2015

Real-time Shadows for Gigapixel Displacement Maps

Kevin Anthony Cherry
Louisiana State University and Agricultural and Mechanical College

REAL-TIME SHADOWS FOR GIGAPIXEL DISPLACEMENT MAPS

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science and Electrical Engineering

by
Kevin Cherry
B.S., Louisiana State University, 2008
M.S., Louisiana State University, 2011
December 2015

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Shadows portray helpful information in scenes. From a scientific visualization standpoint, they help to add data without unnecessary clutter. In video games they add realism and depth. In common graphics pipelines, due to the independent and parallel rendering of geometric primitives, shadows are difficult to achieve. Objects require knowledge of each other and therefore multiple renders are needed to collect the necessary data. The collection of this data comes with its own set of trade offs. Our research involves adding shadows into a lunar rendering framework developed by Dr. Robert Kooima [1]. The NASA-collected data contains a multi-gigapixel displacement map describing the lunar topology. This map does not fit entirely into main memory and therefore out-of-core paging is utilized to achieve real-time speeds. Current shadow techniques do not attempt to generate occluder data on such a scale, and therefore we have developed a novel approach to fit this situation. By using a chain of pre-processing steps, we analyze the structure of the displacement map and calculate horizon lines at each vertex. This information is saved into several images and used to generate shadows in a single pass, maintaining real-time speeds. The algorithm is even capable of generating soft shadows without extra information or loss of speed. We compare our algorithm with common approaches in the field as well as two forms of ground truth; one from ray tracing and the other from the gigapixel lunar texture data, showing real shadows at the time it was collected.

CHAPTER 1. INTRODUCTION

1.1 Introduction

There have been many different shadow algorithms created over the years, however most of them stem from two main approaches; that of shadow maps [2] and shadow volumes [3]. Standard shadow maps can only generate hard shadows, that is, fragments that are either completely lit or completely in shadow. The region completely in shadow is known as the umbra. The region partially shadowed that forms a gradient from shadowed to lit is known as the penumbra. More sophisticated shadow map algorithms are generally concerned with better utilization of map resolution [4] [5], the use of soft shadow techniques strictly to hide aliasing artifacts [6] [7], or, as in one of our techniques, Multi-Pass Gaussian [8], the use of soft shadow techniques for greater scene realism. Shadow volumes normally only cast hard shadows as well, however certain techniques including post processing can create soft shadows [9] [10] [11]. The main technique presented here uses neither shadow maps nor shadow volumes and instead relies on several pre processing steps that allow the algorithm to exploit the specific characteristics of the target scene's gigapixel displacement map. We then use the results of this data to generate shadows in real-time during the same pass needed to render the scene.

1.1.1 Why Are Shadows Important

We find shadows in simulations, 3D images, games and more. Designers and programmers utilize shadows primarily to portray extra information about a scene. Shadows help the user infer depth and order in a 3D virtual world. Shadows also help to create a stronger sense of realism and allow separate geometric objects to seem to belong to the same scene. Therefore shadows allow extra information to be displayed in a scene in a visually pleasing and non-cluttered way. In the field of information visualization, this quality is highly desired. For a lunar rendering, shadows help to get some sense of relative depth with respect to craters and raised surfaces. They may even be used to discover possible regions where the sun's rays can never hit and therefore substances such as ice could be found, as it would have never melted away.

1.1.2 Why Are Shadows Difficult to Generate

In general real-time shadows are hard to get right. Implementing basic techniques can be difficult and more advanced techniques even more so, however the real problem lies not only in implementation difficulty, but in the theoretical sense as well. This means it is difficult to even produce an algorithm or a concept that not only generates shadows, but does so in an accurate and visually pleasing way and at real-time speeds. The following sections explain why this is so.

1.1.2.1 In General Environments. There are many ways to develop a graphics pipeline. One must first consider what the primitives of the system are and then design around those. In a system such as ray tracing [12], the primitives are beams of light. In this system, shadows are calculated for free since any place in the scene that a light ray does not visit, will not be lit. While this is more accurate in accordance with how the real world works, most pipelines do not natively support this style of rendering due to its slow nature. OpenGL [13] and DirectX [14] chose to use geometry as its graphic primitive. This means that scenes are described and rendered with respect to the geometry within them. Each piece of geometry is sent through the pipeline separately and without knowledge of what has come before or what will come after it. This is done purposefully to better utilize the parallelism inherit in a GPU (which is massively greater than in a standard CPU). This approach works great for most visual effects and allows for very fast rendering. It also allows for SPMD style shaders to be written and applied to each vertex, face, and fragment. Problems

arise when one wants to program special effects such as reflections, refractions and shadows. In this style of architecture, lighting calculations can be done at each fragment and using the light source as a reference, we can tell if a fragment is lit or not. So each piece of geometry can be correctly shaded using the model of your choice. However, shadows are more than just fragments that face away from a light source. Those types of shadows are known as attached shadows. Cast shadows occur when a fragment would be lit if not for an occluding piece of geometry, i.e. part of another object or another part of its own geometry that is between that fragment and the light source. In a parallel pipeline where each geometric object has no knowledge of other geometry in the scene, this effect becomes difficult. Furthermore, one cannot even access other parts of the same geometric object. The most one can know about in a single thread's execution of a shader is the immediately adjacent vertices, which is not enough information to accurately ascertain even self occlusion from other parts of the same geometry. Some early 3D games used the concept of shadow blobs where darkened circles are placed on the ground under the occluding object. This is mainly found on the player of the game. Drop shadows can be used to match the geometry of the occluder and are flattened on the occluding surface. The main difference from this and a cast shadow is that drop shadows only exhibit affine transformations. They do not take into account perspective projections and can only scale uniformly. This difference allows drop shadows to be easier and faster to compute but causes them to be far less realistic than a true cast shadow.

1.1.2.2 Gigapixel Environment. Most of the problems of shadow generation become more difficult as the amount of data needing to be rendered increases. Because of this most shadow algorithms are only suited for environments of less than a million vertices. In our research the environment calls for over a billion vertices to fully render the single lunar object. This means that existing approaches would have to scale far beyond what they were designed to cover. The most successful approach is to exploit specific information about how this scene is rendered. In general when presented with a scene that a traditional shadow algorithm isn't suited for, seeing exactly how the scene is constructed and using that information can prove to be very advantageous. In our scene the information is provided from a giant displacement map on the order of several gigabytes of information. In this paper we will explain our novel approaches and how they compare against the traditional methods in both this gigapixel environment and other, more traditional environments.

1.1.3 Generic Shadow Generation Procedure

In general shadow algorithms can be divided into two main steps, gathering and querying. The gathering stage is normally in the form of rendering the scene to either a depth, color or stencil buffer. The way this data is collected and more importantly the way it is stored is crucial for speeding up the querying stage. For the gather stage, there are several trade offs. We need to consider in which space the information is obtained. In object space, we have complete geometric information but collecting that information can be slow and storing it can be costly. In image space, collecting is fast and storage can be bounded but information is not perfect. For the query stage, we must be able to quickly access the data we stored and use that to efficiently calculate whether a fragment is in shadow or not. One of the unique characteristics of our main algorithm is that it requires only a single pass. The gathering stage is completed as a series of pre-process phases and several images are generated as a result. This is done during construction time therefore the only stage needed for online rendering is the querying stage. This stage can be combined with the normal render so as to require no extra passes in the scene.

1.1.4 Shadow Problems

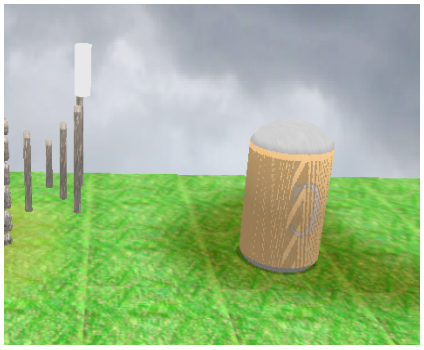
With any shadow algorithm comes trade-offs. One must be aware of the problems inherit in the algorithm and possible ways to mitigate each. One of the most common problems is that of aliasing [15].

1.1.4.1 Perspective Aliasing. Depth values follow a logarithmic scale meaning fragments close to camera have a larger range of values than those closer to the far clipping plane. This can be mitigated by using techniques such as Cascaded Shadow Maps [16], which allow for multiple shadow maps to have different resolutions more appropriate to the area of the viewing frustum they cover. Maps closer to the view's near clipping place require a higher resolution than those closer to the far clipping plane. An example of this perspective aliasing artifact can be seen in Figure 1.1.

An example of perspective aliasing. Greater map resolution or use of Cascaded Shadow Maps [16] would help reduce the effects and produce higher quality shadows.

Figure 1.1: Perspective Aliasing

1.1.4.2 Projective Aliasing. Projective aliasing occurs when light beams are near parallel to geometry faces (see Figure 1.2). From the point of view of the light source, a near parallel face can appear as a single fragment (or a very small number of fragments). When rendering from the camera, fragments on the affected surface are projected onto a small area in the shadow map and therefore all use only a tiny subset of the map. This leads several pixels to be incorrectly lit or shadowed. This can be made worse by either having the light rays be completely parallel to the surface, or if the viewing rays are perpendicular to the surface.



The shadowing on the orange bag is an example of projective aliasing. The light is pointing downward and light ray are mostly parallel to the side of the bag. This causes several shadow map values in that area to map to a limited number of pixels; in cases where the light rays are completely parallel, all shadow map values affected are mapped to a single pixel. Increasing map resolution can help, but won't solve the problem fully.

Figure 1.2: Projective Aliasing

1.1.4.3 Temporal Aliasing. When using Cascaded Shadow Maps [16], movement of the camera can cause map boundaries to shift and as a result, different resolution shadow maps to be used for a given pixel. When a shadow map is used on an area that has a resolution different from the shadow map previously used, a flicker can occur. This flicker is known as temporal aliasing.

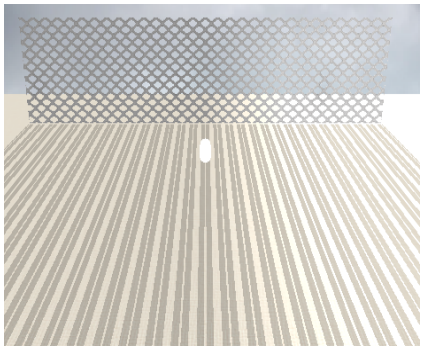
1.1.4.4 Acne/Moiré. Pixel depths are represented as floating point values and have a limited number of bits. In shadow maps after collecting these depths from the point of view of the light source, we compare them against the projected fragment's Z coordinate. When comparing two similar limited-precision floating point values, the comparison results may not always be predictable. Even though we could be looking at the same fragment projected now into light space as the one whose depth was collected in the shadow map during the initial rendering, the values may not be the same. This fragment, therefore may be considered to be in shadow yet the fragment(s) next to it may be considered lit even though there are no occluders present. This creates a visually disturbing pattern known as shadow acne and it is a form of moiré. An example can be seen in Figure 1.3. An easy way to fix moiré is to add a small bias to the pixel's depth. This can either be added when the shadow map is created, pushing fragments a little further from the light and therefore making them appear lit when viewed/compared from the camera's perspective; or the fragments projected value can have a bias subtracted from it before being compared to the shadow map, accomplishing the same result. This can be done either manually in the fragment shader, or by using OpenGL's polygon offset functionality.

An example of shadow acne also known as a moiré pattern. Due to limitations in floating point precision and a limited number of bits in the depth buffer, fragments which should be lit can sometimes incorrectly appear as in shadow. A bias must be used to prevent this.

Figure 1.3: Acne / Moiré

1.1.4.5 Dueling Frusta. The phenomena known as dueling frusta occurs when the light's frustum faces towards the camera frustum, thereby creating one frustum at a 180 degree angle of the other (see Figure 1.4). As stated earlier depth values follow a logarithmic scale and more precision is given to values closer to the near clipping plane. This means that the lowest precision part of the light's frustum is exactly where the highest precision part of the camera's frustum is. When viewing the scene from the camera, we want high precision shadow information by the near clipping plane and yet we receive low precision information instead and vice versa for the far clipping plane. This can lead to extreme aliasing.

1.1.4.6 Speed. Some algorithms are faster than others. Shadow maps are susceptible to aliasing problems but their speed can be controlled by buffer resolution. Shadow volumes, however, do not display any aliasing but are dependent upon scene complexity and as such, can become very slow in detailed scenes.



The light source (white cylinder) is perfectly facing the camera, making the viewing rays and light rays parallel but in opposite directions. The shadow map used here is at a resolution of 8192x8192 and yet the shadow produced is barely recognizable as coming from the chain-link fence.

Figure 1.4: Dueling Frusta

Our main algorithm has speed similar to shadow maps but contains no aliasing artifacts, thereby containing the benefits of both approaches without the inherit drawbacks of either.

1.1.5 Shadow Algorithmic Choices

When choosing or creating one's own shadow algorithm, it is important to know which choices are available. Each decision comes with its own set of trade-offs and one should be aware of these.

1.1.5.1 Which Space to Collect Data in. One of the main questions to consider when thinking about one's own shadow algorithm, is where to collect the data. Typically the choice is between collection in object space, where rendering primitives are generated and utilized, or in image space, where buffer resolution and uniform geometric distribution is key. In object space data is obtained without pixel discretization, thereby bypassing aliasing artifacts. One has perfect, continuous knowledge of the scene, but as such, one has to decide what is important and what isn't. Collecting too much data here can slow down the algorithm and increase memory footprint. One can also instruct the GPU to generate additional geometry to be used later on, but again this can slow down the algorithm. In image space the amount of information is upper bounded by the buffer size, so memory and time usage can be controlled. This makes gathering data computationally efficient and can lead to fast querying as well. This information however has been through the rasterizer and the pixel sampling that occurred to transfer the scene onto the image buffer has caused lost of scene data to occur. We lose data both due to the discretization process and by the fact that any occluded geometry is not represented. Therefore the information gathered is not perfect and is subject to aliasing. If we are gathering information using a depth buffer, any anti-aliasing techniques would not help since they would distort scene structure and lead to inaccurate information. In our approach there are two stages where data is collected, cache generation and horizon calculation. These steps will be discussed in greater length in a later section, but in general, cache generation collects its data in object space without the need to generate any additional geometry. The horizon calculation stage has three different methods, but the fastest and most accurate of these collects its data in image space.

1.1.5.2 How to Best Utilize Buffer Size when Collecting in Image Space. When one collects data in image space, another problem that often occurs is how to best use the pixel density of the buffer. Buffer size is bounded by hardware, so continually increasing the size is not an option. A better approach is to think of ways to better utilize the resolution given. One popular way of doing this is to have multiple buffers

each covering a different part of the viewing frustum, a technique known as Cascaded Shadow Maps [16]. This reduces the work each buffer must do and allows the resolution to cover a smaller area, which reduces aliasing by allowing more samples per region. Another approach is to redistribute the buffer samples such that areas with high geometric density receive more samples than other areas. This allows areas that need it to have a higher density of sampling over areas that don't need it as much. Both Light Space Perspective Shadow Maps [5] and Sample Distribution Shadow Maps [17] use this approach.

