
            outOfBounds = (sceneZMin == 0.0f);
```

Appendix D-2: Conservative DDA traversal scheme HLSL code. (Part 2)

```
            //Calculate max scene depth with thickness.
#if defined(Z_BUFFER_IS_HYPERBOLIC)
            float sceneZMinLinear = linearizeDepth(sceneZMin,
                cbFrustumData.linearDepthConversion);
            sceneZMax = hyperbolizeDepth(sceneZMinLinear + cbSsrData.zThickness,
                cbFrustumData.linearDepthConversion);
#else
            sceneZMax = sceneZMin + cbSsrData.zThickness;
#endif

            //Calculate the maximum ray depth with linear interpolation.
            rayZMax = conservativeDepthInterp(interpPoint, interpVec, tCurrent);

#if SSRT_LAYERS > 1
            //Multilayer early exit
            if (outOfBounds || sceneZMin >= getFarZDepth())
                break;
#endif

            //Break if ray intersects depth buffer.
            if (intersectsDepthBuffer(sceneZMin, sceneZMax, rayZMin, rayZMax))
            {
                hitLayer = layer;
                break;
            }
        }
        if (hitLayer >= 0)
            break;
    }

    currPixel += float2(0.5f, 0.5f);
    hitPointSS = float3(currPixel, sceneZMin);

    //Dicard hitpoints that are outside of the view.
    //Discard points that do not intersect depth buffer
    return ((any(hitPointSS.xy < 0)
        || any(hitPointSS.xy >= cbFrustumData.resolution.xy))
        || !intersectsDepthBuffer(sceneZMin, sceneZMax, rayZMin, rayZMax));
}
```

Appendix D-3: Conservative DDA traversal scheme HLSL code. (Part 3)

Appendix E Hi-Z Setup

```
//Based on "Hi-Z Screen Space Cone Tracing" by Yasin Uludag (GPU Pro 5).
bool hiZTracing(float3 pSS0, float3 pSS1, inout float iterations,
    out float3 hitPointSS, out float hitLayer)
{
    //Map Screen space points to (UV coordinate, depth)
    //Map to range [0, 1] of the HiZ Reslution (expanded to nearest power of 2).
    pSS0.xy *= cbSsrData.invHiZResolution;
    pSS1.xy *= cbSsrData.invHiZResolution;

    //Calculate Screen space direction vector
    float3 vSS = pSS1 - pSS0;

    //Get the step direction and a small offset to enter the next pixel during traversal
    float3 stepSign;
    stepSign.x = (vSS.x >= 0) ? 1.0f : -1.0f;
    stepSign.y = (vSS.y >= 0) ? 1.0f : -1.0f;
    stepSign.z = (vSS.z >= 0) ? 1.0f : -1.0f;
    float2 stepOffset = stepSign.xy * (HIZ_STEP_EPSILON * cbSsrData.invHiZResolution);

    //Ignore zero components to avoid divide by zero.
    float3 vSSAbs = abs(vSS);
    vSS.x = (vSSAbs.x < DIR_ZERO_EPSILON_X) ? DIR_ZERO_EPSILON_X * stepSign.x : vSS.x;
    vSS.y = (vSSAbs.y < DIR_ZERO_EPSILON_Y) ? DIR_ZERO_EPSILON_Y * stepSign.y : vSS.y;
    vSS.z = (vSSAbs.z < DIR_ZERO_EPSILON_Z) ? DIR_ZERO_EPSILON_Z * stepSign.z : vSS.z;

    //Convert step to be from 0 to 1 instead of -1 to 1.
    float2 step = saturate(stepSign.xy);

    float3 vSSInv = 1.0f / vSS;

    //Calculate linear depth interpolation parameters. Only used for linear depth.
    float pSS0InvZ = 1.0f / pSS0.z;
    float pSS1InvZ = 1.0f / pSS1.z;
    //For Perspective Correct interpolation of Linear Depth.
    //Interpolated like this: Z = 1 / (interpPoint + interpVec * t)
    float interpPoint = pSS0InvZ;
    float interpVec = pSS1InvZ - pSS0InvZ;
    //For calculating parametric ray distance t for a given linear ray depth.
    //The equation Z = 1 / (interpPoint + interpVec * t) solved for t.
    float calcT0 = -pSS0InvZ;
    float calcT1 = 1.0f / (pSS1InvZ - pSS0InvZ);

    //Go to the next pixel to avoid self intersection.
    const float2 startingRayPixel = getHiZPixel(pSS0.xy, cbSsrData.hiZResolution);
    const float2 tStartPixelXY = ((startingRayPixel + step)
        / cbSsrData.hiZResolution + stepOffset - pSS0.xy) * vSSInv.xy;
    float tParameter = min(tStartPixelXY.x, tStartPixelXY.y);
    float2 tSceneZMinMax = float2(1.0f, 0.0f);
    float mipLevel = cbSsrData.hiZStartMipLevel;
    hitLayer = -1.0f;
```

Appendix E-1: Hi-Z traversal scheme setup HLSL code. (Adapted from [23]) (Part 1)

```
    //Perform the HiZ traversal Loop.
#if defined(TRAVERSAL_SCHEME_MIN_MAX_HI_Z)
    minMaxHiZTraversalLoop(step, stepOffset, pSS0, vSS, vSSInv, calcT0, calcT1,
        mipLevel, iterations, tParameter, tSceneZMinMax, hitLayer);
#else
    minHiZTraversalLoop(step, stepOffset, pSS0, vSS, vSSInv, calcT0, calcT1,
        mipLevel, iterations, tParameter, tSceneZMinMax, hitLayer);
#endif

    //Calculate the screenspace hitpoint.
#if defined(Z_BUFFER_IS_HYPERBOLIC)
    hitPointSS = float3(pSS0 + vSS * tParameter);
#else
    hitPointSS.xy = pSS0.xy + vSS.xy * tParameter;
    hitPointSS.z = 1.0f / (interpPoint + interpVec * tParameter);
#endif

    //Map hitpoint to range of resolution. (remove the padded power of 2 resolution)
    hitPointSS.xy *= cbSsrData.hiZResolution;

    return ((mipLevel != -1.0f) //If not on lowest level, missed
        || ((tParameter < tSceneZMinMax.x || tParameter > tSceneZMinMax.y)));
}
```

Appendix E-2: Hi-Z traversal scheme setup HLSL code. (Adapted from [23]) (Part 2)

Appendix F Min Hi-Z (Single Layer Only)

```
//Based on "Hi-Z Screen Space Cone Tracing" by Yasin Uludag (GPU Pro 5).

float2 getHiZPixel(float2 ray, float2 levelSize)
{
    return floor(ray * levelSize);
}

float2 getHiZLevelSize(float mipLevel, float2 rootLevelSize)
{
    return floor(rootLevelSize / min(exp2(mipLevel), rootLevelSize));
}