1.1.5.3 How to handle soft shadows. There are many decisions to be made to generate soft shadows. For example, if one is generating a blurred shadow by way of a low pass filter [18] (allowing low frequency data through while diluting high frequency data such as edges), the choice of kernel is important. A simple Gaussian kernel [19] works out nicely but does have parameters that must be set, such as its standard deviation. Setting these parameters to ideal values may prove to be non-trivial in some situations. Another decision is whether or not to make the algorithm exhibit signs of contact-hardening, whereby the shadow edges are sharper if the occluder is closer. Contact-hardening shadows are more realistic but could require more information to be gathered. The equation used to calculate the penumbra portion can vary depending on the factors one wants to include. One of the equations that can be used comes from Fernando [20] and is as follows:

$$\omega_{penumbra} = \frac{(d_{receiver} - d_{blocker}) * \omega_{light}}{d_{blocker}} \tag{1.1}$$

where ω is width and d is distance. Another equation that can be used to incorporate observer distance comes from Klein [21] and is as follows:

$$\omega_{penumbra} = \frac{(d_{receiver} - d_{blocker}) * \omega_{light}}{d_{blocker} * d_{observer}} \tag{1.2}$$

1.2 Related Works

1.2.1 Shadow Volumes

Shadow volumes [3] gather their data in object space and as such has perfect information about the scene. They operate by examining each edge in the scene to see if it is a silhouette edge. A silhouette edge is one in which the adjoining faces differ in their normal's orientation towards the light (e.g. one faces towards and the other away). For each silhouette edge found, two new vertices are created and translated away from the light towards infinity, along the vector formed by the light and silhouette edge's vertices. Now with two vertices from the silhouette edge and two new vertices created, we have a quad. Each silhouette edge forms its own quad and these quads together (from the same object) form a shadow volume. Any fragments inside this volume are blocked from the light by the object whom created the volume. This process needs to be performed for each piece of geometry that is a possible occluder, potentially creating lots of new geometry for the GPU to handle. Even though the user will never directly see the shadow volumes, the new vertices must be rendered and rasterized for the algorithm to work correctly. Therefore the main downfall of the algorithm is in the calculation and rendering of the shadow volumes whereby extra, albeit linear, amount of GPU work is done per face. This extra work made the algorithm impractical to be used in real-time applications [22] when it was first introduced. It wasn't until 1991 that Tim Heidmann came up with the idea of using stencil buffers to speed up the algorithm [23] and caused it to be possible to run in real-time. He used stencil buffers to count the number of times a shadow volume was entered or exited. Going from the camera in a straight line towards a location in the scene, we start the stencil buffer at a value of zero and

add one upon entering a shadow volume and subtract one upon exiting. If the final value upon arriving at the location is zero, then we have entered the same number of times as we have exited a volume, and that location is lit. If the final value is greater than zero, then we are inside one or more volumes and therefore are blocked from the light source. The following is an English prose description of the stencil buffer procedure created by Heidmann. We first render our scene without shadow volumes to the depth buffer.

Init

- Enable depth testing
- Enable writing to the depth buffer
- Render scene normally

Increment stencil buffer

- Enable stencil testing
- Clear the stencil buffer to all zeros
- Enable writing to stencil buffer
- Disable writing to depth and color buffers
- Enable back face culling
- Set stencil buffer to increment on fragments that pass the depth test
- Render shadow volumes (note: only rendering front faces)

Decrement stencil buffer

- Disable back face culling
- Enable front face culling
- Set stencil buffer to decrement on fragments that pass the depth test
- Render shadow volumes (note: only rendering back faces)

Render scene

- Enable writing to depth and color buffers
- Disable culling
- Disable writing to stencil buffer
- Render entire scene with ambient light only regardless of stencil value
- Render scene with full lighting only where stencil buffer bit is zero

Several improvements have been made to this basic form of the algorithm. These improvements either address accuracy concerns when the camera lies inside a shadow volume, or speed concerns. Even though stencil buffers provided a much needed speed boast to the procedure, there is still much room for improvement, and several researchers have come up with ways to dramatically decrease time needed for the algorithm.

Heidmann's approach was known as depth pass since it only alters the stencil buffer when a fragment passes the depth test. In his approach we are counting the shadow volumes in front of an object, and therefore only count shadow volumes that pass the depth test (our depth buffer already has the normal scene rendered to it). The problem with this occurs when the camera lies inside of a shadow volume. At this point

the stencil buffer count becomes incorrect. John Carmack in 2004 [24] noticed this and created his variation known as depth fail. In Carmack's approach we only count shadow volumes that fail the depth test, which means we are counting volumes behind the object. The count is altered such that back faces increment the count whereas front faces decrement it. This solves the problem of the camera's location but requires that a cap be added to the end of all shadow volumes.

1.2.2 Shadow Maps

Shadow Maps [2] is a technique created a year after shadow volumes in which the data in the gathering stage is collected in image space. The downside, of course, is the problems that come with collecting imperfect discretized data (that is data collected post-rasterization), however it outperforms shadow volumes with regards to speed. Depending upon the scene, aliasing artifacts (which are the result of collecting this imperfect data) can be mitigated such that the lost of information during collection is not that much of an issue. The following is a description of the procedure.

- Render scene from point of view of the light source into a depth buffer (shadow map)
- Render scene normally, and performs these operations on the GPU :
  - Project each fragment into light space
  - Use its light space position as texture coordinates into the shadow map
  - If fragment's distance from light is equal or less than shadow map value then
    - No other fragment is between it and the light source so it is lit
  - Else
    - Some other fragment is blocking it from the light and it is in shadow

Like shadow volumes, there have been numerous improvements to the basic shadow map algorithm since its initial inception. The following paragraphs comprise a list of some of those improvements.

1.2.2.1 Cascaded Shadow Maps.

Cascaded Shadow Maps [16] focuses on having multiple shadow maps covering different regions of the viewing frustum. This places an upper bound on aliasing artifacts and reduces perspective aliasing. Temporal aliasing can still be a problem as regions pop from one shadow map unto another as the camera moves. Projective aliasing can be slightly mitigated by this, but can still be an issue.

1.2.2.2 Sample Distribution Shadow Maps.

If one is using Cascaded Shadow Maps, figuring out where to place the boundary of each map can be difficult. The usual approach is to apply an exponential split. This is to better match how fragments' depth values are distributed and can perform better than a simple linear split. Both linear and exponential methods disregard where actual objects lie in the scene. In Sample Distribution Shadow Maps [17], the boundaries are calculated using the scene's geometry. The idea is to have each shadow map be responsible for more or less the same amount of geometry. To this end, SDSM tries to calculate the geometric distribution over the viewing frustum and partition the shadow maps such that, although their physical coverage varies, their geometric coverage is similar. This prevents having some maps with little to no geometry and other maps being overloaded. In general this allows the multiple map approach to be more successful by better utilizing each map.

The basic shadow map and shadow volume algorithms cannot produce soft shadows. Therefore techniques have been developed to alter or add to these algorithms such that soft shadows can be rendered. The most popular of these algorithms are Percentage Closer Filtering [25] and Variance Shadow Maps [26].

1.2.2.3 Percentage Closer Filtering. Percentage closer filtering [25] is the most common soft shadow technique. Instead of considering only the current fragment, we consider the neighborhood of the fragment (an area known as a box filter or kernel) and calculate the percentage of neighboring fragments that are in shadow. If the percentage is zero, the pixel is lit. If the percentage is one, the pixel is in the umbra of the shadow. Any percentage in between indicates the pixel is in the penumbra and the percentage determines how dark the pixel appears. For example, if we examine a 3×3 neighborhood around the pixel (8 taps other than the pixel itself) and 3 of its neighbors are lit (i.e. have a value of zero) while the pixel itself and its 5 remaining neighbors are in shadow (i.e. have a value of one), then the final percentage for the pixel in the center is $3/9$ or 33% in shadow. In our experience, PCF produces visually satisfying results, but a kernel size of 4×4 or 5×5 is required. This increases the number of taps quadratically since, given a square kernel of size N, each pixel requires N^2 shadow map references. In contrast, both of our Gaussian algorithms require $2N$ taps due to the separable nature of a Gaussian kernel.

1.2.2.4 Variance Shadow Maps. Variance shadow maps [26] store not just the depth from the point of view of the light source, but also the depth squared, or second moment of the depth value, in a separate channel. These values are used in the main rendering pass to compute the mean and variance over a filter region. Chebyshev's inequality [27] (Theorem 1 and Equation 5 in [26]) is then used to estimate an upper bound on the percentage of pixels in the filter region that are in shadow (the same value computed by a PCF filter). This ultimately determines the darkness of the penumbra at that pixel as the light intensity can be scaled by this value. To avoid sampling all points in the filter region, one can apply a Poisson disk distribution [28] to choose a small subset of samples to represent the entire region. One can also use a pre-filtering technique such as a Gaussian blur for smoother results. It is interesting to note that very little extra calculation and memory overhead are required, yet satisfying soft shadows are generated. This approach works for different types of lights. The soft shadow results, however, have uniform width irrespective of occluder distance or other contact-hardening criteria.

1.2.3 Hybrid and Contact-hardening Shadows

Some soft shadow techniques focus on creating contact-hardening shadows and some algorithms accomplish this by creating a hybrid approach using concepts from both shadow maps and shadow volumes.

1.2.3.1 Penumbra Maps. Penumbra Maps [29] can be used to calculate soft shadows. They take the typical shadow map approach to generate depth values from the point of view of the light source. They then generate a second texture, the penumbra map, by finding the silhouette edges and expanding them using cones on the vertices and sheets connecting these cones along silhouette edges. The final scene is then rendered using the shadow and penumbra maps. This approach assumes a spherical light source, however different light shapes are supported. Though not described in their paper in exactly these terms, this approach does create contact-hardening results.

1.2.3.2 Percentage-Closer Soft Shadows. Expanding upon the PCF approach described above, the PCSS, or Percentage-Closer Soft Shadow [20], algorithm creates contact-hardening shadows by varying the PCF kernel size at each fragment in accordance with the size of the light source and distance from the occluder. This is similar to our PCF approach, however PCSS uses the light's size as well as a more complex occluder search algorithm. For our approach, light size is not taken into account and only occluder distance is used, as obtained from the original shadow map. We also use an approximation of occluder-receiver distance.

1.2.3.3 Screen Space Soft Shadows. Since algorithms like both PCF and PCSS require multiple texture lookups [30] describe an algorithm that uses a separable Gaussian kernel, requiring fewer lookups. They

call their approach Screen Space Soft Shadows (SSSS). Our approaches are also performed in screen space and the technique described by SSSS is similar to our single pass Gaussian approach. However in our multi-pass approach, as mentioned earlier we use a Poisson disk distribution to reduce the total number of taps to a constant, kernel size agnostic value. This also allows us to render the scene once per blur as opposed to twice; once for horizontal and once for vertical. This further allows us to explore much bigger kernel sizes.

1.2.3.4 Multi-View Soft Shadowing. Multi-View Soft Shadowing (MVSS) [31] was created by nVidia. This approach uses multiple shadow maps from different points on an area light source. These points are chosen using a Poisson disk sampling pattern. For each pixel, each shadow map is queried and an average over all maps is taken. This average controls the strength of the shadow at that pixel. PCF with a constant kernel size of 2x2 is applied to all shadow map references.

1.2.3.5 Contact-hardening Soft Shadows using Erosion. Klein et al [21] uses an erosion operator. They first generate classic hard shadows, then perform edge detection with a Laplacian kernel. For those pixels detected to be shadow edges, the occluder and camera distance are stored in separate channels. This information is then used to calculate penumbra width, which in turn is used to scale an erosion kernel applied in the next pass. The final pass can use PCF with a kernel size also scaled by the penumbra width.

1.2.4 Shadows for Bump and Normal Maps

In a similar spirit to our main algorithm, horizon encoding shadows, there have been attempts to create shadows for bump and normal mapped geometry. Whereas we use a displacement map, the techniques do have a few aspects in common with ours and are important to go over. Horizon mapping can be used for bump mapped shadows since bump maps do not alter geometry and therefore one is attempting to apply shadows to a smooth surface. Our approach uses a technique similar to accessibility maps, which are similar to horizon maps but only encode a single value to describe the horizon, even though our displacement map does alter the geometry. The reason for our choice in using horizon line encoding even though we are generating shadows for a truly bumpy terrain will become apparent later, but suffice to say, the magnitude of the geometry we are handling as well as the caching mechanism of the displacement map warrants the use of uncommon techniques that are normally applied to other situations. Below we explore techniques for bump mapped shadows with and without using horizon maps.

1.2.4.1 Without Horizon Maps. Kautz [32] shows a technique for pre-computing faux shadows over a bump map. For each pixel, rays are casted in all directions and either hit another bump, ”shadow rays,” or leave the surface, ”light rays.” An ellipse is then fitted that divides light rays, inside the ellipse, from shadow rays, outside the ellipse. The parameters describing these ellipses are encoded into three textures and are queried in the GPU . A simple comparison with respect to these textures and the light’s angle can indicate whether a pixel is in shadow or not. They even have an altered approach for generating anti-aliased shadows. While this approach does rely on preprocessing of an image to store shadow information, the similaries with our method mostly end there. As mentioned this approach only works with a bump map and further, their approach forms a different type of approximation from our own.

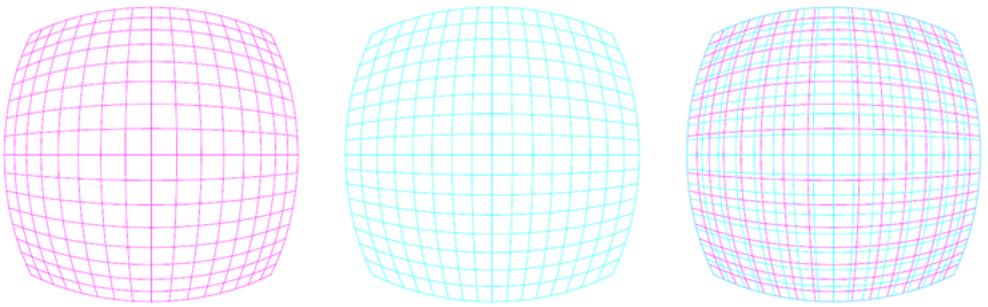
1.2.4.2 With Horizon Maps. Sloan [33] uses horizon mapping for bump map shadows, wherein they capture horizon information from eight cardinal directions, for each pixel, and interpolate to form the other values. The light source is projected into the local tangent space of the pixel and visibility is calculated by a ray going from the horizon to the surface normal. In contrast, our algorithm operates per vertex, captures horizon information from about 90 directions, and is calculated by going from the vertex towards the light

source. We also add several optimizations that may not be applicable to Sloan's approach, however, both our algorithms do spread the horizon values across the channels of the texture.

CHAPTER 2. BACKGROUND

2.1 SCM

Spherical Cube Maps [1] is a concept developed by Dr. Robert Kooima that replaces the normal linear cube mapping commonly done (Figure 2.1). When rendering a texture onto a sphere, distortion occurs at the poles and near the equator. The texture is not uniformly sampled everywhere on the sphere because of this.



(a) Linear cube map (b) Spherical cube map (c) Overlaid

A comparison of linear and spherical cube map sample uniformity

Figure 2.1: SCM Cube Mapping

A better approach is to start with a cube, and adaptively tessellate the faces while moving the newly created vertices outwards from the center (Figure 2.2). After several iterations of this, the geometry resembles that of a sphere and UV coordinates are more linearly interpolated across the surface.

(a) 0 (b) 1 (c) 2 (d) 3 (e) 4

Recursive subdivisions of the cube

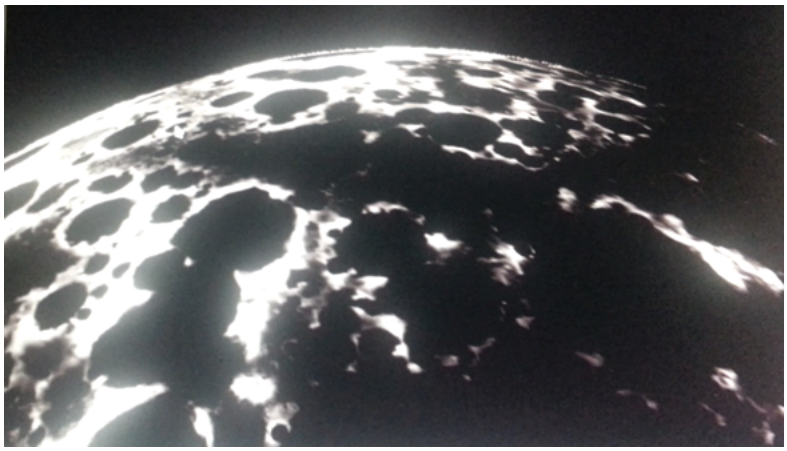
Figure 2.2: SCM Subdivide

This is how the lunar rendering is done. As the user zooms into an area, tessellation occurs and vertices are moved according to the lunar data at that point. This allows for more fine grained vertex displacement and progressively better topology detail. As the user zooms out, geometry adapts and less detail is shown.