//Based on "Hi-Z Screen Space Cone Tracing" by Yasin Uludag (GPU Pro 5).
void minHiZTraversalLoop(float2 step, float2 stepOffset,
    float3 pSS0, float3 vSS, float3 vSSInv, float calcT0, float calcT1,
    inout float mipLevel, inout float iterations, inout float tParameter,
    inout float2 tSceneZMinMax, inout float hitLayer)
{
    [loop]
    while (mipLevel >= 0.0f && iterations < cbSsrData.maxSteps && tParameter <= 1.0f)
    {
        iterations++;

        const float2 maxRayPointXY = pSS0.xy + vSS.xy * tParameter;

        const float2 levelSize = getHiZLevelSize(mipLevel, cbSsrData.hiZResolution);
        const float2 pixel = getHiZPixel(maxRayPointXY, levelSize);

        //Get the min depth plane
        float2 sceneZMinMax;
        sceneZMinMax.x = readDepthBuffer(depthSRV, pixel, 0, mipLevel).x;
        if (sceneZMinMax.x == 0.0f)
            sceneZMinMax.x = getFarZDepth();

        //Parametric Distance to pixel edge.
        const float2 tPixelXY = ((pixel + step)
            / levelSize + stepOffset - pSS0.xy) * vSSInv.xy;
        const float tPixelEdge = min(tPixelXY.x, tPixelXY.y);

        //Parametric Distance to Depth Plane
#if defined(Z_BUFFER_IS_HYPERBOLIC)
        tSceneZMinMax.x = (sceneZMinMax.x - pSS0.z) * vSSInv.z;
#else
        tSceneZMinMax.x = (1.0f / sceneZMinMax.x + calcT0) * calcT1;
#endif

        //If ray intersects extents of the scene.
        mipLevel--;//Go down a miplevel
        if (tSceneZMinMax.x <= tPixelEdge)
        {
            tParameter = max(tParameter, tSceneZMinMax.x);

            //If at lowest resolution, also check if within thickness.
            if (mipLevel == -1.0f)//This is actually mip level 0.
            {
```

Appendix F-1: Min Hi-Z traversal scheme loop HLSL code. (Adapted from [23]) (Part 1)

```
                //Get Parametric distance to Thickness offset plane
#if defined(Z_BUFFER_IS_HYPERBOLIC)
                float sceneZMinLinear = linearizeDepth(sceneZMinMax.x,
                    cbFrustumData.linearDepthConversion);
                sceneZMinMax.y = hyperbolizeDepth(sceneZMinLinear
                    + cbSsrData.zThickness, cbFrustumData.linearDepthConversion);
                tSceneZMinMax.y = (sceneZMinMax.y - pSS0.z) * vSSInv.z;
#else
                sceneZMinMax.y = sceneZMinMax.x + cbSsrData.zThickness;
                tSceneZMinMax.y = (1.0f / sceneZMinMax.y + calcT0) * calcT1;
#endif

                //If the ray is not within thickness
                if (tParameter > tSceneZMinMax.y)
                {
                    tParameter = tPixelEdge;

                    //Go Up a miplevel
                    //Go up 2 levels to account for optimization of always decrementing
                    mipLevel = min(cbSsrData.hiZMaxMipLevel, mipLevel + 2.0f);
                }
            }
        }
        else
        {
            tParameter = tPixelEdge;

            //Go Up a miplevel
            //Go up 2 levels to account for optimization of always decrementing mipLevel
            mipLevel = min(cbSsrData.hiZMaxMipLevel, mipLevel + 2.0f);
        }
    }
}
```

Appendix F-2: Min Hi-Z traversal scheme loop HLSL code. (Adapted from [23]) (Part 2)

Appendix G Min-Max Hi-Z

```
//Based on "Hi-Z Screen Space Cone Tracing" by Yasin Uludag (GPU Pro 5).
void minMaxHiZTraversalLoop(float2 step, float2 stepOffset,
    float3 pSS0, float3 vSS, float3 vSSInv, float calcT0, float calcT1,
    inout float mipLevel, inout float iterations, inout float tParameter,
    inout float2 tSceneZMinMax, inout float hitLayer)
{
    //Perform HiZ Traversal
    [loop]
    while (mipLevel >= 0.0f && iterations < cbSsrData.maxSteps && tParameter <= 1.0f)
    {
        iterations++;

        const float2 maxRayPointXY = pSS0.xy + vSS.xy * tParameter;

        const float2 levelSize = getHiZLevelSize(mipLevel, cbSsrData.hiZResolution);
        const float2 pixel = getHiZPixel(maxRayPointXY, levelSize);

        //Parametric Distance to pixel edge.
        const float2 tPixelXY = ((pixel + step)
            / levelSize + stepOffset - pSS0.xy) * vSSInv.xy;
        const float tPixelEdge = min(tPixelXY.x, tPixelXY.y);

        float layer = 0;
        for (layer; layer < SSRT_LAYERS; layer++)
        {
            //Get the min depth plane
            float2 sceneZMinMax = readDepthBuffer(depthSRV, pixel, layer, mipLevel);
            //If sampled out of bounds
            if (sceneZMinMax.y == 0.0f)
                sceneZMinMax.xy = float2(getFarZDepth(), 0.0f);

#if (SSRT_LAYERS > 1)
            //Multilayer Early exit
            if (sceneZMinMax.x >= getFarZDepth())
            {
                layer = SSRT_LAYERS;
                break;
            }
#endif

            //Parametric Distance to Depth Planes
#if defined(Z_BUFFER_IS_HYPERBOLIC)
            tSceneZMinMax = (sceneZMinMax.xy - pSS0.z) * vSSInv.z;
#else
            tSceneZMinMax = (1.0f / sceneZMinMax + calcT0) * calcT1;
#endif
```

Appendix G-1: Min-Max Hi-Z traversal scheme loop HLSL code. (Adapted from [23]) (Part 1)

```
#if (SSRT_LAYERS == 1)
            //If ray intersects extents of the scene.
            mipLevel--;
            if (tSceneZMinMax.x <= tPixelEdge && tParameter <= tSceneZMinMax.y)
            {
                tParameter = max(tParameter, tSceneZMinMax.x);

                //Go down a miplevel. Happens above
            }
            else
            {
                tParameter = tPixelEdge;

                //Go Up a miplevel
                //Go up 2 levels to account for optimization of always decrementing mip
                mipLevel = min(cbSsrData.hiZMaxMipLevel, mipLevel + 2.0f);
            }
#else
            //If the ray intersects extents of the scene
            if (tSceneZMinMax.x <= tPixelEdge && tParameter <= tSceneZMinMax.y)
            {
                tParameter = max(tParameter, tSceneZMinMax.x);

                if (mipLevel == 0.0f)
                {
                    hitLayer = layer;
                }

                //Go down a miplevel
                mipLevel--;

                break;
            }
#endif
        }

#if (SSRT_LAYERS > 1)
        //If a hit was found, stop.
        if (hitLayer >= 0)
            break;

        //If ray did not intersect any Hi-Z level, go to pixel edge
        if (layer == SSRT_LAYERS)
        {
            tParameter = tPixelEdge;

            //Go Up a miplevel
            mipLevel = min(cbSsrData.hiZMaxMipLevel, mipLevel + 1.0f);
        }
#endif
    }
}
```

Appendix G-2: Min-Max Hi-Z traversal scheme loop HLSL code. (Adapted from [23]) (Part 2)

Appendix H Render Time Standard Deviations

Some of the numbers are too small to be displayed with 4 decimal points. Any calculations in this thesis with standard deviations used the calculated standard deviation or variance values. Some tables below have with a bracket connecting two values in the Traverse column, which indicates that there was not a significant difference in traversal time between the connected traversal schemes using the method described in Section 3.4.

Appendix H-1: Standard deviation of average GPU time (ms). (Sponza scene) (1920x1080 resolution) (1 Layer)

	Geometry Pass			**Lighting Pass**		**SSR**		
Traversal Scheme	**Draw**	**Sort**	**Build**	**Tile Bounds**	**Cull + Shade**	**Traverse**	**Apply Hits**	**Total**
3D Ray March	0.0041	---	0.0000	0.0043	0.0000	0.0253	0.0045	0.0326
NC-DDA	0.0031	---	0.0000	0.0048	0.0000	0.0051	0.0018	0.0140
C-DDA	0.0047	---	0.0000	0.0031	0.0050	0.0416	0.0018	0.0432
Min Hi-Z	0.0043	---	0.0018	0.0048	0.0050	0.2892	0.0038	0.2879
Min-Max Hi-Z	0.0831	---	0.0043	0.0045	0.0803	0.0050	0.0066	0.1189

Appendix H-2: Standard deviation of average GPU time (ms). (Sponza scene) (1920x1080 resolution) (2 Layer)

	Geometry Pass			**Lighting Pass**		**SSR**		
Traversal Scheme	**Draw**	**Sort**	**Build**	**Tile Bounds**	**Cull + Shade**	**Traverse**	**Apply Hits**	**Total**
3D Ray March	0.0799	0.0822	0.0018	0.0000	0.0031	0.0706	0.0038	0.1246
NC-DDA	0.0225	0.0160	0.0041	0.0000	0.0848	0.1998	0.0031	0.1906
C-DDA	0.0094	0.1107	0.0037	0.0000	0.0063	0.2063	0.0045	0.2091
Min-Max Hi-Z	0.0037	0.0089	0.0000	0.0000	0.0018	0.0117	0.0047	0.0154

Appendix H-3: Standard deviation of average GPU time (ms). (Sponza scene) (1920x1080 resolution) (4 Layer)

	Geometry Pass			**Lighting Pass**		**SSR**		
Traversal Scheme	**Draw**	**Sort**	**Build**	**Tile Bounds**	**Cull + Shade**	**Traverse**	**Apply Hits**	**Total**
3D Ray March	0.0128	0.0718	0.0050	0.0000	0.0056	0.2060	0.0089	0.1971
NC-DDA	0.1223	0.0132	0.0778	0.0000	0.0079	0.3749	0.0057	0.3586
C-DDA	0.0119	0.0100	0.0041	0.0018	0.0781	0.1886	0.0018	0.1846
Min-Max Hi-Z	0.0087	0.0087	0.0056	0.0031	0.0058	0.0126	0.0107	0.0216

Appendix H-4: Standard deviation of average GPU time (ms). (Sponza scene) (1920x1080 resolution) (8 Layer)

	Geometry Pass			**Lighting Pass**		**SSR**		
Traversal Scheme	**Draw**	**Sort**	**Build**	**Tile Bounds**	**Cull + Shade**	**Traverse**	**Apply Hits**	**Total**
3D Ray March	0.0141	0.1082	0.0781	0.0706	0.0730	0.3068	0.0092	0.3199
NC-DDA	0.0571	0.1027	0.1129	0.0056	0.0109	0.1910	0.0050	0.2298
C-DDA	0.0089	0.0771	0.0770	0.0996	0.0776	0.1761	0.0000	0.2296
Min-Max Hi-Z	0.0099	0.0877	0.1833	0.0043	0.0849	0.1637	0.0063	0.2303

Appendix H-5: Standard deviation of average GPU time (ms). (Bistro scene) (1920x1080resolution) (1 Layer)

	Geometry Pass			Lighting Pass		SSR		
Traversal Scheme	**Draw**	**Sort**	**Build**	**Tile Bounds**	**Cull + Shade**	**Traverse**	**Apply Hits**	**Total**
3D Ray March	0.3571	---	0.0000	0.0025	0.0000	0.1325	0.0879	0.3849
NC-DDA	0.0135	---	0.0000	0.0031	0.0000	0.0041	0.0000	0.0297
C-DDA	0.0535	---	0.0000	0.0035	0.0000	1.0110	0.0000	1.0076
Min Hi-Z	0.0038	---	0.0111	0.0035	0.0000	0.0094	0.0048	0.0299
Min-Max Hi-Z	0.0025	---	0.1709	0.0038	0.0852	0.0881	0.0000	0.2599

Appendix H-6: Standard deviation of average GPU time (ms). (Bistro scene) (1920x1080resolution) (2 Layer)

	Geometry Pass			Lighting Pass		SSR		
Traversal Scheme	**Draw**	**Sort**	**Build**	**Tile Bounds**	**Cull + Shade**	**Traverse**	**Apply Hits**	**Total**
3D Ray March	0.2137	0.0782	0.0050	0.0000	0.0000	0.0138	0.0000	0.2200