2.2 Learning the Essentials

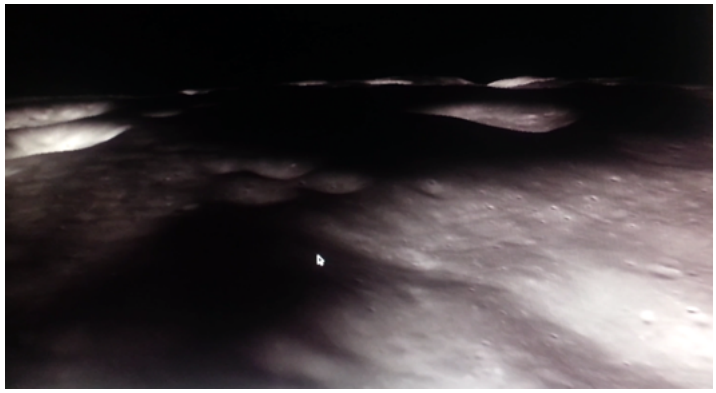
The final goal is to create shadows on the moon's surface using the SCM framework to render the gigapixel data. Before work can begin on this, the SCM framework, both in theory and implemented, must be examined and learned. To this end the code base was retrieved and placed on an accessible desktop machine. The initial goal was to make small changes and ensure these changes would be applied, thereby ensuring the build process was functioning on the machine. The first change attempted was to turn the moon red using

the fragment shader. This simple task helped to establish some sort of work flow and a better understanding of what the shaders and ultimately the client side code did. Other changes were made in the pursuit of full understanding such as creating a high contrast surface on the moon, Figure 2.3 and creating a grayscale elevation map, Figure 2.4. Another modification was to create a colored elevation map, Figure 2.5.



Shader modification to the SCM framework to show the moon in higher contrast than normal.

Figure 2.3: High Contrast Surface



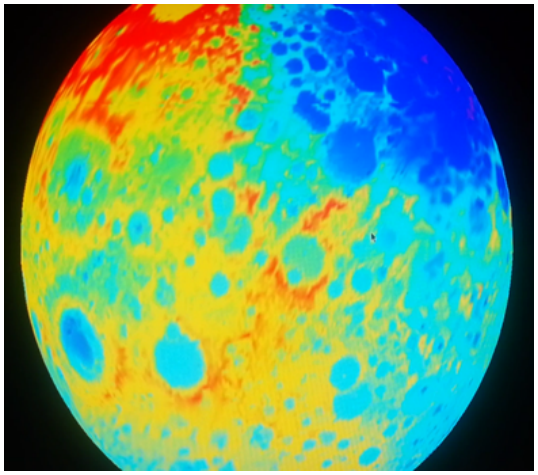
Grayscale elevation map using fragment's object space radius from moon's center.

Figure 2.4: Grayscale Elevation Map

After getting a better understanding of the SCM code, reading published research papers on shadows was the next step. One of those papers, Cascaded Shadow Maps [16], was an interesting read and was intriguing to think of that algorithm implemented on the moon, Figure 2.6.

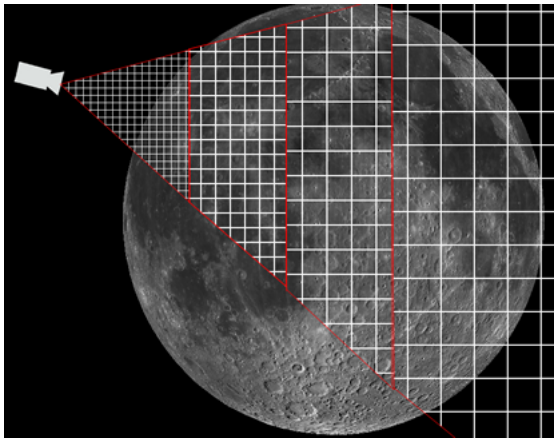
Another paper, Sample Distribution Shadow Maps [17], was read as part of a presentation assignment in which a summary must be given. The basic concept, as discussed earlier in the Related Works section, is to examine the scene's geometric distribution and utilize this information to better decide where the boundaries of each shadow map should lie when using Cascaded Shadow Maps [16].

While reading these papers helped gain a rudimentary understanding of some of the advanced shadow methods, an understanding of the basics was lacking. Therefore while trying to get shadow maps to work in the SCM framework, work was also being done in separate stand-alone OpenGL projects to help with basic



Color elevation map using fragment's object space radius from moon's center. Shader uses linear interpolation between 6 main colors (i.e. $red->yellow->green->cyan->blue->purple$)

Figure 2.5: Elevation Map using Colors



Concept drawing for how CSM partitions would look for the SCM moon rendering

Figure 2.6: CSM Partitions for SCM Moon Rendering

concepts. Having separate blank projects helped to experiment with simple OpenGL basics in a smaller and more controlled environment. These projects were used to help fix issues in the pursuit of shadows in SCM .

While work on the SCM framework was slowly progressing, it was discouraging that certain goals were not being met in a timely manner. A decent amount of work went into just creating an fbo , rendering the entire moon to that fbo , then simply displaying it on the screen onto a full screen quad. This was necessary so as to collect the depth information from the point of view of the light source, thereby creating a shadow map. An SCM -Shadow class was created to help organize the shadow code and was injected into the SCM framework. After correctly implementing the fbo and getting the moon to render correctly to the color and depth buffers, it was at this point that some resemblance of shadows started to appear. They weren't fully correct but visually something resembling shadows was taking place, therefore at least some of the code written was correct. It seemed as though the shadows were being mapped to the different individual SCM pages or squares in the geometry and wasn't really looking right but at least there was a white moon with black shadow regions (lit portions of the moon was colored plain white so as to enhance the appearance

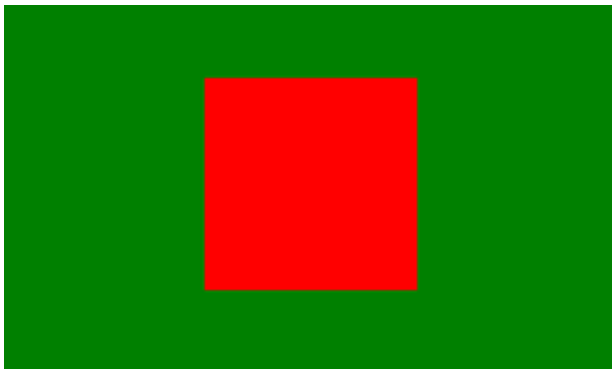
of shadows). In addition to this, work on basic OpenGL concepts were proving more troublesome than they should have been. With OpenGL if one doesn't have an intimate understanding of what it does and how it does it, then mistakes are very easy to make. Missing one thing can mean ending up with a blank screen and no error messages.

The goal was to write short programs from scratch and when completed, start over with a new program. Learning exactly what OpenGL was expecting and having it become second nature was key. This included memorizing most of the ceremony stuff and understanding what each of the values mean. It was also important to understand the meaning behind why OpenGL requires so much work to do what seems like simple tasks. OpenGL wants to give the programmer complete control so it gives only the basic functions necessary to set everything up. The more powerful the functions and the more they do, the less control one has. So instead of just giving a single function to make a shader program, for example, it gives several functions meant to set a shader source, compile it, and attach it.

Previously when code was written to accomplish a task and found to work correctly, it was saved into a library so as to refer to it again. This made projects a little easier to do but hurt in the full comprehension of said task because sometimes full understanding of how that tasks works could be lacking. Also, looking back over the code that was compiled into the framework, certain gaps in understanding was apparent as certain things were done that were ultimately unnecessary. This could lead to multiple improvements in several parts of the library. So while it may have taken awhile to relearn everything, the concepts that were relearned were done so at a deeper level. For example, the difference between glActiveTexture and glBindTexture was not well understood. Now the understanding is that there are texture units set aside in the graphics card and glActiveTexture says which one is currently being pointed to. When one binds a texture, one is transferring that texture into the active texture unit. So one can theoretically have as many textures as desired via glGenTextures but only a fixed amount of them can be bound at a time because these exists a fixed amount of slots available (said amount is hardware dependent). One of the issues noticed in the framework was when glActiveTexture was used in unnecessary places showing a lack of understanding of the purpose of that function.

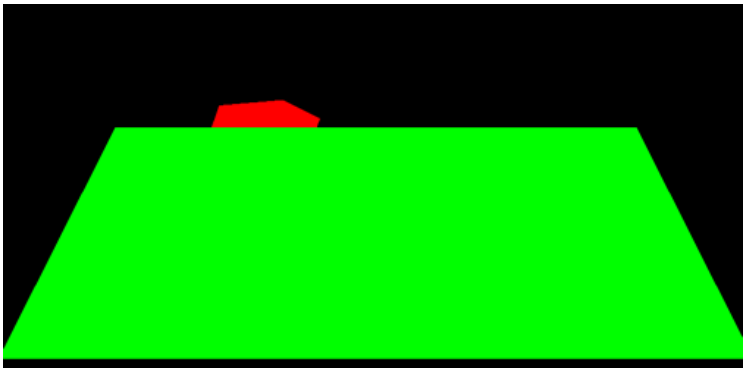
At this point there were some kind of shadows showing up on the moon (even though they were wrong) and countless different attempts at correcting those shadows. The logic seemed fine but wouldn't produce anything. The good thing was that at that point, a decent amount of understanding of the SCM framework was present, or at least the part that needed to be modified. A good work flow had been established and knowledge of how to change which shaders were used and how to render everything to an fbo to manipulate it was gained. This didn't mean that shadows were correct on the moon, however. Progression was slow and the decision was made to forgo work directly on the SCM framework, and to focus instead on custom scenes developed in a different project. This project was created as part of learning OpenGL . Even though shadows were not appearing in this environment either, the task was easier to create shadows here as opposed to the moon. Later after shadows were working on the custom scenes, and overall understanding of shadow algorithms improved, the plan was to re-implement these algorithms in SCM . Figures 2.7, 2.8, 2.9 show the progression of learning about the fbo and how to write data to it.

After learning more about how to write to an fbo , Figures 2.10, and 2.11 show a sample scene with initial attempts at utilizing the buffer information. This was the basis of learning how to use shadow maps and how to correctly apply them to a very simple scene. The textures used were just default from the 3D modeler as the concern at the time was not to make a beautiful scene, but to get shadow map (at least the basic algorithm) working at some capacity.



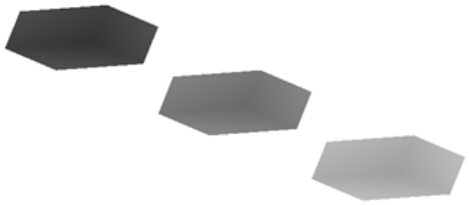
Practice rendering to an off-screen fbo and displaying the attached color texture on-screen.

Figure 2.7: Rendering from Fbo Color Buffer



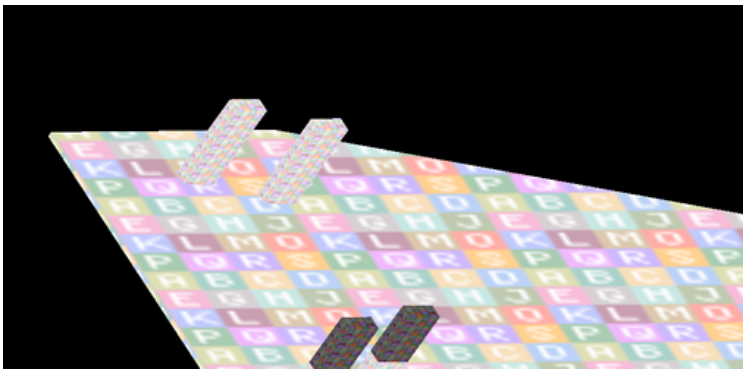
Plane created from separate vertices drawn using GL_QUADS. A bug caused the red box to be rendered behind the plane when according to the z value, it should be in front. Later it was discovered that GL_DEPTH_TEST wasn't enabled.

Figure 2.8: More Rendering from an Fbo Color Buffer



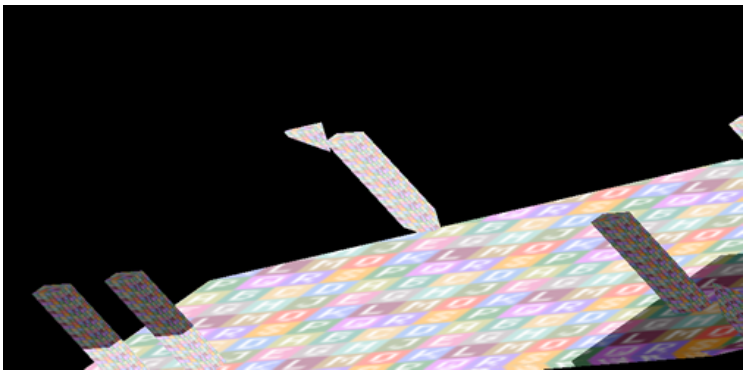
After ensuring the enabling of GL_DEPTH_TEST, the scene was rendered and drawn from the depth texture attached to the fbo . The scene is composed of 3 boxes with various z values. Another bug was found where the projection matrix was never set leaving it at the default identity matrix, therefore the boxes were not displayed using perspective projection.

Figure 2.9: Depth Map of 3 Boxes



This scene is created with a very simple model made in Wings3D and has the default texture attached to it. Something still doesn't seem right about the way the model is rendered. After examining this, the thought was that there was still something wrong with the projection transformation.

Figure 2.10: Finally Some Sort of Shadows



For some reason the scene was being clipped by the near and far planes even though in code they appeared to have correct values.

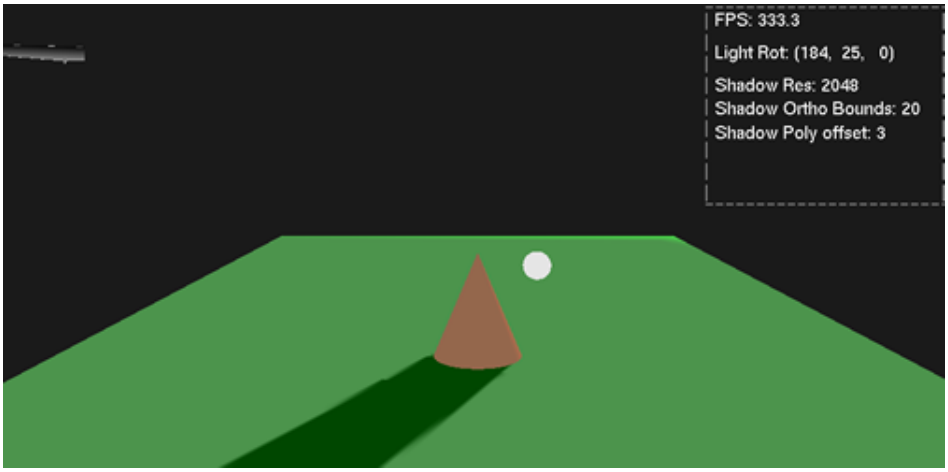
Figure 2.11: Same Scene as Above with Different View

For reasons unknown at the time, the reshape function in glut was proving difficult to get working. When glutReshapeFunc was set and normal code for reshaping the window was ran, the scene wouldn't display correctly. The scene was being rendered it to an fbo and the color texture bound to it was being displayed in a 2D quad stretched to fit the screen. The problem turned out to be that the default projection transformation (as with any default transformation) is initially set to the identity matrix. For a projection matrix this is equivalent to a call to glOrtho using -1 to +1 for all values. This makes sense since everything gets projected into a -1 to +1 cube during the normal rendering steps of a scene. So if one doesn't modify the camera frustum, then the frustum is already in this shape and no actual transformation occurs when projecting. If one does modify this to, say, a perspective projection by enlarging the far clipping plane, then during this transformation into the -1 to +1 cube, geometry closer to the far clipping plane gets shrunk as the entire frustum gets transformed. This is why objects further away appear smaller in this style of projection. For the custom scene being used, the reshape function was modifying the projection matrix stack and changing it to a perspective projection. This was great for the scene and when rendered directly to the screen it looked fine. But if we wanted to render a flat plane with a color texture on it from the fbo , then we had to make sure the plane was in the viewing frustum. It was in the frustum if we kept the projection matrix stack as the identity matrix, but not if it was changed it a perspective projection. Realizing this was a big step forward.