NC-DDA	0.1291	0.1265	0.0049	0.0000	0.0043	0.0163	0.0031	0.1860
C-DDA	0.1177	0.1489	0.0038	0.0018	0.1160	0.0104	0.0018	0.2134
Min-Max Hi-Z	0.1678	0.1193	0.0047	0.0000	0.0050	0.0048	0.0000	0.1976

Appendix H-7: Standard deviation of average GPU time (ms). (Bistro scene) (1920x1080 resolution) (4 Layer)

	Geometry Pass			Lighting Pass		SSR		
Traversal Scheme	**Draw**	**Sort**	**Build**	**Tile Bounds**	**Cull + Shade**	**Traverse**	**Apply Hits**	**Total**
3D Ray March	0.0958	0.0706	0.0048	0.0000	0.0037	0.0973	0.0000	0.1383
NC-DDA	0.1239	0.1811	0.0050	0.0037	0.0000	0.0281	0.0018	0.2247
C-DDA	0.1560	0.1142	0.0045	0.0000	0.0018	0.6083	0.0018	0.6243
Min-Max Hi-Z	0.1507	1.6062	0.0073	0.0025	0.0000	0.0057	0.0000	1.6010

Appendix H-8: Standard deviation of average GPU time (ms). (Bistro scene) (1920x1080 resolution) (8 Layer)

	Geometry Pass			Lighting Pass		SSR		
Traversal Scheme	**Draw**	**Sort**	**Build**	**Tile Bounds**	**Cull + Shade**	**Traverse**	**Apply Hits**	**Total**
3D Ray March	0.1428	0.0052	0.0032	0.0000	0.0106	0.1081	0.0041	0.1806
NC-DDA	0.1542	0.0112	0.0043	0.0053	0.0101	0.0751	0.0000	0.1536
C-DDA	0.2319	0.0065	0.0031	0.0046	0.0053	0.0506	0.0025	0.2394
Min-Max Hi-Z	0.0376	0.0057	0.0068	0.0064	0.0084	0.0068	0.0035	0.0400

Appendix H-9: Standard deviation of average GPU time (ms). (Sponza scene) (1280x720 resolution) (1 Layer)

	Geometry Pass			Lighting Pass		SSR		
Traversal Scheme	**Draw**	**Sort**	**Build**	**Tile Bounds**	**Cull + Shade**	**Traverse**	**Apply Hits**	**Total**
3D Ray March	0.0050	---	0.0000	0.0000	0.0000	0.0917	0.0018	0.0910
NC-DDA	0.0018	---	0.0000	0.0000	0.0000	0.0399	0.0035	0.0417
C-DDA	0.0037	---	0.0000	0.0000	0.0000	0.0248	0.0000	0.0267
Min Hi-Z	0.0057	---	0.0000	0.0000	0.0000	0.0257	0.0183	0.0316
Min-Max Hi-Z	0.0051	---	0.0000	0.0000	0.0000	0.0037	0.0018	0.0067

Appendix H-10: Standard deviation of average GPU time (ms). (Sponza scene) (1280x720 resolution) (2 Layer)

	Geometry Pass			Lighting Pass		SSR		
Traversal Scheme	**Draw**	**Sort**	**Build**	**Tile Bounds**	**Cull + Shade**	**Traverse**	**Apply Hits**	**Total**
3D Ray March	0.0032	0.0049	0.0000	0.0049	0.0031	0.0091	0.0043	0.0122
NC-DDA	0.0104	0.0059	0.0018	0.0038	0.0031	0.0188	0.0051	0.0208
C-DDA	0.0081	0.0065	0.0000	0.0035	0.0041	0.0259	0.0000	0.0305
Min-Max Hi-Z	0.0031	0.0063	0.0018	0.0025	0.0031	0.0043	0.0041	0.0072

Appendix H-11: Standard deviation of average GPU time (ms) (Sponza scene) (1280x720 resolution) (4 Layer)

	Geometry Pass			Lighting Pass		SSR		
Traversal Scheme	**Draw**	**Sort**	**Build**	**Tile Bounds**	**Cull + Shade**	**Traverse**	**Apply Hits**	**Total**
3D Ray March	0.0067	0.0053	0.0031	0.0000	0.0043	0.0172	0.0048	0.0221
NC-DDA	0.0105	0.0038	0.0031	0.0018	0.0025	0.0613	0.0031	0.0614
C-DDA	0.0766	0.0071	0.0038	0.0037	0.0045	0.1545	0.0031	0.1570
Min-Max Hi-Z	0.0038	0.0057	0.0052	0.0000	0.0018	0.0043	0.0000	0.0097

Appendix H-12: Standard deviation of average GPU time (ms). (Sponza scene) (1280x720 resolution) (8 Layer)

	Geometry Pass			Lighting Pass		SSR		
Traversal Scheme	**Draw**	**Sort**	**Build**	**Tile Bounds**	**Cull + Shade**	**Traverse**	**Apply Hits**	**Total**
3D Ray March	0.0083	0.0031	0.0041	0.0031	0.0077	0.0107	0.0045	0.0253
NC-DDA	0.0939	0.0031	0.0052	0.0052	0.0082	0.5488	0.0056	0.5489
C-DDA	0.0948	0.0018	0.0035	0.0053	0.0106	0.0524	0.0057	0.1116
Min-Max Hi-Z	0.0046	0.0000	0.0050	0.0025	0.0025	0.0588	0.0026	0.0576

Appendix H-13: Standard deviation of average GPU time (ms). (Bistro scene) (1280x720 resolution) (1 Layer)

	Geometry Pass			Lighting Pass		SSR		
Traversal Scheme	**Draw**	**Sort**	**Build**	**Tile Bounds**	**Cull + Shade**	**Traverse**	**Apply Hits**	**Total**
3D Ray March	0.1631	---	0.0000	0.0000	0.0955	0.1376	0.0018	0.2023
NC-DDA	0.0778	---	0.0000	0.0000	0.0659	0.0050	0.0000	0.1025
C-DDA	0.0120	---	0.0000	0.0000	0.0018	0.0048	0.0000	0.0207
Min Hi-Z	0.1991	---	0.0096	0.0000	0.0018	0.1918	0.0000	0.2712
Min-Max Hi-Z	0.0101	---	0.0000	0.0000	0.0051	0.0056	0.0018	0.0097

Appendix H-14: Standard deviation of average GPU time (ms). (Bistro scene) (1280x720 resolution) (2 Layer)

	Geometry Pass			Lighting Pass		SSR		
Traversal Scheme	**Draw**	**Sort**	**Build**	**Tile Bounds**	**Cull + Shade**	**Traverse**	**Apply Hits**	**Total**
3D Ray March	0.1375	0.0075	0.0048	0.0041	0.0766	0.0100	0.0000	0.1530
NC-DDA	0.1395	0.0746	0.0048	0.0032	0.0000	0.0769	0.0000	0.1703
C-DDA	0.0442	0.0072	0.0045	0.0051	0.0000	0.0070	0.0000	0.0477
Min-Max Hi-Z	0.0290	0.0061	0.0053	0.0043	0.0000	0.0018	0.0000	0.0263

Appendix H-15: Standard deviation of average GPU Time (ms). (Bistro scene) (1280x720 resolution) (4 Layer)

	Geometry Pass			Lighting Pass		SSR		
Traversal Scheme	**Draw**	**Sort**	**Build**	**Tile Bounds**	**Cull + Shade**	**Traverse**	**Apply Hits**	**Total**
3D Ray March	0.0358	0.1316	0.0031	0.0000	0.0723	0.1319	0.0000	0.2039
NC-DDA	0.1680	0.0087	0.0035	0.0000	0.0035	0.1366	0.0000	0.2081
C-DDA	0.1568	0.0730	0.0000	0.0018	0.0038	0.1778	0.0000	0.2358
Min-Max Hi-Z	0.0980	0.0103	0.0795	0.0749	0.0048	0.1247	0.0000	0.1993

Appendix H-16: Standard deviation of average GPU Time (ms). (Bistro scene) (1280x720 resolution) (8 Layer)

	Geometry Pass			Lighting Pass		SSR		
Traversal Scheme	**Draw**	**Sort**	**Build**	**Tile Bounds**	**Cull + Shade**	**Traverse**	**Apply Hits**	**Total**
3D Ray March	0.0464	0.0048	0.0037	0.0043	0.0055	0.0723	0.0049	0.0848
NC-DDA	0.0458	0.0051	0.0051	0.0048	0.0077	0.0311	0.0018	0.0570
C-DDA	0.1200	0.0057	0.0041	0.0038	0.0759	0.1121	0.0000	0.1706
Min-Max Hi-Z	0.2253	0.1132	0.1563	0.1498	0.0000	0.1561	0.0000	0.3419

Appendix H-17: Standard deviation of average GPU time (ms). (Sponza scene) (1920x1080 resolution) (1 Layer) (Uses Infinite Thickness)

	Geometry Pass			Lighting Pass		SSR		
Traversal Scheme	**Draw**	**Sort**	**Build**	**Tile Bounds**	**Cull + Shade**	**Traverse**	**Apply Hits**	**Total**
3D Ray March	0.0050	---	0.0000	0.0043	0.0018	0.1826	0.0018	0.2255
NC-DDA	0.0048	---	0.0000	0.0031	0.0051	0.0061	0.0041	0.0210
C-DDA	0.0055	---	0.0000	0.0047	0.0051	0.0979	0.0000	0.0998
Min Hi-Z	0.0043	---	0.0031	0.0043	0.0048	0.0040	0.0049	0.0160

Appendix H-18: Standard deviation of average GPU time (ms). (Bistro scene) (1920x1080 resolution) (1 Layer) (Uses Infinite Thickness)

	Geometry Pass			Lighting Pass		SSR		
Traversal Scheme	**Draw**	**Sort**	**Build**	**Tile Bounds**	**Cull + Shade**	**Traverse**	**Apply Hits**	**Total**
3D Ray March	0.0121	---	0.0000	0.0031	0.0000	0.0063	0.0025	0.0215
NC-DDA	0.0152	---	0.0000	0.0031	0.0000	0.0032	0.0049	0.0296
C-DDA	0.0041	---	0.0000	0.0018	0.0000	0.0059	0.0000	0.0072
Min-Max Hi-Z	0.0041	---	0.0101	0.0035	0.0000	0.0037	0.0018	0.0191

Appendix H-19: Standard deviation of average GPU Time (ms). (Sponza scene) (1280x720 resolution (1Layer) (Uses Infinite Thickness)

	Geometry Pass			Lighting Pass		SSR		
Traversal Scheme	**Draw**	**Sort**	**Build**	**Tile Bounds**	**Cull + Shade**	**Traverse**	**Apply Hits**	**Total**
3D Ray March	0.0051	---	0.0000	0.0000	0.0000	0.0079	0.0035	0.0119
NC-DDA	0.0037	---	0.0000	0.0000	0.0000	0.0051	0.0050	0.0065
C-DDA	0.0045	---	0.0000	0.0018	0.0000	0.0289	0.0018	0.0355
Min-Max Hi-Z	0.0045	---	0.0000	0.0000	0.0000	0.0043	0.0000	0.0049

Appendix H-20: Standard deviation of average GPU Time (ms). (Bistro scene) (1280x720 resolution) (1 Layer) (Uses Infinite Thickness)

	Geometry Pass			Lighting Pass		SSR		
Traversal Scheme	**Draw**	**Sort**	**Build**	**Tile Bounds**	**Cull + Shade**	**Traverse**	**Apply Hits**	**Total**
3D Ray March	0.0095	---	0.0000	0.0000	0.0000	0.0048	0.0000	0.0063
NC-DDA	0.0115	---	0.0000	0.0000	0.0025	0.0032	0.0000	0.0181
C-DDA	0.0163	---	0.0000	0.0000	0.0018	0.0776	0.0000	0.0828
Min-Max Hi-Z	0.0097	---	0.0101	0.0000	0.0000	0.0041	0.0000	0.0117

Appendix H-21: Standard deviation of average Traverse Time (ms) with varying Batch Size. Corresponds to Table 4-23. (1920x1080 resolution) (1 Layer)

	No Batch	**Batch-2**	**Batch-4**	**Batch-6**	**Batch-8**
Sponza	0.0253	0.1789	0.1509	0.1242	0.1213
Bistro	0.132	0.0785	0.0859	0.0104	0.0077

Appendix H-22: Standard deviation of average GPU Time (ms) with varying stride length. Corresponds to Table 4-24. (1920x1080 resolution) (1 Layer)

	Stride-1	**Stride-2**	**Stride-4**	**Stride-8**
Sponza	0.0051	0.1866	0.0895	0.0839