2.3 Creating the Shadow Application

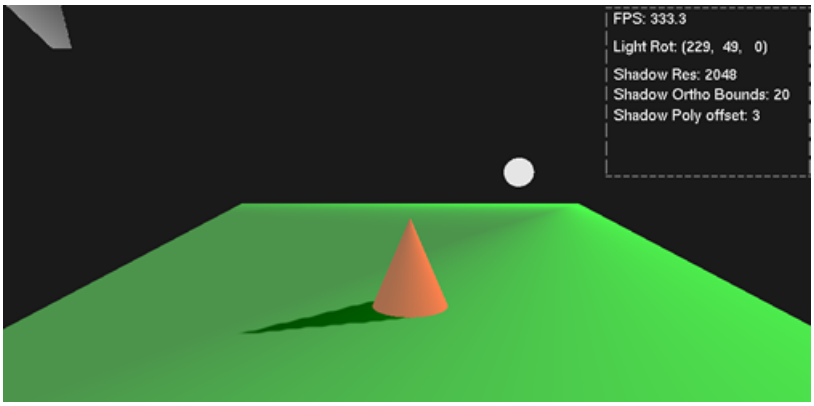
Even though a greater understanding of how things work was obtained, there was still not much progress towards the goal. It seemed like some aspect of OpenGL in my application was always missed every time a restart from scratch occurred. It was because of this that the decision was made to create a framework from scratch so at least basic concepts such as vertex and fragment shader compilation, fbo reading and writing, lighting setup and properties, and other ceremony code would not have to be re-typed each time. Since there was now a better understanding of the fundamentals of graphics and OpenGL , making a framework wouldn't hurt any comprehension but only serve to speed up the process of getting shadows to work by saving time not rewriting what was already working. This framework came to be called GLshadow. It was written in C++ (up until this point only C was being used) because SCM was written in C++. Classes were created for shaders, FBOs, lights, scene, and shadows. A separate project was created to consume GLshadow and to create the custom scenes that would require shadowing.

The decision was made to stick primarily to view and projection transforms and not worry about model transforms for now. With this shadows started to work somewhat after using everything learned up to this point. Eventually they began to work quite well after a bug was fixed. The fix was from a realization that a mistake was made with respect to when to reverse the view matrix and when to use it as is. When rendering from the point of view of the light source, one should simply use the light's model transformations as view transformations by applying the inverse of it to the ModelView matrix. In general, to render the scene from another geometry's point of view, one uses the inverse of that geometry's model transformation as the view transform in the ModelView matrix and then just render the rest of the geometry like normal. Another mistake was found in the form of using the light's rotation as its position. Once these issues were fixed, the shadows started looking a lot better. A HUD was added to display the current frames per second, light rotation value, shadow resolution, shadow orthographic bounds, and shadow polygon offset. Figures 2.12, 2.13, 2.14, 2.15, and 2.16 show the progression of shadows in the custom scenes.



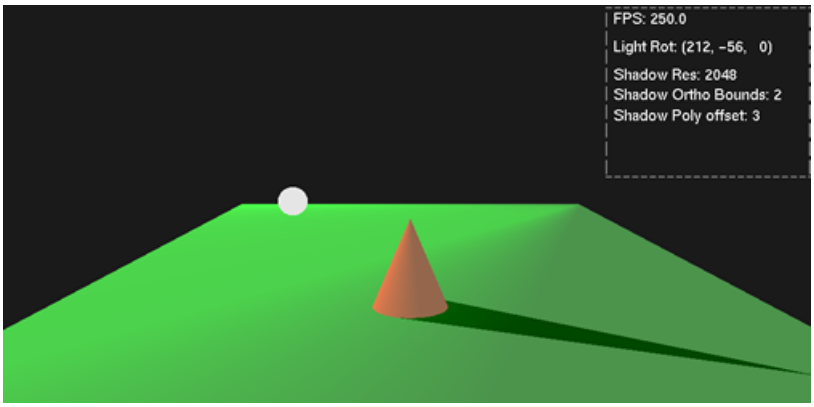
The sphere represents the position of the light source and its color matches the diffuse color of the light. In the top left is the depth map of what the light sees. In the top right is the HUD. The shadow map's orthographic boundaries represent an isomorphic size, i.e. in the figure above, each direction is -20 to +20.

Figure 2.12: Illumination and Shadow Casting of an Orange Cone on a Green Plane



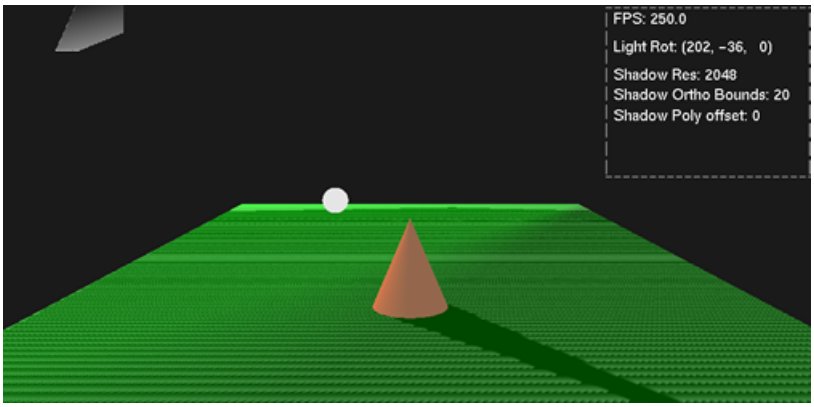
Plane and cone have per pixel lighting calculations. The plane's normals have been set to create a faux bevel appearance; just makes the plane nicer to look at.

Figure 2.13: Illumination of Plane



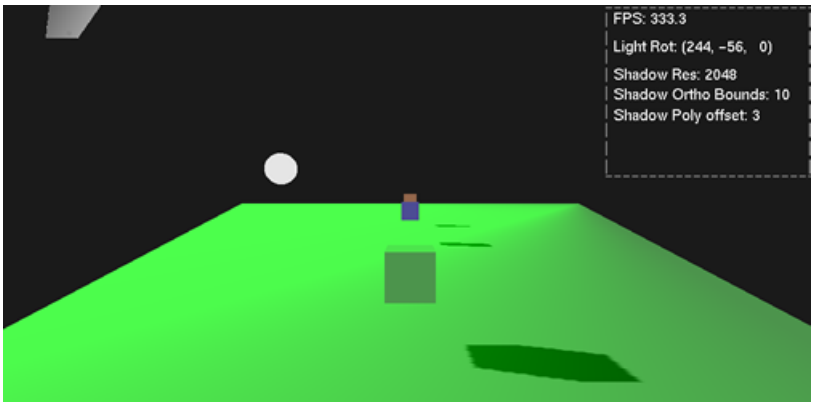
We can sharpen the shadows by either reducing the orthographic bounds or increasing shadow resolution.

Figure 2.14: High Resolution Shadows



Decreasing the polygon offset produces bad shadow acne. Back face culling should get rid of this but due to a bug in the program, it is not.

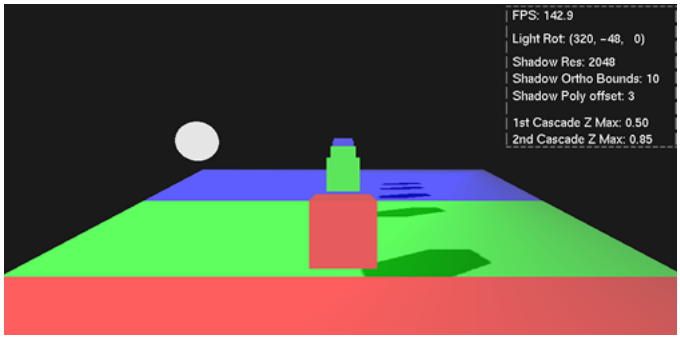
Figure 2.15: Shadow Acne



Simple scene showing shadows from the boxes being cast onto a plane.

Figure 2.16: Scene with 3 Boxes Each Casting a Shadow

Next goal was to start working on creating multiple shadows. The framework was adjusted to allow a scene to be rendered with a pointer to a light source (which could later be adjusted to make for an array of light sources). The light source would then render the scene to each of its shadow maps, and then render the scene once more using the collected shadow data. The light would then render itself as a glut sphere and its depth map would render like it was doing before. The HUD is a separate class whose rendering is invoked by the client application. In the shader a uniform was added to allow for the different cascades to be tinted different colors, as this was something other CSM implementations seemed to do. It turns out to be a good debugging tool to know where the shadow map boundaries lie (or at least where they should lie). Figure 2.17 shows what the cascades look like in the custom floating box scene.



The partition boundaries are not yet accurate, as can be seen by the absence of shadow in the red zone; however there are 3 separate shadow maps with scene geometry rendered to them. In the shader rendering the scene, gl_FragCoord.z is used to determine which shadow map to reference.

Figure 2.17: Initial Attempt at CSM

The light gizmo was changed to a model of an arrow so as to better show it is a directional light as opposed to a spot or point. Multiple light sources was never fully implemented but instead was changed to a single light source that has a drop down list box to control which algorithm it uses for rendering shadows.

Work continued on the framework and more core functionality was developed. The decision was made to migrate the code from C++ to C# and use a library called SharpGL [34], which is basically a C# wrapper for OpenGL . This was done for several reasons: 1. Windows Forms could be used which is an old but established GUI system. It comes with everything needed to create a GUI for testing the code. Searching online for a good C++ OpenGL GUI resulted in finding either paid software or free but overly complex

for what was needed. Each one also came with its own learning curve and a good amount of setup in the code. 2. C# has more language features that make for quicker development and prototyping. 3. The .Net framework comes with a lot of functionality and is more consistent than adding multiple C++ libraries from different sources and using them all together. After the decision to switch over to C# and SharpGL , the entire code base had to be converted. This was done hand and fortunately it didn't take long. The SharpGL library contained mostly the same functions as OpenGL with the same parameters. Some of the functions required some tweaking but most required very little work. After converting the code, new classes were added and everything was partitioned into two main .dlls; GLstandard and GLshadow, and an app was created called Client. The Client app consumed these dlls and created a GUI to display the results of their work. GLstandard contains all the standard code required by OpenGL to do basic things like compile and link a shader, use frame buffer objects, and modify the viewport. It also contains a base Light class to be derived from in GLshadow. GLshadow contains code specifically for shadow implementation and utilizes functionality in GLstandard. It includes Light classes that inherit from the base Light class. These classes turn the basic light into a light that uses shadow maps, shadow volumes, or any other type of shadow implementation. It is setup such that each new shadow method implemented derives from the base Light class. This includes a Shadow class implementation that goes along with the specific shadow method. If optimizations change the way shadows are calculated in a major way, then new Light and Shadow classes are created and used in lieu of modifying existing ones. Several test scenes were added to the existing set of custom scenes to further test the quality of the shadow algorithms implemented. These included a scene with a pool deck, chairs and umbrellas; a chain link fence; a ball bouncing on the ground; and a log cabin, punching bag, and pyramid.

2.4 Integrating SCM Framework

The main framework was now in C# while the SCM TIFF rendering code was written in C++. A decision had to be made to integrate the two. There were a few options here: 1. Recode the main framework into C++. If this was done, the benefits that were the driving force behind moving to C# in the first place would be lost. This would also cost a decent amount of development time especially since some concepts may be harder to accomplish or require more coding effort in C++. 2. Recode SCM framework into C#. Dr. Kooima had reservations about this approach and warned that this would be no easy task. He even stated this might not be 100% possible due to certain aspects in the multi-threading parts of the code as well as other parts. This approach, like the first, would also require a decent amount of development time. 3. Get C# to run the native, compiled C++ source code. Option 3 seemed like the best option as it didn't involve having to recode either framework. This option could be accomplished via com interop, which is built into .Net. One just has to know what to add to the C++ source to get everything to work. To practice this, a simple example was created with a single function. This example worked fine, however no literature was found online for how to convert entire classes, only global functions. Not only this, but having to add a lot of extra code into the SCM framework for each class and method is not very desirable and could be time consuming. Instead a program called Swig [35] was found after an online search. Swig can be applied to a C++ project and will create a separate file with all the hooks written automatically. This file can then be compiled with the project and a .dll will be produced that can be consumed from a C# project. Corresponding C# classes are generated from the C++ classes and can be added into the C# project. These classes provide the interface information and create a bridge to the corresponding code in the .dll. Once Swig was integrated into the project, the SCM code was able to be invoked from within the C# framework.

The first problem encountered after getting the native C++ code to work from within the C# app, was with the LibTIFF library. For the SCM framework to function correctly, the version had to be 4.0.3. This version did not have a Visual Studio build available from the developers and therefore needed to be compiled

from source to function correctly in this environment. For the source to be built, first the libraries ZLib, LibPNG, and LibJPEG had to be built. This was not a trivial task and took several days to figure out. After awhile of trying, a version of LibTIFF known as BigTIFF was found online and a Windows version with support for Visual Studio existed. This library was able to read the displacement map and color map TIFFs , but not the normal map. This wasn't the correct version of LibTIFF required however for the time, it worked good enough. Eventually LibTIFF was successfully compiled and now the correct version is running with Visual Studio and is working fine. This is important not just because we can now read in the normal map, but also since the custom shadow SCM TIFFs cannot be read with the BigTIFF library, and so required the correct version of LibTIFF as well.

There are several scenes in the application and normally adding a new scene isn't a problem. Adding a scene for the moon was more complicated, however, since the way the code works, the shader must be given to the SCM code and one cannot use their own compiled program. There were some other code that needed to be changed as well, therefore to make integration easier, all scenes were removed and the app was entirely focused on the single moon scene. Once that worked and the SCM function calls were fully integrated, other features were worked on. Later on, in an effort to ensure the algorithms being developed would work on the previously included scenes as well as the moon, the scenes were re-introduced into the app. At first this wasn't easy since a lot of the code was dependent on there being a single scene. To compensate for this, static global values were introduced that would affect all scenes in addition to scene specific values that would only affect the current scene. At this point, the moon was considered just another scene, alongside others selectable from the drop down box in the GUI . A few default values had to be set up to handle the large scale difference from the moon over the other scenes. One of these values was the bounding box of the light for shadow maps. There is a control for adjusting this value in the GUI , however when jumping from the moon scene to another scene, or vice versa, the range of valid values had to be adjusted. Also the camera can be moved around, but the scale of its movement is adjusted depending on whether the scene is the moon or not.

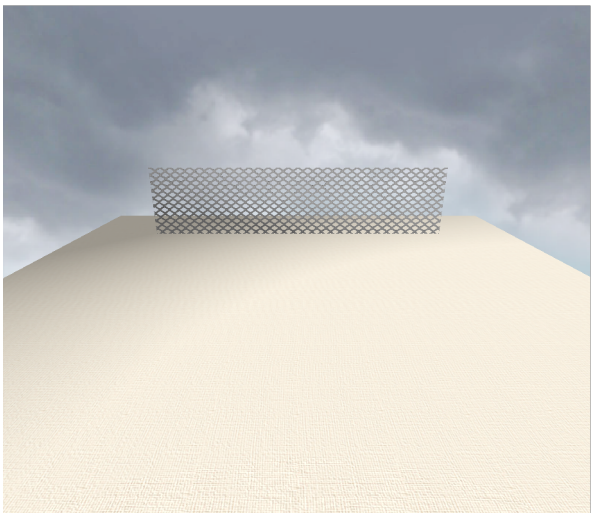
2.5 Custom Scenes

The list of custom scenes created and used in the app and in figures in this document, not including the moon, can be found in Figure 2.18. Information regarding the details of each of the scenes, such as number of vertices, triangles and total meshes, can be found in Table 2.1. We try to use a variety of scene sizes to better test the speed and flexibility of our algorithms. Ultimately the goal was to produce shadows on the moon, which use a displacement map, but most of the algorithms implemented in the code can apply to other scenes as well. While we can't test every type of scene, these scenes were meant to provide a decent set of testing environments.

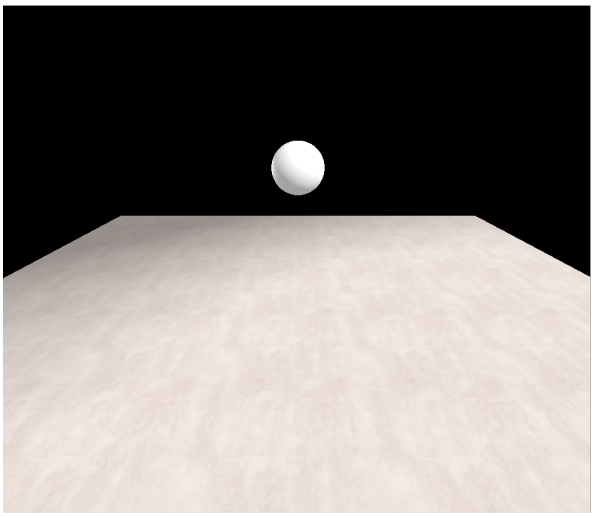
Table 2.1: Custom Scene Statistics

Statistics from custom scenes including total number of vertices, triangles, and separate meshes.

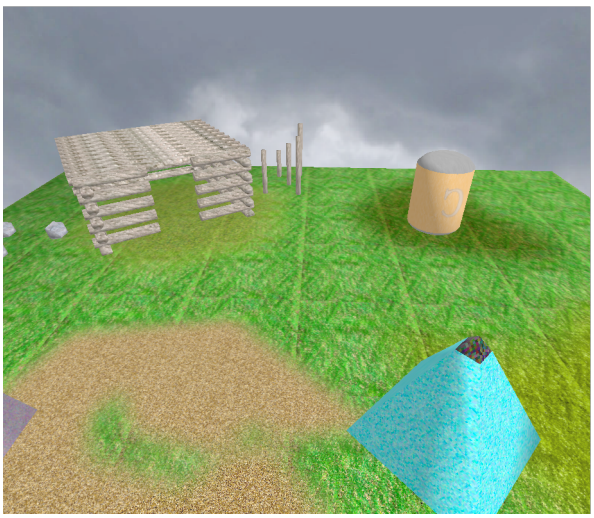
Scene Name	Vertices	Triangles	Meshes
Fence	16	24	2
Ball	880	1,752	2
Log Cabin	1,407	2,630	13
Pool	8,463	11,003	1



(a) Fence



(b) Ball



(c) Log Cabin

(d) Pool

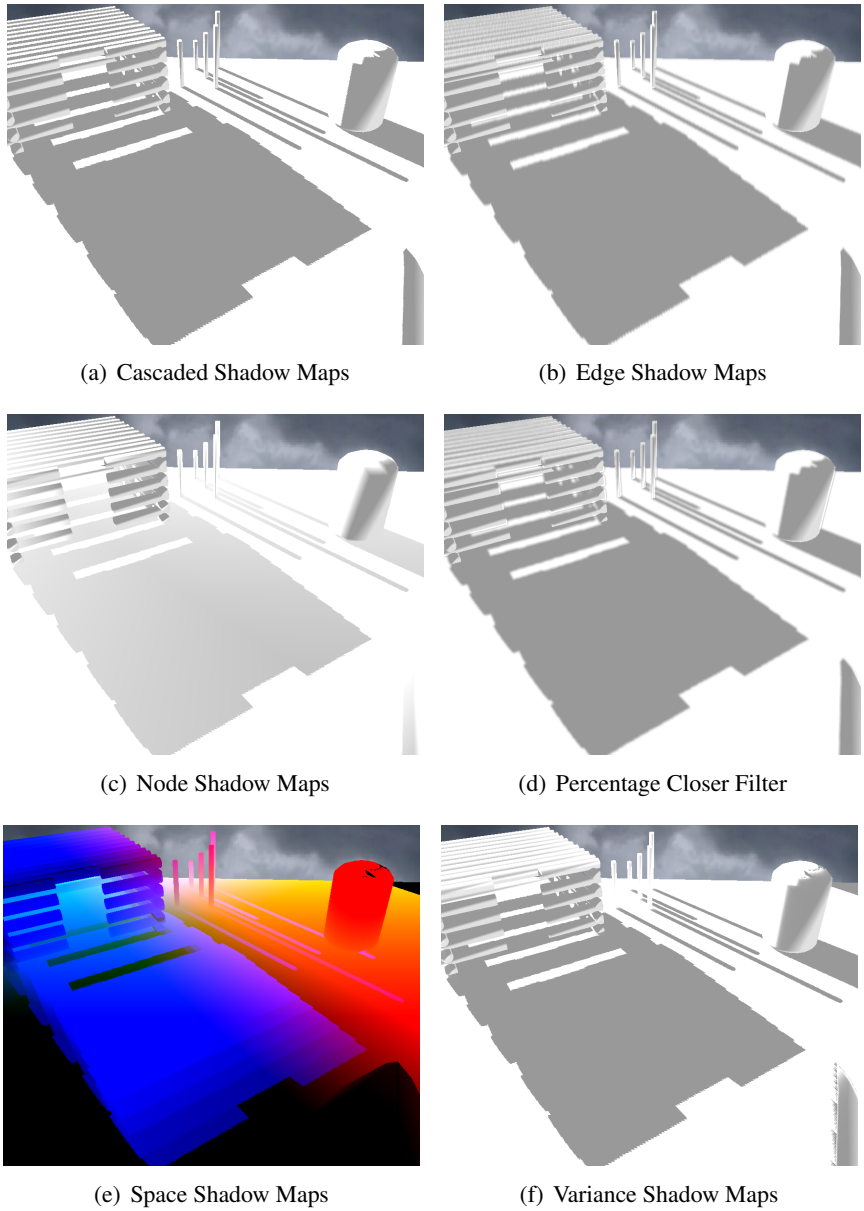
Scenes, in addition to the moon, that are used in testing the shadow algorithms.

Figure 2.18: Custom Scenes

CHAPTER 3. SHADOW MAPS AND VOLUMES

3.1 Algorithms Implemented

The code base has classes for implementing different kind of shadow algorithms. The following algorithms have been implemented and an example of each can be seen in Figure 3.1.



(a) Cascaded Shadow Maps

(b) Edge Shadow Maps

(c) Node Shadow Maps

(d) Percentage Closer Filter

(e) Space Shadow Maps

(f) Variance Shadow Maps

Figure 3.1: Shadow Map Algorithms

3.1.1 Cascaded Shadow Maps

The basic Cascaded Shadow Map algorithm is implemented and can handle up to 4 cascades (Figure 3.1(a)). There is no special code for deciding where to place the splits like in Parallel Split Shadow Maps [36] or

SDSM [17]. Instead the values are hard coded in. Figure 3.1(a) shows this algorithm in a simple, non-textured scene. Note that there are some issues still remaining in the implementation such as the scene geometry not rendering completely correct and the light's near and far clipping planes are hard coded in. Nevertheless the shadow cascades do appear to work adequately and were mostly done to gain knowledge in the area and to see how they appear in various scenes. For the moon as will be discussed later, we only use a small patch to compare algorithms both on visual results and on speed, therefore with such a small area, cascades wouldn't be of much benefit. Unless otherwise noted, the CSM approach uses only a single cascade in timings and images.

3.1.2 Edge Shadow Maps

The general idea here is to render to an off screen buffer a value of 1 or 0 depending on whether a pixel is in shadow or not. We then blur the contents of this buffer using a Gaussian. The final render uses this buffer with values in between 1 and 0 to determine whether a pixel is lit, in penumbra, or fully in shadow. This algorithm inspired another algorithm, called Multi-Pass Gaussian Shadows (MPG), which we will fully introduce later. There are three approaches to MPG and Edge Shadow Maps represents the 'single-pass' approach. Figure 3.1(b) shows Edge Shadow Maps.

3.1.3 Node Shadow Maps

Another way to generate a floating point value in the 0 to 1 range and use it for soft shadows, is to calculate the basic shadow map algorithm and use it only on the vertices. Node Shadow Maps (Figure 3.1(c)) gathers its information like normal, however when using this information, only the vertices tap into the shadow map. If the vertex is in shadow, it sets a value of 0. If the vertex is lit, it sets a value of 1. This value is then interpolated to the fragments. This method only works for highly tessellated scenes or regions where the area of a triangle only covers a single pixel after rasterization. In these cases, the accuracy can approach that of normal shadow maps. To help with error, an option was added to toggle using fragment shadow taps as a mask. If basic shadow maps done in the fragment shader indicates that the pixel should be in shadow, the value interpolated from the vertex shader is allowed to be used as the penumbra value. If it indicates the pixel should be lit, the interpolated value is not used and instead, the pixel is changed to being not in shadow. For highly tessellated geometry, this fragment mask should not be necessary, but for geometry with a low vertex count, without this option shadows may not appear at all.

3.1.4 Percentage Closer Filtering

This is the basic PCF implementation as described in the related works section and can be seen in Figure 3.1(d). This was implemented since it is one of the most common soft shadow algorithms to add to shadow maps. It is so common that some graphics card, like those from nVidia, will perform PCF automatically if certain OpenGL and GLSL commands are used. Instead of relying on this behavior, however, we decided to implement PCF ourselves so as to have complete control over the parameters like kernel size. This betters helps us to compare PCF to our other algorithms.

3.1.5 Space Shadow Maps

In Space Shadow Maps (Figure 3.1(e)) each lit fragment is colored according to its position in one of four different spaces. The choices for space are object, world, eye, or clip space and can be control via a GUI element. A fragment in shadow is colored according to the position of its occluder. The purpose of this algorithm is mainly for debugging shadow implementation issues.

3.1.6 Variance Shadow Maps

As stated in the related works section, this is our implementation of Variance Shadow Maps and can be seen in Figure 3.1(f). This, like PCF, is another popular soft shadow approach to shadow maps, although direct hardware support doesn't exist. This algorithm will be compared to our other approaches that produce soft shadows.

3.1.7 Shadow Volumes

To implement shadow volumes correctly, we need to know all the vertex adjacency information, which consists of knowing which vertices are next to which other vertices. This information needs to be generated from the geometry and in our implementation, that functionality is not present. Instead, adjacency information for a very simple scene involving just a cube has been hard-coded in and used (see Figure 3.2). This algorithm was implemented mostly to further understand how shadow volumes work and understand both depth pass and depth fail versions. Further understanding lead to contemplation on whether or not this shadow technique should be applied to the moon rendering. In the end the intense amount of extra GPU work lead to this not been implemented for the moon. We believe that considering the extreme amount of vertices already present in normal rendering, to compound the amount of geometry to process by examining each edge and adding many new volumes for the GPU to render would be prohibitively slow and would not be able to ran at real-time speeds.

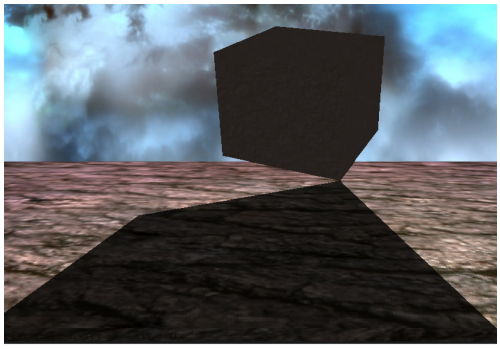


Figure 3.2: Shadow Volumes

3.2 Multi-Pass Gaussian Shadows

Our algorithm, "Multi-pass Gaussian Contact-hardening Soft Shadows" [8], was published by Scitepress and presented at the "International Conference on Computer Graphics Theory and Applications (GRAPP 2015)" conference, held by Visigrapp. Multi-pass Gaussian shadows (MPG) is an approach that focuses on creating variable penumbra widths that scale with occluder distance. Toward this end we explore three different alterations. Two of our approaches require only one pass. One of them uses a PCF filter ("MPG PCF") and the other a Gaussian filter ("MPG SP", where SP stands for single pass). Our main alteration uses multiple Gaussian passes ("MPG MP", where MP stands for multi-pass). The occluder distance controls the number of passes. The size of the kernel can either be fixed or vary with each pass and the standard deviation of the kernel is linked to that size via the 3-sigma rule, which enforces that sigma be equal to size divided by three. We use a Poisson disk distribution with a custom number of taps to control where we sample the Gaussian kernel. It is important to note that this value is separate from kernel size. As kernel size increases, the taps expand to cover the new area. All three operate in screen space as a post process

over what we call a *distance map*. The resulting information in this map is then used to generate the shadow regions in the final pass. This information will control the penumbra intensities of each fragment and will create contact-hardening shadows.

3.2.1 Description

MPG begins in the first pass with the standard shadow map being created by rendering only the depth information of the scene from the point of view of the light source into a depth buffer. This gives us a reference and a starting point for the rest of the calculations.

The next pass renders from the point of view of the camera. Here we perform the normal rendering pass that would generate hard shadows from the previously collected shadow map. However, instead of drawing fragments either lit or in shadow, we save this binary value into one of the channels of our distance map (with zero for lit and one for shadowed) along with the distance to the occluder in another channel. The distance to the occluder is simply the difference between the distance of the current fragment (projected into light space) to the light source and the value looked up in the shadow map. The distance is then raised to a user-specified power, Dp, and then multiplied by a user-specified value, Dm. This is done to help control how sharply the penumbra portion grows and how long the contact-hardening portion lasts before spreading into a penumbra region. These values should scale with the general size of the scene and estimated average occluder distance. When figuring out the proper value for these two variables, it helps to draw the occluder distance into the scene via a color gradient so one can visually examine the modified occluder distance to ensure a proper progression of penumbra scale.

The third pass is optional and depending on the scene, it can help areas where shadows form thin lines on the receiver. It can also help to expand the penumbra such that both and outer and inner penumbra region is visible. This pass dilates the occluder distance values found in the distance map that was written during the second pass. Simply apply a Gaussian filter over the channel to expand the penumbra region in later passes.

Pass four is where the three alterations come into play. The first alteration uses a PCF filter to modify the binary shadow value from our distance map using the occluder distance to control filter size. The second alteration is similar but uses a Gaussian filter to modify the shadow value and uses the occluder distance to control the size of the Gaussian kernel. The last alteration performs multiple Gaussian blur passes modifying the shadow value and decreasing the occluder distance with each pass. Those fragments with an occluder distance that has been decremented to zero (or started out at zero) do not get blurred for all subsequent passes. The kernel size can either be fixed or can be upper bounded with the first pass starting at small values (e.g. size of 3x3) and can increase with each pass until reaching the upper bound. This will speed up the multi-pass part. We precomputed up to 64 two-dimensional Poisson points in the inclusive range of -1 to +1. The first point is fixed at the origin so the center texel is always examined. After the first point, the Poisson algorithm proceeds as usual to create a valid Poisson distribution. Before we make our blur pass, we take from the first of these Poisson points up to the desired amount and use these as (x, y) offsets into our Gaussian kernel. We compute these Gaussian weights for each offset and then normalize them. Each offset along with its corresponding Gaussian weight is sent to the shader. Once in the fragment shader, we need only loop through the values given. For each iteration, we add the offset to the current fragment location in screen space and sample the distance map at that point. We then multiple it by the normalized Gaussian value and add this to our cumulative total. After the loop, this total is written back into the distance map as the new shadow value.

The final pass renders the scene normally and uses the modified, previously binary, shadow value from the distance map with 0 meaning the fragment is lit, 1 meaning the fragment is part of the umbra region, and values in between specifying the intensity of the penumbra region. The following describes, using pseudocode and figures, the process of the algorithm.

3.2.1.1 Original Scene. Figure 3.3 shows the original scene we will be calculating shadows for. This is a part of the Pool scene, one of our custom scenes whose properties are stated in Table 2.1.

Figure 3.3: Original scene without shadows.

3.2.1.2 Pass 1. Figure 3.4 shows the results of the first pass which is simply rendering to the shadow map. This figure shows the actual depth information collected. This information will be important for the next pass.

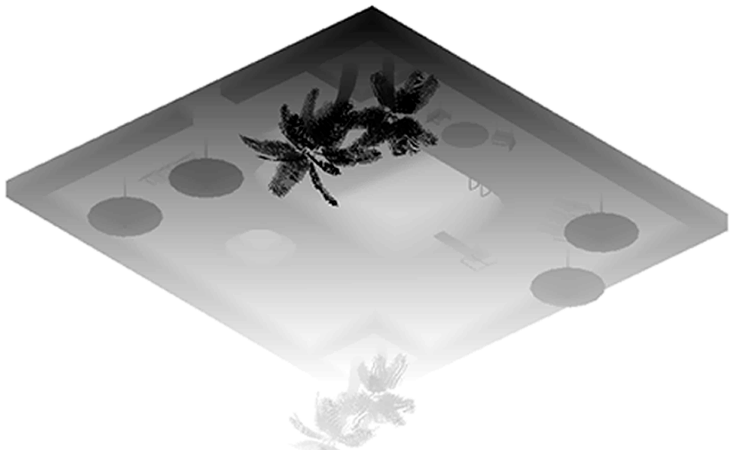
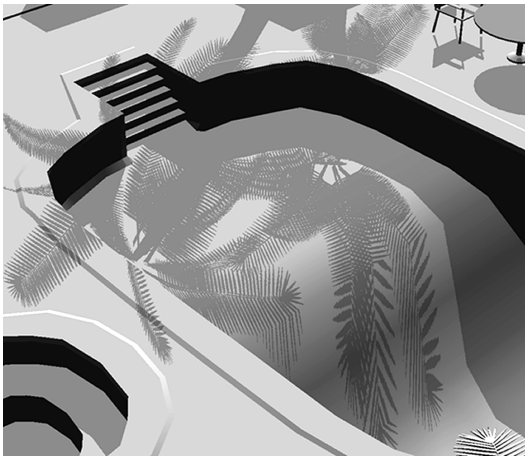


Figure 3.4: Depth based shadow map.