Bistro	0.0041	0.1155	0.0056	0.0077

Appendix I Traversal Time Z-Tests

To perform the two-sample z-test in a left one-tailed manner, Scheme 1 and Scheme 2 (and their respective data) should be swapped such that $\bar{x}_1 < \bar{x}_2$. This is not the case in the tables below. Instead the tables are displayed in a consistent ordering.

Appendix I-1: Z-tests for differing traversal schemes. (Sponza scene) (1920x1080 resolution) (1 Layer)

Scheme 1	$\bar{x}_1$ (ms)	s_1^2 (ms^2)	**Scheme 2**	$\bar{x}_2$ (ms)	s_2^2 (ms^2)	z	**Reject?**
3D Ray March	4.1230	0.0006	NC-DDA	3.1453	0.0000	-207.162	1
3D Ray March	4.1230	0.0006	C-DDA	4.2137	0.0017	-10.186	1
3D Ray March	4.1230	0.0006	Min Hi-Z	3.7517	0.0837	-7.005	1
3D Ray March	4.1230	0.0006	Min-Max Hi-Z	1.1160	0.0000	-637.606	1
NC-DDA	3.1453	0.0000	C-DDA	4.2137	0.0017	-139.475	1
NC-DDA	3.1453	0.0000	Min Hi-Z	3.7517	0.0837	-11.480	1
NC-DDA	3.1453	0.0000	Min-Max Hi-Z	1.1160	0.0000	-1562.960	1
C-DDA	4.2137	0.0017	Min Hi-Z	3.7517	0.0837	-8.659	1
C-DDA	4.2137	0.0017	Min-Max Hi-Z	1.1160	0.0000	-404.519	1
Min Hi-Z	3.7517	0.0837	Min-Max Hi-Z	1.1160	0.0000	-49.902	1

Appendix I-2: Z-tests for differing traversal schemes. (Sponza scene) (1920x1080 resolution) (2 Layer)

Scheme 1	$\bar{x}_1$ (ms)	s_1^2 (ms^2)	**Scheme 2**	$\bar{x}_2$ (ms)	s_2^2 (ms^2)	z	**Reject?**
3D Ray March	6.5783	0.0050	NC-DDA	7.6897	0.0399	-28.729	1
3D Ray March	6.5783	0.0050	C-DDA	7.7833	0.0426	-30.272	1
3D Ray March	6.5783	0.0050	Min-Max Hi-Z	2.2550	0.0001	-331.063	1
NC-DDA	7.6897	0.0399	C-DDA	7.7833	0.0426	-1.787	0
NC-DDA	7.6897	0.0399	Min-Max Hi-Z	2.2550	0.0001	-148.742	1
C-DDA	7.7833	0.0426	Min-Max Hi-Z	2.2550	0.0001	-146.552	1

Appendix I-3: Z-tests for differing traversal schemes. (Sponza scene) (1920x1080 resolution) (4 Layer)

Scheme 1	$\bar{x}_1$ (ms)	s_1^2 (ms^2)	**Scheme 2**	$\bar{x}_2$ (ms)	s_2^2 (ms^2)	z	**Reject?**
3D Ray March	11.5173	0.0425	NC-DDA	12.8307	0.1406	-16.815	1
3D Ray March	11.5173	0.0425	C-DDA	13.2757	0.0356	-34.478	1
3D Ray March	11.5173	0.0425	Min-Max Hi-Z	2.6170	0.0002	-236.150	1
NC-DDA	12.8307	0.1406	C-DDA	13.2757	0.0356	-5.808	1
NC-DDA	12.8307	0.1406	Min-Max Hi-Z	2.6170	0.0002	-149.127	1
C-DDA	13.2757	0.0356	Min-Max Hi-Z	2.6170	0.0002	-308.850	1

Appendix I-4: - tests for differing traversal schemes. (Sponza scene) (1920x1080 resolution) (8 Layer)

Scheme 1	$\bar{x}_1$ (ms)	s_1^2 (ms^2)	**Scheme 2**	$\bar{x}_2$ (ms)	s_2^2 (ms^2)	z	**Reject?**
3D Ray March	16.5553	0.0941	NC-DDA	14.7857	0.0365	-26.821	1
3D Ray March	16.5553	0.0941	C-DDA	15.2437	0.0310	-20.310	1
3D Ray March	16.5553	0.0941	Min-Max Hi-Z	2.4170	0.0268	-222.704	1
NC-DDA	14.7857	0.0365	C-DDA	15.2437	0.0310	-9.654	1
NC-DDA	14.7857	0.0365	Min-Max Hi-Z	2.4170	0.0268	-269.276	1
C-DDA	15.2437	0.0310	Min-Max Hi-Z	2.4170	0.0268	-292.157	1

Appendix I-5: Z-tests for differing traversal schemes. (Bistro scene) (1920x1080 resolution) (1 Layer)

Scheme 1	$\bar{x}_1\ (ms)$	$s_1^2\ (ms^2)$	**Scheme 2**	$\bar{x}_2\ (ms)$	$s_2^2\ (ms^2)$	z	**Reject?**
3D Ray March	4.2180	0.0176	NC-DDA	2.7120	0.0000	-62.204	1
3D Ray March	4.2180	0.0176	C-DDA	3.6020	1.0221	-3.309	1
3D Ray March	4.2180	0.0176	Min Hi-Z	2.7457	0.0001	-60.691	1
3D Ray March	4.2180	0.0176	Min-Max Hi-Z	1.2343	0.0078	-102.678	1
NC-DDA	2.7120	0.0000	C-DDA	3.6020	1.0221	-4.822	1
NC-DDA	2.7120	0.0000	Min Hi-Z	2.7457	0.0001	-18.080	1
NC-DDA	2.7120	0.0000	Min-Max Hi-Z	1.2343	0.0078	-91.759	1
C-DDA	3.6020	1.0221	Min Hi-Z	2.7457	0.0001	-4.639	1
C-DDA	3.6020	1.0221	Min-Max Hi-Z	1.2343	0.0078	-12.779	1
Min Hi-Z	2.7457	0.0001	Min-Max Hi-Z	1.2343	0.0078	-93.425	1

Appendix I-6: Z-tests for differing traversal schemes. (Bistro scene) (1920x1080 resolution) (2 Layer)

Scheme 1	$\bar{x}_1\ (ms)$	$s_1^2\ (ms^2)$	**Scheme 2**	$\bar{x}_2\ (ms)$	$s_2^2\ (ms^2)$	z	**Reject?**
3D Ray March	9.9877	0.0002	NC-DDA	6.5553	0.0003	-878.405	1
3D Ray March	9.9877	0.0002	C-DDA	6.2843	0.0001	-1172.890	1
3D Ray March	9.9877	0.0002	Min-Max Hi-Z	1.5967	0.0000	-3142.380	1
NC-DDA	6.5553	0.0003	C-DDA	6.2843	0.0001	-76.622	1
NC-DDA	6.5553	0.0003	Min-Max Hi-Z	1.5967	0.0000	-1594.560	1
C-DDA	6.2843	0.0001	Min-Max Hi-Z	1.5967	0.0000	-2242.000	1