3.2.1.3 Pass 2. In this pass we utilize the information from pass 1. Figure 3.5 shows S and D after executing pass 2. The steps involved in pass 2 are as follows:

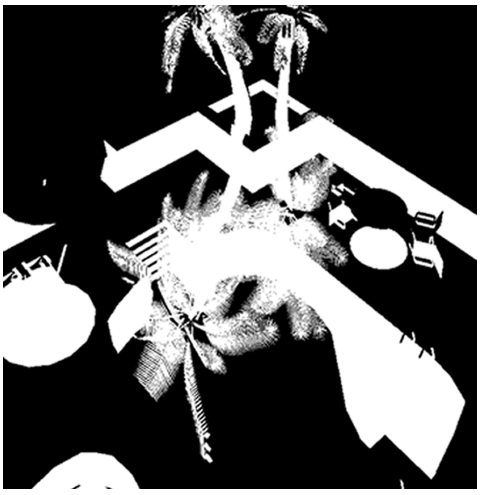
- For each fragment
  - Compare fragment against shadow map
  - Let S be 0 if fragment is lit, 1 otherwise
  - Let D be (occluder distance) $^{Dp} \cdot Dm$, where

    - Dp is the distance power
    - Dm is the distance multiplier
- Write S into a channel in the distance map
- Write D into a different channel in the distance map



(a) Hard Shadows

(b) D



(c) S

Figure 3.5: (a) Hard shadows. (b) D, which is a map of occluder distances. (c) S, which is a binary map showing umbra regions.

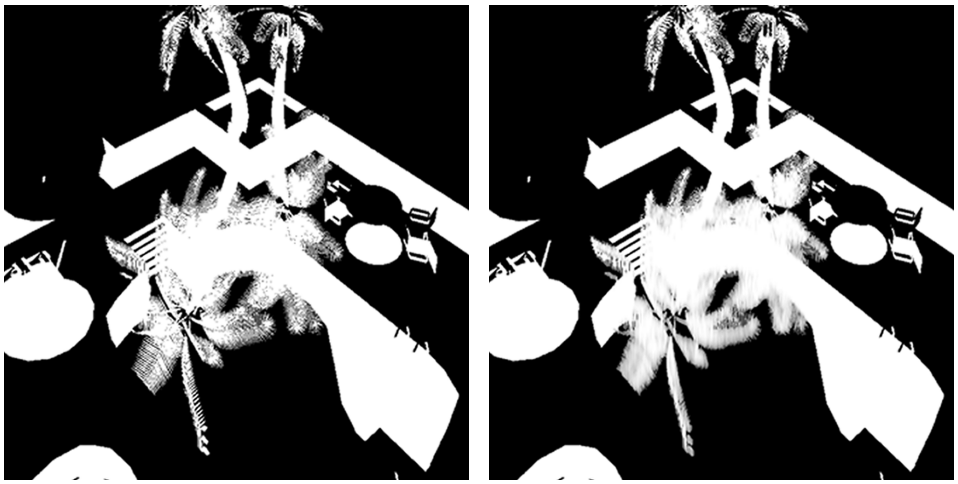
3.2.1.4 Pass 3 (Optional). Figure 3.6 shows the result of pass 3, our dilation pass. This pass is optional and is used to create an outer penumbra region since the basic steps of the algorithm only modify penumbra intensities of those fragment already in shadow. This means that without this step, the algorithm will create inner penumbrae regions but not outer.

- Dilate all D values using a Gaussian kernel
- Write new values back into the distance map

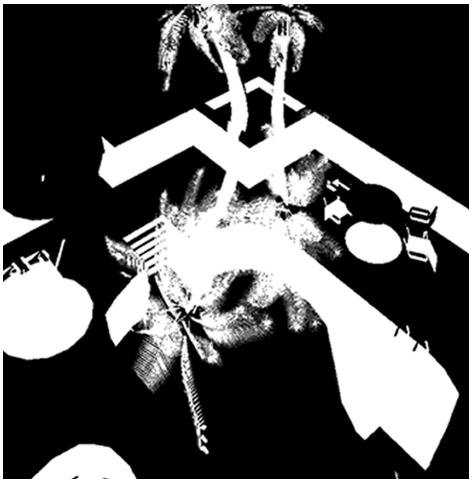
Figure 3.6: D after dilation

3.2.1.5 Pass 4. Figure 3.7 shows all three alterations of pass 4. The alterations are as follows: Choose one of the following and do for each fragment, then write results back into the distance map:

- MPG PCF
  - Apply PCF to S value using D to control kernel size
  - Write new S value
- MPG SP
  - Apply Gaussian filter to S value using D to control kernel size
  - Write new S value
- MPG MP
  - For each pass up to a specified limit, if $D > 0$ do
    - Let KS be the kernel size, which is either fixed or progressively widening with each pass
    - Let P be the set of precomputed 2D Poisson disk points we wish to use
    - For each P_i in P
      - Calculate the Gaussian weight
      - Multiply P_i by KS to get the texture offset
    - Normalize all Gaussian weights
    - Send texture offset and normalized Gaussian weight for all P_i to the shader
    - Render using sent data to apply the Gaussian filter to blur all S values
    - Decrement all D values
    - Write new S value
    - Write new D value



(a) MPG PCF (b) MPG SP

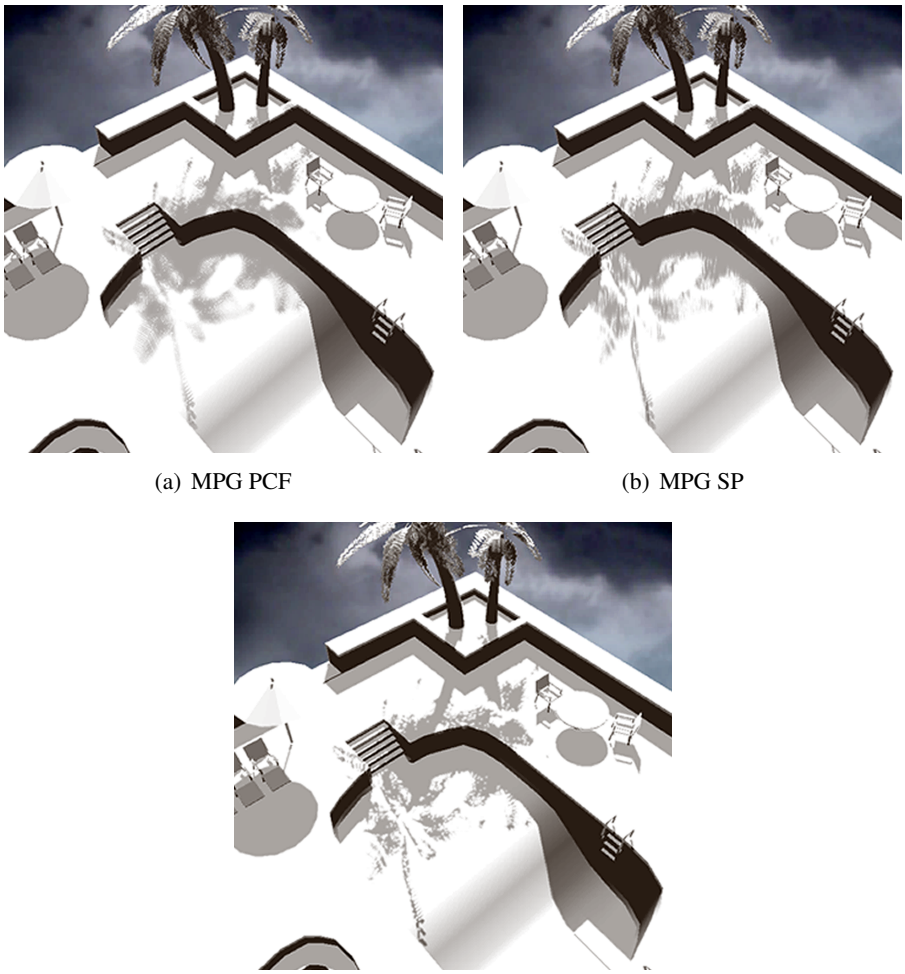


(c) MPG MP

Figure 3.7: Map, D, after applying either the (a) PCF, (b) Single Pass, or (c) Multi Pass approach.

3.2.1.6 Pass 5. Figure 3.8 shows the final rendering and final results of the algorithm for all three alterations. The steps involved in the final pass are as follows:

- Render scene normally
- Read from the distance map
- If $S = 0$
  - Fragment is lit
- Else if $S = 1$
  - Fragment is in shadow
- Else
  - fragment is in penumbra with intensity S



(a) MPG PCF (b) MPG SP

(c) MPG MP

Figure 3.8: Final result of either (a) PCF, (b) Single Pass, or (c) Multi Pass approach.

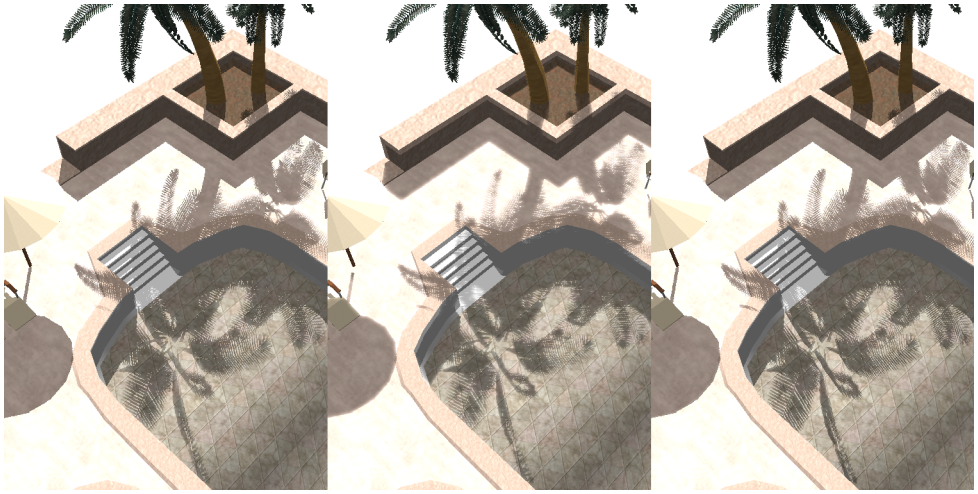
3.2.2 Results

We now show the results of executing our MPG algorithm and other soft shadow techniques on the different custom scenes in our app. Recall that scene descriptions can be found in Figure 2.1 in Section 2.5. Since the main objective of our MPG algorithm is better realism through contact-hardening soft shadows and not better utilization of map density, each shadow map was given a resolution of 2048x2048 so as to mostly eliminate aliasing artifacts. When comparing algorithms, each algorithm was given parameters that resulted in the best results for that scene.

In Figure 3.9 we visually compare the results of the different approaches on the Pool scene. We compare the PCF, single Gaussian pass, and multiple Gaussian pass approaches from our algorithm. The palm tree leaves are simply quads with a texture using the alpha channel to denote the areas between the leaves.

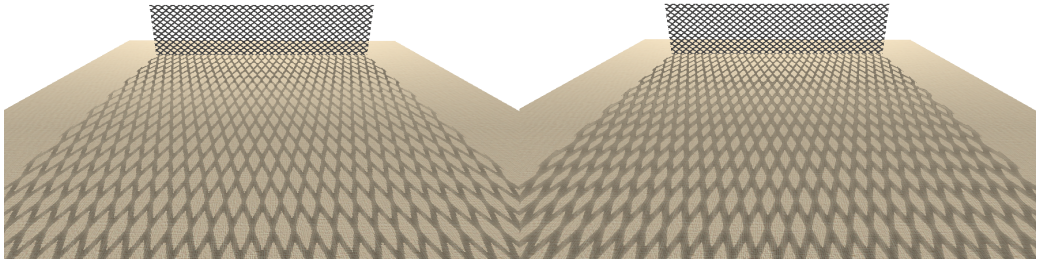
Figure 3.10 shows the effects of the dilate pass. We used the PCF approach on the Fence scene to show how using the dilate pass may not always be best depending on the scene. In this case, the outer penumbra created by the dilate pass causes the shadow to be overly blurry and thicker than they should be. Also in areas with dense thin shadow lines, the dilate pass can cause the lines to merge, thereby losing some of their detail. Sometimes this can be desirable and lead to realistic results, so each scene should be examined

carefully before deciding whether to use this pass.



This shows the difference between the three approaches. All approaches use a 2048x2048 map. (left) Using PCF as the filter, (center) using single pass Gaussian as the filter, (right) using multi-pass Gaussian as the filter. The PCF approach appears too sharp in many areas and has leaves where the needles seem to go from light to dark causing a disturbing pattern. The single pass Gaussian is too blurred in many areas and as such it looses a lot of detail. The multi-pass approach shows the detail in the leaves and still allows for a soft blur around the needles.

Figure 3.9: Difference in approaches

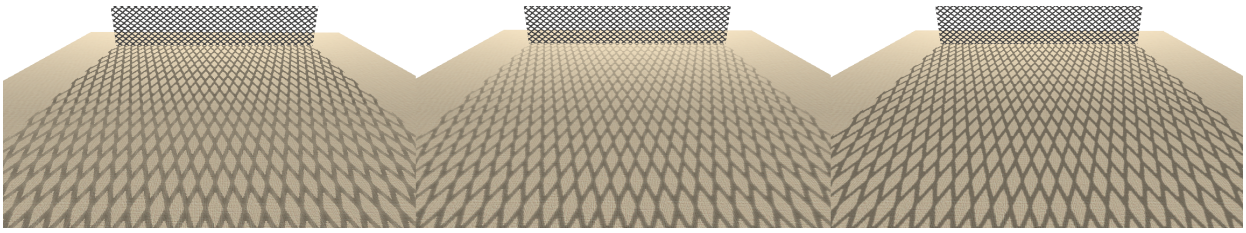


Our PCF approach without (left) and with (right) a pre-filter dilation over the distance map. This shows that sometimes Pass 3 from the algorithm is harmful to the final result. In the above case we have unrealistically thick lines in the shadow. Therefore Pass 3 should be omitted in this scene. The specific scene and approach must be considered before deciding whether to use the dilation in Pass 3 or not.

Figure 3.10: Effects of dilation

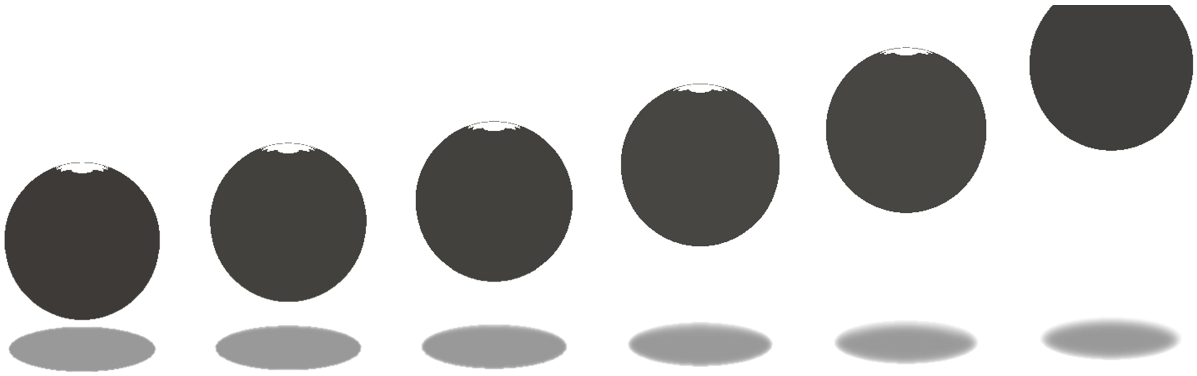
We examine the Fence scene again in Figure 3.11. Here we show all three approaches of our algorithm. In the PCF approach the shadow transitions too harshly from hard to soft shadows and this causes a noticeable change in shadow intensity. The single pass approach is too light at the base of the fence. The multi-pass approach combines consistent shadow intensity with a smooth transition from hard to soft shadows.

Figure 3.12 shows how the penumbra grows along with the occluder distance. The scene is shown with the penumbra clearly marked with a dark gradient. The umbra portion is inside of the gradient.