Appendix I-7: Z-tests for differing traversal schemes. (Bistro scene) (1920x1080 resolution) (4 Layer)

Scheme 1	$\bar{x}_1\ (ms)$	$s_1^2\ (ms^2)$	**Scheme 2**	$\bar{x}_2\ (ms)$	$s_2^2\ (ms^2)$	z	**Reject?**
3D Ray March	19.2880	0.0095	NC-DDA	11.9667	0.0008	-395.888	1
3D Ray March	19.2880	0.0095	C-DDA	10.3703	0.3700	-79.293	1
3D Ray March	19.2880	0.0095	Min-Max Hi-Z	1.8757	0.0000	-978.282	1
NC-DDA	11.9667	0.0008	C-DDA	10.3703	0.3700	-14.359	1
NC-DDA	11.9667	0.0008	Min-Max Hi-Z	1.8757	0.0000	-1929.190	1
C-DDA	10.3703	0.3700	Min-Max Hi-Z	1.8757	0.0000	-76.489	1

Appendix I-8: Z-tests for differing traversal schemes. (Bistro scene) (1920x1080 resolution) (8 Layer)

Scheme 1	$\bar{x}_1\ (ms)$	$s_1^2\ (ms^2)$	**Scheme 2**	$\bar{x}_2\ (ms)$	$s_2^2\ (ms^2)$	z	**Reject?**
3D Ray March	28.2200	0.0117	NC-DDA	26.0113	0.0056	-91.916	1
3D Ray March	28.2200	0.0117	C-DDA	24.0683	0.0026	-190.499	1
3D Ray March	28.2200	0.0117	Min-Max Hi-Z	1.9923	0.0000	-1326.060	1
NC-DDA	26.0113	0.0056	C-DDA	24.0683	0.0026	-117.582	1
NC-DDA	26.0113	0.0056	Min-Max Hi-Z	1.9923	0.0000	-1745.780	1
C-DDA	24.0683	0.0026	Min-Max Hi-Z	1.9923	0.0000	-2368.940	1

Appendix I-9: Z-tests for differing traversal schemes. (Sponza scene) (1280x720 resolution) (1 Layer)

Scheme 1	$\bar{x}_1\ (ms)$	$s_1^2\ (ms^2)$	**Scheme 2**	$\bar{x}_2\ (ms)$	$s_2^2\ (ms^2)$	z	**Reject?**
3D Ray March	1.7630	0.0084	NC-DDA	1.2790	0.0016	-26.514	1
3D Ray March	1.7630	0.0084	C-DDA	1.7983	0.0006	-2.038	0
3D Ray March	1.7630	0.0084	Min Hi-Z	1.7100	0.0007	-3.049	1
3D Ray March	1.7630	0.0084	Min-Max Hi-Z	0.4900	0.0000	-76.009	1
NC-DDA	1.2790	0.0016	C-DDA	1.7983	0.0006	-60.509	1
NC-DDA	1.2790	0.0016	Min Hi-Z	1.7100	0.0007	-49.684	1
NC-DDA	1.2790	0.0016	Min-Max Hi-Z	0.4900	0.0000	-107.725	1
C-DDA	1.7983	0.0006	Min Hi-Z	1.7100	0.0007	-13.542	1
C-DDA	1.7983	0.0006	Min-Max Hi-Z	0.4900	0.0000	-285.919	1
Min Hi-Z	1.7100	0.0007	Min-Max Hi-Z	0.4900	0.0000	-257.035	1

Appendix I-10: Z-tests for differing traversal schemes. (Sponza scene) (1280x720 resolution) (2 Layer)

Scheme 1	$\bar{x}_1\ (ms)$	$s_1^2\ (ms^2)$	**Scheme 2**	$\bar{x}_2\ (ms)$	$s_2^2\ (ms^2)$	z	**Reject?**
3D Ray March	2.7307	0.0001	NC-DDA	3.1230	0.0004	-103.025	1
3D Ray March	2.7307	0.0001	C-DDA	3.2783	0.0007	-109.399	1
3D Ray March	2.7307	0.0001	Min-Max Hi-Z	0.9613	0.0000	-963.582	1
NC-DDA	3.1230	0.0004	C-DDA	3.2783	0.0007	-26.610	1
NC-DDA	3.1230	0.0004	Min-Max Hi-Z	0.9613	0.0000	-614.192	1
C-DDA	3.2783	0.0007	Min-Max Hi-Z	0.9613	0.0000	-483.693	1

Appendix I-11: Z-tests for differing traversal schemes. (Sponza scene) (1280x720 resolution) (4 Layer)

Scheme 1	$\bar{x}_1\ (ms)$	$s_1^2\ (ms^2)$	**Scheme 2**	$\bar{x}_2\ (ms)$	$s_2^2\ (ms^2)$	z	**Reject?**
3D Ray March	3.8557	0.0003	NC-DDA	5.4483	0.0038	-137.131	1
3D Ray March	3.8557	0.0003	C-DDA	5.9343	0.0239	-73.226	1
3D Ray March	3.8557	0.0003	Min-Max Hi-Z	1.1287	0.0000	-843.958	1
NC-DDA	5.4483	0.0038	C-DDA	5.9343	0.0239	-16.013	1
NC-DDA	5.4483	0.0038	Min-Max Hi-Z	1.1287	0.0000	-385.278	1
C-DDA	5.9343	0.0239	Min-Max Hi-Z	1.1287	0.0000	-170.263	1

Appendix I-12: Z-tests for differing traversal schemes. (Sponza scene) (1280x720 resolution) (8 Layer)

Scheme 1	$\bar{x}_1\ (ms)$	$s_1^2\ (ms^2)$	**Scheme 2**	$\bar{x}_2\ (ms)$	$s_2^2\ (ms^2)$	z	**Reject?**
3D Ray March	4.6463	0.0001	NC-DDA	6.6730	0.3012	-20.223	1
3D Ray March	4.6463	0.0001	C-DDA	6.9207	0.0027	-233.013	1
3D Ray March	4.6463	0.0001	Min-Max Hi-Z	1.0393	0.0035	-330.728	1
NC-DDA	6.6730	0.3012	C-DDA	6.9207	0.0027	-2.461	1
NC-DDA	6.6730	0.3012	Min-Max Hi-Z	1.0393	0.0035	-55.906	1
C-DDA	6.9207	0.0027	Min-Max Hi-Z	1.0393	0.0035	-409.141	1

Appendix I-13: Z-tests for differing traversal schemes. (Bistro scene) (1280x720 resolution) (1 Layer)

Scheme 1	$\bar{x}_1\ (ms)$	$s_1^2\ (ms^2)$	**Scheme 2**	$\bar{x}_2\ (ms)$	$s_2^2\ (ms^2)$	z	**Reject?**
3D Ray March	1.3747	0.0189	NC-DDA	1.1277	0.0000	-9.827	1
3D Ray March	1.3747	0.0189	C-DDA	1.4910	0.0000	-4.628	1
3D Ray March	1.3747	0.0189	Min Hi-Z	1.3650	0.0368	-0.224	0
3D Ray March	1.3747	0.0189	Min-Max Hi-Z	0.5563	0.0000	-32.552	1
NC-DDA	1.1277	0.0000	C-DDA	1.4910	0.0000	-285.740	1
NC-DDA	1.1277	0.0000	Min Hi-Z	1.3650	0.0368	-6.776	1