This shows the difference between the three approaches. (left) Using PCF as the filter, (center) using single pass Gaussian as the filter, (right) using multi-pass Gaussian as the filter.

Figure 3.11: Difference in approaches



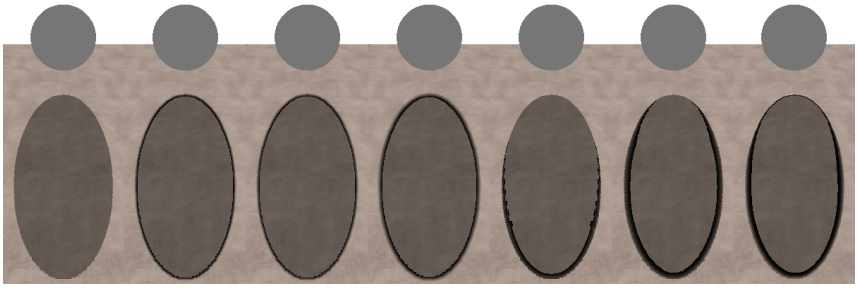
Here we see as the ball gets higher in the air and therefore further from the receiver, its penumbra region grows.

Figure 3.12: Penumbrae regions

In Figure 3.13 we show the effects of multiple algorithms on a simple elongated shadow cast in the Ball scene. From left to right we examine hard shadows, uniform Gaussian blur soft shadows, PCF with two different settings, and our contact-hardening approach with PCF, then single pass Gaussian, and finally our multi-pass Gaussian approach. This shows a comparison between hard shadows, uniform soft shadows, and contact-hardening soft shadows. The image has a false coloring applied to it such that the penumbrae regions are more clearly visible. The inner gray area is the umbra region. The darker gradiant outside of the umbra region is the penumbra region.

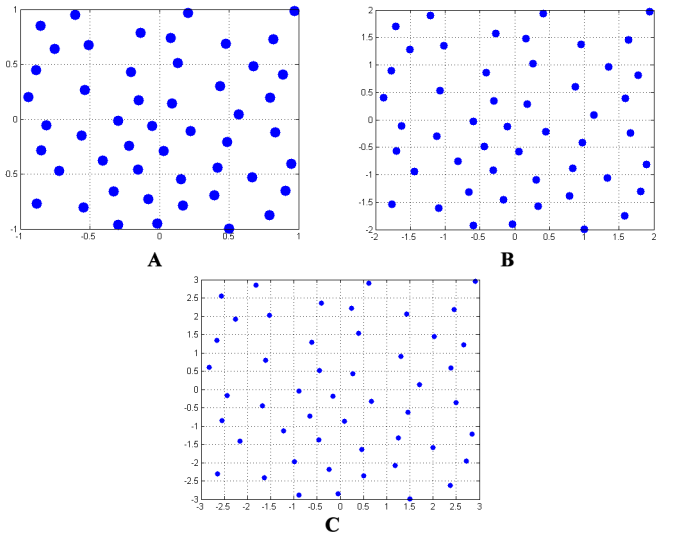
We examine the effects of changing the number of Poisson points used to sample our Gaussian kernel in Figure 3.14. For small kernels less points are needed to sufficiently cover the area and provide an adequate approximation. As the kernel increases in size, more points are needed to keep a decent approximation. In the figure we look at three representative kernel sizes of 3x3, 5x5 and 7x7. We use 50 Poisson points for each kernel size to show the coverage of a set number of points over the different sizes.

In Figure 3.15 we further examine the effects that the number of Poisson points have. Here we use a 29x29 Gaussian kernel and varying the number of points from 1 to 20. Even with a small number of points on such a large kernel, the results quickly approximate the effects of sampling the entire Gaussian.



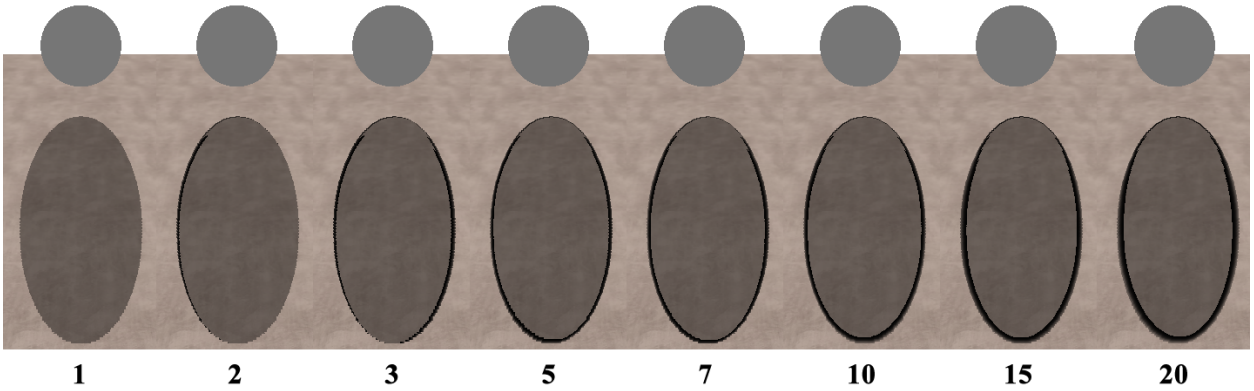
We have coded in the shader a special penumbra type that clearly distinguishes umbrae from penumbrae regions. The dark lines indicate penumbra and the shaded portion within is the umbra. This figure shows the comparison of several algorithms using the Ball scene. The ball's shadow is elongate and the penumbra region is clearly marked as in Figure 4 so as to show the effects of the contact-hardening algorithms. A) Normal hard shadows. B) Uniform Gaussian blur using kernel size of 5x5. C) Uniform PCF with kernel size of 5x5. D) Uniform PCF with kernel size of 7x7. E) Our own PCSS-like approach with no dilation and dynamic kernel size ranging from 1x1 (i.e. no filter) up to 19x19. Dp was set to 3.0 and Dm was set to 200. F) Our own single pass Gaussian approach without dilation and dynamic kernel size ranging from 1x1 up to 19x19. Dp was set to 1.9 and Dm was 200. G) Our own multi-pass Gaussian approach without dilation and a maximum of 10 Gaussian passes. Each successive pass increased the kernel size starting from 3x3 and going up to 19x19 with 20 Poisson taps. Dp was 3.0 and Dm was 200. The contact-hardening approaches in E, F and G are more realistic than the uniform soft shadow approaches. The multi-pass approach in G achieves the best results and uses the smallest number of taps among the contact-hardening approaches.

Figure 3.13: Algorithm comparison



This shows the coverage of 50 Poisson points in a Gaussian kernel. As the size of the kernel increases, the space between the points increase and coverage becomes more sparse. The number of points need to be chosen such that the total coverage is sufficient. One can use less taps than that of a separable Gaussian approach and still receive satisfactory results. A) Kernel size of 3. Points go from -1 to +1. B) Kernel size of 5. C) Kernel size of 7.

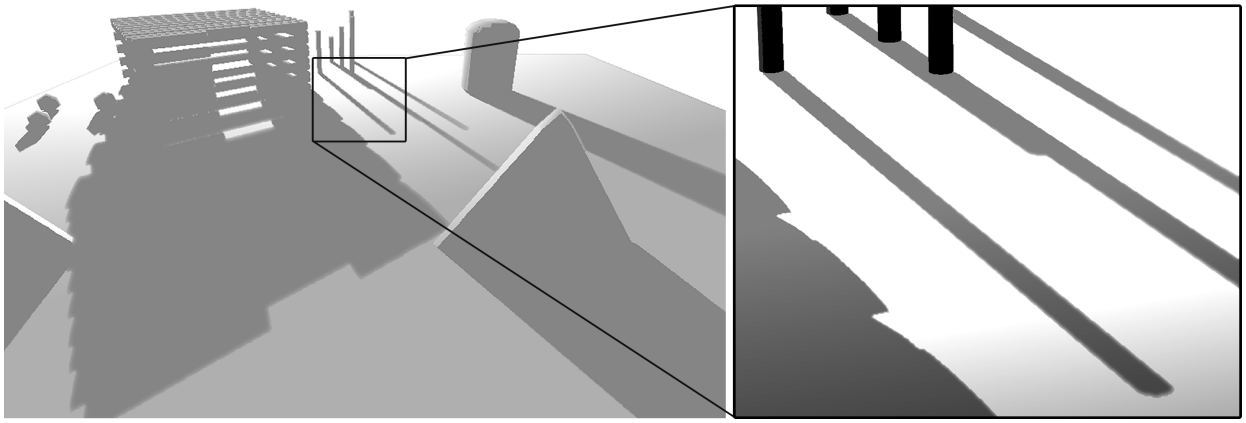
Figure 3.14: Poisson points



This is our multi-pass approach with 10 passes and varying amounts of Poisson points used for a 29x29 Gaussian kernel. The number of points used is under each image. We fix the first point at the origin, allowing the first sample to be at the center.

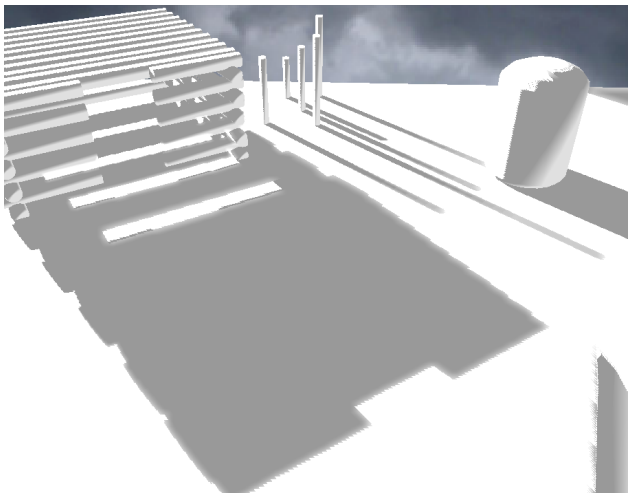
Figure 3.15: Number of Poisson points

Figure 3.16 shows our Log Cabin scene without textures to better show the shadows. We can see in the figure on the left the full scene with our multi-pass Gaussian shadows. Notice how the shadows become progressively more blurred as they move away from the occluder. This is especially apparent in the log cabin. In the same figure on the right is a close up of the wooden poles by the cabin. One can see from the poles the transition from hard to soft shadows. Some intensity normalization was applied to the figure to better show the shadows on the poles. This same scene with all three alterations of our MPG algorithm can be seen in Figure 3.17. This figure allows one to compare the approaches and see the differences they make in the soft shadows they generate. Different parameters were used in this image than that of Figure 3.16.

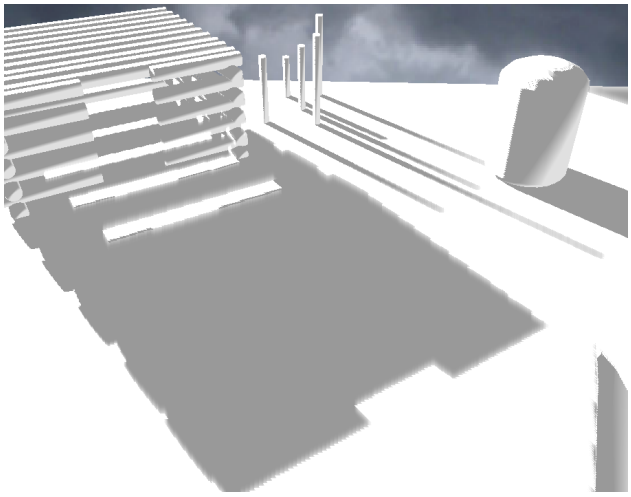


(Left) Our multi-pass approach with a 4096x4096 shadow map shown without textures for better shadow clarity. (Right) Closeup of the multi-pass Gaussian showing the transition from hard to soft shadows.

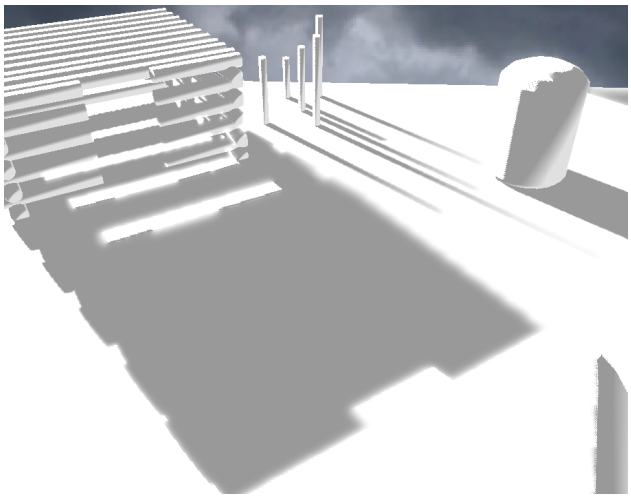
Figure 3.16: Multi-pass approach



(a) MPG PCF



(b) MPG SP



(c) MPG MP

Figure 3.17: Results of using MPG on the Log Cabin scene

3.2.3 Discussion

Our first approach, using PCF as a filter, is similar to PCSS and produces decent results. Our second approach, using a Gaussian filter in a single pass, is similar to SSSS. Our third approach, using a Gaussian filter in multiple passes, produces the best results in our test scenes. The fastest of these is the single pass approach, however if the max number of Gaussian passes in the multi-pass approach is limited to small numbers, it too can achieve interactive speeds. Since the number of Poisson taps is constant and independent of kernel size, we can use bigger kernels at faster speeds than a separable Gaussian approach. Kernel size can start small and increase with each pass thereby making it even faster but still achieving great results. There are two main parts to any penumbra, the inner and the outer penumbra. The inner penumbra is the part that would be an umbra fragment in a hard shadow algorithm but instead has its color brightened to represent a part of the penumbra. The outer penumbra is the part that would be lit and right next to the umbra in a hard shadow algorithm but instead has its color darkened to represent a part of the penumbra. In the multi-pass algorithm, the dilate pass mostly controls the outer penumbra, as it mainly causes the hard shadow edges to grow outward. The multiple Gaussian passes after this control the inner umbra, as they only lighten the color of already shadowed pixels, i.e. lit pixels are untouched. It is important to know this distinction so one can make informed decisions about such aspects as choosing a proper kernel size for the dilate pass and choosing the max kernel size for the Gaussian multi-pass. There is an optimization for the multi-pass approach whereby another pass is added right before the dilate pass. This pass examines a neighbor around each pixel, that is at least as big as the max kernel size for the Gaussian pass, and inspects each neighbor's S value. If all neighbors, including the center pixel, are in shadow, the D value for the center pixel is changed to zero. The purpose of this is to detect pixels that are deep inside the umbra of a shadow and therefore should be skipped during all Gaussian blur passes. This greatly increases the speed of the algorithm, however this has side effects. Since the Gaussian passes are responsible for creating the inner penumbra, if a pixel is chosen that would have normally been blurred by later passes, then that pixel becomes immune to any blur passes. This means the pixels closest to the hard shadow edge that are chosen to be part of the inner umbra and have their D values set to zero, are the pixels where the inner penumbra cannot grow past. Therefore a kernel bigger than the max kernel size must be chosen for this pass in order to not restrict how deep the inner penumbra grows. As the kernel size grows for this pass, the time savings diminish. Care must be taken when using this optimization. Since this is a post-process algorithm, other algorithms can be combined to achieve greater results. For example, our objective is not to increase nor better utilize shadow map density. Therefore techniques such as Cascaded Shadow Maps [16], Light Space Perspective Shadow Map [5], Sample Distribution Shadow Maps [17], and others can be used to greatly reduce aliasing before applying our algorithm.