NC-DDA	1.1277	0.0000	Min-Max Hi-Z	0.5563	0.0000	-416.977	1
C-DDA	1.4910	0.0000	Min Hi-Z	1.3650	0.0368	-3.598	1
C-DDA	1.4910	0.0000	Min-Max Hi-Z	0.5563	0.0000	-696.512	1
Min Hi-Z	1.3650	0.0368	Min-Max Hi-Z	0.5563	0.0000	-23.086	1

Appendix I-14: Z-tests for differing traversal schemes. (Bistro scene) (1280x720 resolution) (2 Layer)

Scheme 1	$\bar{x}_1\ (ms)$	$s_1^2\ (ms^2)$	**Scheme 2**	$\bar{x}_2\ (ms)$	$s_2^2\ (ms^2)$	z	**Reject?**
3D Ray March	2.9460	0.0001	NC-DDA	2.7247	0.0059	-15.639	1
3D Ray March	2.9460	0.0001	C-DDA	2.6917	0.0000	-113.916	1
3D Ray March	2.9460	0.0001	Min-Max Hi-Z	0.7103	0.0000	-1200.610	1
NC-DDA	2.7247	0.0059	C-DDA	2.6917	0.0000	-2.342	1
NC-DDA	2.7247	0.0059	Min-Max Hi-Z	0.7103	0.0000	-143.493	1
C-DDA	2.6917	0.0000	Min-Max Hi-Z	0.7103	0.0000	-1502.280	1

Appendix I-15: Z-tests for differing traversal schemes. (Bistro scene) (1280x720 resolution) (4 Layer)

Scheme 1	$\bar{x}_1\ (ms)$	$s_1^2\ (ms^2)$	**Scheme 2**	$\bar{x}_2\ (ms)$	$s_2^2\ (ms^2)$	z	**Reject?**
3D Ray March	5.7133	0.0174	NC-DDA	4.6663	0.0187	-30.199	1
3D Ray March	5.7133	0.0174	C-DDA	4.4453	0.0316	-31.368	1
3D Ray March	5.7133	0.0174	Min-Max Hi-Z	0.8897	0.0155	-145.548	1
NC-DDA	4.6663	0.0187	C-DDA	4.4453	0.0316	-5.399	1
NC-DDA	4.6663	0.0187	Min-Max Hi-Z	0.8897	0.0155	-111.867	1
C-DDA	4.4453	0.0316	Min-Max Hi-Z	0.8897	0.0155	-89.686	1

Appendix I-16: Z-tests for differing traversal schemes. (Bistro scene) (1280x720 resolution) (8 Layer)

Scheme 1	$\bar{x}_1\ (ms)$	$s_1^2\ (ms^2)$	**Scheme 2**	$\bar{x}_2\ (ms)$	$s_2^2\ (ms^2)$	z	**Reject?**
3D Ray March	10.0457	0.0052	NC-DDA	9.5893	0.0010	-31.767	1
3D Ray March	10.0457	0.0052	C-DDA	9.2550	0.0126	-32.476	1
3D Ray March	10.0457	0.0052	Min-Max Hi-Z	1.0903	0.0244	-285.161	1
NC-DDA	9.5893	0.0010	C-DDA	9.2550	0.0126	-15.748	1
NC-DDA	9.5893	0.0010	Min-Max Hi-Z	1.0903	0.0244	-292.507	1
C-DDA	9.2550	0.0126	Min-Max Hi-Z	1.0903	0.0244	-232.742	1

Appendix I-17: Z-tests for differing traversal schemes. (Sponza scene) (1920x1080 resolution) (1 Layer) (Uses Infinite thickness)

Scheme 1	$\bar{x}_1\ (ms)$	$s_1^2\ (ms^2)$	Scheme 2	$\bar{x}_2\ (ms)$	$s_2^2\ (ms^2)$	z	Reject?
3D Ray March	1.6070	0.0334	NC-DDA	2.8037	0.0000	-35.866	1
3D Ray March	1.6070	0.0334	C-DDA	3.8607	0.0096	-59.570	1
3D Ray March	1.6070	0.0334	Min Hi-Z	0.9690	0.0000	-19.128	1
NC-DDA	2.8037	0.0000	C-DDA	3.8607	0.0096	-59.037	1
NC-DDA	2.8037	0.0000	Min Hi-Z	0.9690	0.0000	-1367.190	1
C-DDA	3.8607	0.0096	Min Hi-Z	0.9690	0.0000	-161.691	1

Appendix I-18: Z-tests for differing traversal schemes. (Bistro scene) (1920x1080 resolution) (1 Layer) (Uses Infinite thickness)

Scheme 1	$\bar{x}_1\ (ms)$	$s_1^2\ (ms^2)$	Scheme 2	$\bar{x}_2\ (ms)$	$s_2^2\ (ms^2)$	z	Reject?
3D Ray March	0.9687	0.0000	NC-DDA	2.1003	0.0000	-878.606	1
3D Ray March	0.9687	0.0000	C-DDA	2.7783	0.0000	-1147.610	1
3D Ray March	0.9687	0.0000	Min Hi-Z	0.8107	0.0000	-119.014	1
NC-DDA	2.1003	0.0000	C-DDA	2.7783	0.0000	-551.826	1
NC-DDA	2.1003	0.0000	Min Hi-Z	0.8107	0.0000	-1455.190	1
C-DDA	2.7783	0.0000	Min Hi-Z	0.8107	0.0000	-1549.280	1

Appendix I-19: Z-tests for differing traversal schemes. (Sponza scene) (1280x720 resolution) (1 Layer) (Uses Infinite thickness)

Scheme 1	$\bar{x}_1\ (ms)$	$s_1^2\ (ms^2)$	Scheme 2	$\bar{x}_2\ (ms)$	$s_2^2\ (ms^2)$	z	Reject?
3D Ray March	0.6217	0.0001	NC-DDA	1.1887	0.0000	-330.324	1
3D Ray March	0.6217	0.0001	C-DDA	1.5853	0.0008	-176.403	1
3D Ray March	0.6217	0.0001	Min Hi-Z	0.4113	0.0000	-127.616	1
NC-DDA	1.1887	0.0000	C-DDA	1.5853	0.0008	-74.156	1
NC-DDA	1.1887	0.0000	Min Hi-Z	0.4113	0.0000	-637.544	1
C-DDA	1.5853	0.0008	Min Hi-Z	0.4113	0.0000	-220.363	1

Appendix I-20: Z-tests for differing traversal schemes. (Bistro scene) (1280x720 resolution) (1 Layer) (Uses Infinite thickness)

Scheme 1	$\bar{x}_1\ (ms)$	$s_1^2\ (ms^2)$	Scheme 2	$\bar{x}_2\ (ms)$	$s_2^2\ (ms^2)$	z	Reject?
3D Ray March	0.4620	0.0000	NC-DDA	0.8503	0.0000	-366.515	1
3D Ray March	0.4620	0.0000	C-DDA	1.1803	0.0060	-50.623	1
3D Ray March	0.4620	0.0000	Min Hi-Z	0.3820	0.0000	-69.282	1
NC-DDA	0.8503	0.0000	C-DDA	1.1803	0.0060	-23.282	1
NC-DDA	0.8503	0.0000	Min Hi-Z	0.3820	0.0000	-495.676	1
C-DDA	1.1803	0.0060	Min Hi-Z	0.3820	0.0000	-56.293	1