CHAPTER 4. HORIZON ENCODING SHADOWS

4.1 Introduction

4.1.1 Horizon Walk Shadows

The initial idea for determining whether a fragment was in shadow or not was to start at the fragment's object space location (interpolated from the vertex shader) and walk towards the light (i.e. parallel but in the opposite direction of the light's rays). At predetermined distance intervals, we could check the displacement map. We would figure out where we were at that point in object space and figure out which location in the displacement map that would map to. Retrieving the value here, we could get the radius from the moon's center. Comparing the radius to what the horizon line's trajectory is, we know if the radius is greater, than this location is blocking the light from our original fragment. If not, we can continue walking along the horizon line, redoing the check at each distance interval. After so many checks (or a set distance) has been reached, if no occluders were found, we would conclude that this point was lit. If an occluder was found, we would stop the walk at that point and conclude the fragment to be in shadow. Using the distance from the fragment to the first occluder, we could even generate contact-hardening soft shadows. The greater the distance from the occluder, the greater the softness of the penumbra. This approach was called the Horizon Walk Shadow algorithm. While the idea itself is not inherently flawed, the execution came with multiple problems. Checking distance at set intervals posed a risk since too short of an interval meant we were potentially making too many taps into the displacement map, thereby slowing the algorithm. If we make have too large of an interval, we might miss an occluder. There is also the decision of how many total checks to make or how much the total distance is that we should walk before ending the search. We created this algorithm and made options for it in the GUI to test each of these parameters (refer to Figure 4.1). After testing this algorithm out and trying to get some results with it, we realized there was a bigger issue that hadn't been realized yet. Due to the way SCM works, it is impossible to check all the locations from a given spot on the moon to an arbitrary distance along the horizon. This is because that information is not cached and sent to the GPU . Basically only the pages that are visible are drawn, and the resolution of each page varies given the proximity to the viewer. Also, due to the hierarchical draw calls, we might not have either complete information or any information at all about a particular region on the moon that we are walking over. In short the data we require is not present at the time we require it, making this algorithm impossible in its current state. The theory behind this approach showed promise, but the execution had to be changed.

4.1.2 Horizon Encoding Shadows

Since the information is all there in the full displacement map TIFF but not fully present in any single render of a frame, we must grab this information in a construction or off-line phase and somehow store it for use during the on-line rendering. This means we needed to come up with a series of pre-processing steps to gather the information we need. After gathering this information, we needed to run our calculations and then store the results. The storage medium had to be sufficiently quick to query so as to maintain real-time speeds of the application. These were the goals that the Horizon Encoding Shadows (HES) algorithm was created to solve. In the new few section we will go over the steps performed in the pre-processing phase and explain how the data gathered and calculated is stored and queried so as to generated shadows.

4.2 Cache Generation

This step uses OpenGL's transform feedback to capture the displaced vertices in each of several renders of different parts of the moon. The general idea is to write code to capture each vertex's object space position

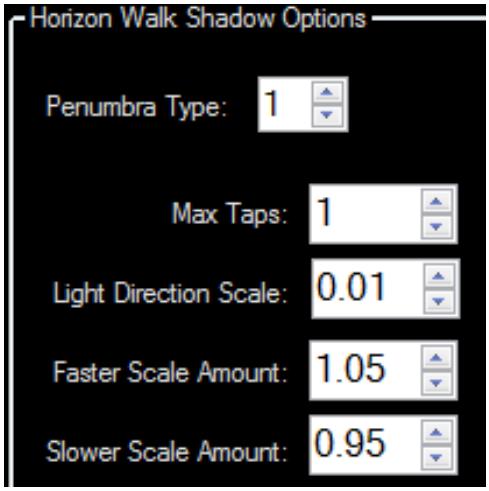
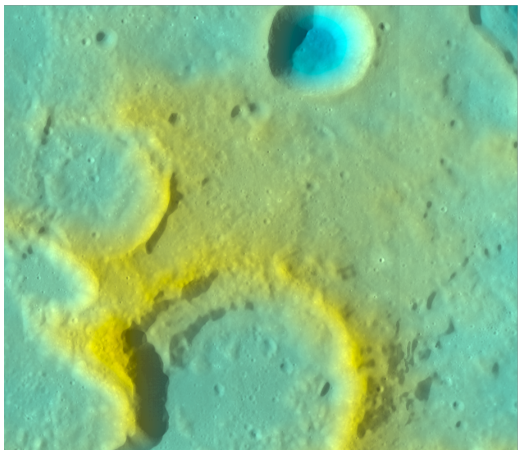


Figure 4.1: Options for Horizon Walk Shadows

and normal in a single render and organize this data by each vertex's SCM depth. The moon is rotated by a set amount along each axis and the transformed vertices of each render is saved into a file.

4.2.1 Creation

Before we process any vertices on the moon, we need to capture a list of those we want to process. We capture their surface normal and their displaced position in object space. Algorithms 4.2.1, 4.2.2, 4.2.3 show this procedure. Algorithm 4.2.1 renders moon and uses OpenGL's transform feedback to capture information (i.e. send from GPU back to CPU) about each vertex rendered, including displaced position, normal vector and SCM depth. This information is stored in variable *verts*. *verts* is then parsed and an array of vertex properties, called *data*, is created. *data* is then written to a file with the prefix "frame-". Since capturing vertex information over the entire moon would take awhile to complete, we only gather information from a small patch. Figure 4.2 shows the colored elevation map of this patch.



The patch we gather information on is located at a longitude range of $14.03°E$ to $17.62°E$ and latitude range of $1.95°S$ to $4.80°S$.

Figure 4.2: Moon patch used

4.2.1.1 Creating for a Patch. This is used for capturing vertices in a small area on the moon. It works best when SCM depth is 4 or deeper.

Algorithm 4.2.1 Renders Moon, Captures Vertices Rendered, and Writes to File

```
1: function RENDERANDCAPTURE( )
2:     verts ← RENDERANDRETURNVERTICES()
3:     data ← PARSEVERTICES(verts)
4:     WRITECACHEDATATOFILE(data)
5: end function
```

Algorithm 4.2.2 Algorithm for capturing a patch

```
1: function CREATEFORPATCH( )
2:     RENDERANDCAPTURE( )                ▷ Moon's rotation is already set in GUI
3: end function
```

4.2.1.2 Creating for Entire Moon. The moon is rotated to different latitude and longitude locations and uses Algorithm 4.2.1 and the function RenderAndCapture to capture information about each location.

Algorithm 4.2.3 Algorithm for capturing entire moon

```
1: function CREATEFORENTIREMOON( )
2:     for lat ← −90 to 90 step rotationIncrement do
3:         for lon ← −180 to 180 step rotationIncrement do
4:             moonrotation ← EULERANGLES(lat, lon, 0)
5:             RENDERANDCAPTURE( )
6:         end for
7:     end for
8: end function
```

4.2.2 Consolidation

The vertex data in each file is read and consolidate into a single file for each SCM depth. The information in each depth file is then sorted by each vertex's normal. The sorting is important so in the next stage we can remove any duplicate vertices we find. The final output of this stage is a single file with unique vertices for each SCM depth.

4.2.2.1 Combining. First we need to gather the vertex information captured in all the cache files. We create file handles to write to each consolidated depth file and read each cache file in one at a time. Each cache file can have information from multiple depths, so we go through each vertex and append its information to the appropriate depth file. This stage produces one file for each depth with a file name of "C-*depth*.txt" where *depth* is the SCM depth (e.g. "C-0.txt", "C-1.txt", etc.). This is shown in Figure 4.3 and Algorithm 4.2.4

4.2.2.2 Sorting. After we have combined the information from all our cache files into single depth files, we need to sort the vertices in each depth file. We can use a simple string sort and sort the lines alphabetically. This is important so we can later efficiently remove any duplicated entries. It is possible that the

Algorithm 4.2.4 Combines Cache Entries

```
1: function CombineCache( )
2:     for d ← 0 to maxDepth do
3:         streamWriters[d] ← StreamWriterFromFile(CacheLocation(d))
4:     end for
5:     for all cacheFile in cacheFiles do
6:         collection ← CreateCollectionFromFile(cacheFile)
7:         for d ← 0 to maxDepth do
8:             for all entry in collection do
9:                 WriteLineToFile(streamWriters[d], entry)
10:            end for
11:        end for
12:    end for
13: end function
```

same vertex may have been captured multiple times in different cache files, therefore we need to remove the duplicates. Without sorting first, the fastest we could remove duplicates would be $O(n^2)$. Sorting in this stage is $O(n * log(n))$ and this allows the next stage, removing duplicates, to be linear, meaning the entire time for both stages is $O(n + n * log(n))$ which is still linearithmic and faster than quadratic for the unsorted removing of duplicates. This stage takes files of the form "C-*depth*.txt" as input and produces files of the form "CS-*depth*.txt" as output (the "C" is for combined and the "S" is for sorted). This is shown in Figure 4.3 and Algorithm 4.2.5.

Algorithm 4.2.5 Sorts Combined Cache Entries

```
1: function SortCache( )
2:     for d ← 0 to maxDepth do
3:         if FileSize(CacheConsolidatedFile(d)) < threshold then      ▷ file can fit in memory
4:             lines ← ReadAllLines(CacheConsolidatedFile(d))
5:             lines ← Sort(lines)                                      ▷ linearithmic
6:             WriteAllLines(CacheSortFile(d), lines)
7:         else                                  ▷ file is too big, need external disk sort
8:             ExternalDiskSort(d)
9:         end if
10:    end for
11: end function
```

4.2.2.3 Removing Duplicates. Since we have sorted all the entries, we only need to create a file handle for the output file, read each line in the input file, and detect whether the current line is the same as the previous line. If it is the same, we continue on to the next line, writing nothing. If the current line is different, we write it to the output file. This step consumes files of the form "CS-*depth*.txt" and produces files of the form "CSR-*depth*.txt" and is shown in Figure 4.3 and Algorithm 4.2.7.

4.2.3 Updated Patch-Centric System

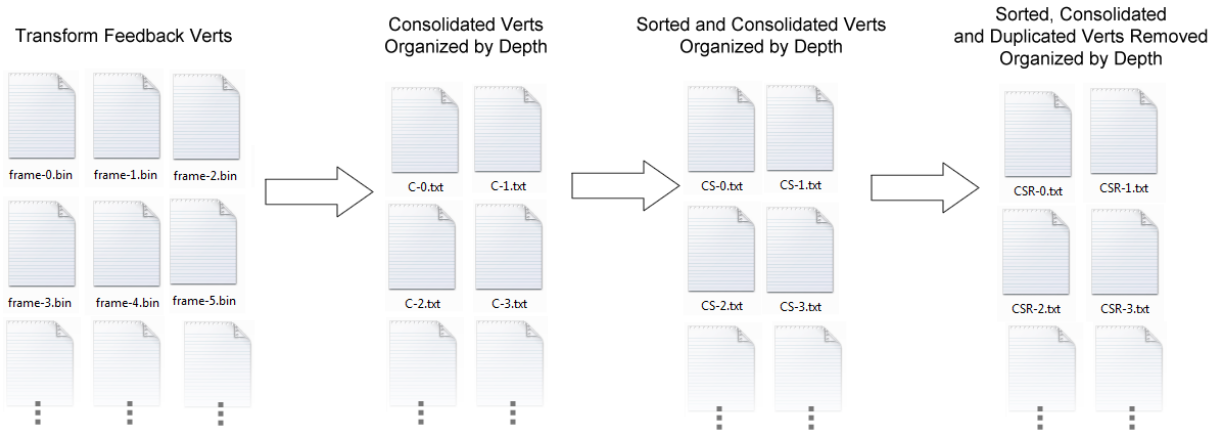
The previously stated way of collecting cache has several downfalls. For example, it is difficult to receive only the vertices for a specific depth and patch. If you sample more than once, you could receive several

Algorithm 4.2.6 Performs External Sort on Combined Cache Entries

```
1: function EXTERNALDISKSORT(d)                ▷ This is basically external merge sort, i.e. linearithmic
2:     inputFile ← STREAMREADERFROMFILE(CACHECONSOLIDATEDFILE(d))
3:     chunk[ ] ← READNLINES(inputFile, chunkSizeLines)        ▷ read the first chunk into string array
4:     chunk ← SORTSTRINGARRAY(chunk)
5:     tempWriter ← STREAMWRITERFROMFILE(TEMPFILE(0))
6:     for i ← 0 to chunkSizeLines do
7:         WRITELINE(tempWriter, chunk[i])
8:     end for
9:     tempReader ← STREAMREADERFROMFILE(TEMPFILE(0))
10:    tempWriter ← STREAMWRITERFROMFILE(TEMPFILE(1))
11:    readerId ← 0
12:    repeat
13:        chunk[ ] ← READNLINES(inputFile, chunkSizeLines)        ▷ read the next chunk
14:        chunks ← SORT(chunk)
15:        c ← 0
16:        templine ← READNEXTLINE(tempReader)
17:        while c < chunkSizeLines & templine ≠ null do
18:            if STRCOMPARE(chunk[c], templine) < 0 then
19:                WRITELINETOFILE(tempWriter, chunk[c])
20:                c ← c + 1
21:            else
22:                WRITELINETOFILE(tempWriter, templine)
23:                templine ← READNEXTLINE(tempReader)
24:            end if
25:        end while
26:        while c < LENGTH(chunk) do
27:            WRITELINETOFILE(tempWriter, chunk[c])
28:            c ← c + 1
29:        end while
30:        while templine ≠ null do
31:            WRITELINETOFILE(tempWriter, templine)
32:            templine ← READNEXTLINE(tempReader)
33:        end while
34:        if readerId = 0 then
35:            readerId ← 1
36:        else
37:            readerId ← 0
38:        end if
39:        tempReader ← STREAMREADERFROMFILE(TEMPFILE(readerId))
40:        tempWriter ← STREAMWRITERFROMFILE(TEMPFILE(1 − readerId))
41:    until ENDOFFILE(inputFile)
42:    RENAMEFILE(TEMPFILE(readerId), CACHESORTFILE(d))
43:    DELETEFILE(TEMPFILE(1 − readerId))
44: end function
```

Algorithm 4.2.7 Removes Duplicate Cache Entries

```
1:  function REMOVEDUPLICATECACHE( )
2:      for d ← 0 to maxDepth do
3:          CLEARFILECONTENTS(REMDUPCACHEFILE(d))
4:          lastLine ← EMPTYSTRING()
5:          lastNorm ← NEWVECTOR3(0,0,0)
6:          for all line in LINESINFILE(SORTEDCACHEFILE(d)) do
7:              norm ← PARSELINE(line).normal
8:              if line ≠ lastLine & norm ≠ lastNorm then
9:                  APPENDLINETOFILE(REMDUPCACHEFILE(d), line)
10:             end if
11:             lastLine ← line
12:             lastNorm ← norm
13:         end for
14:     end for
15: end function
```

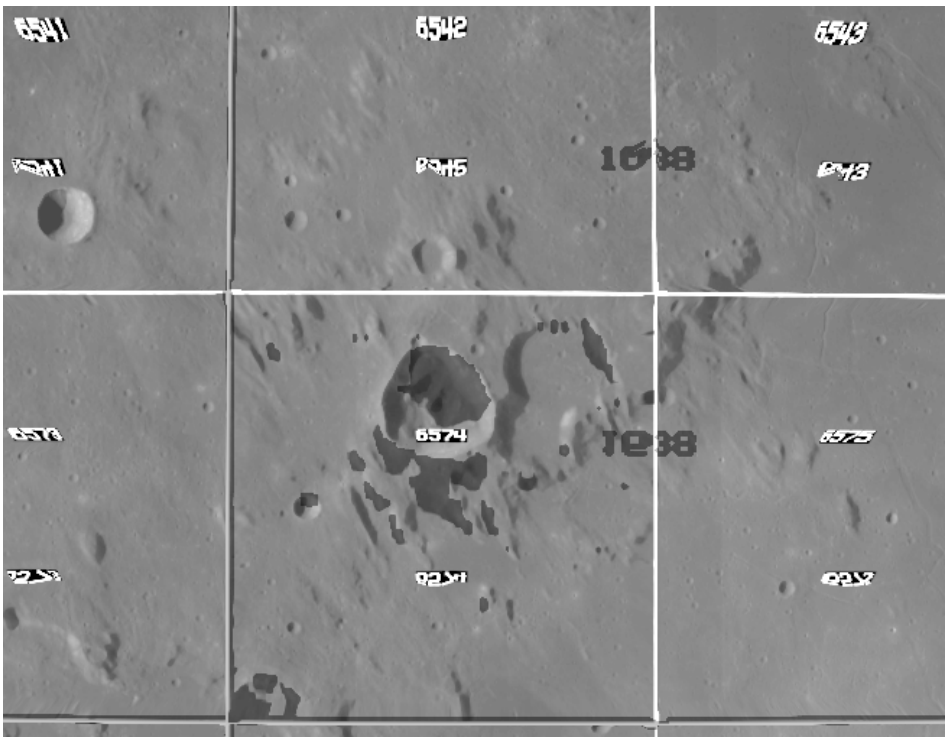


Conversion of vertex capture files into cache files (sorted and with duplicates removed)

Figure 4.3: Captured Vertices to Cache Files

duplicate vertices, hence the need for sorting and removing duplicates. It is also difficult to completely capture all the vertices of a patch and holes can appear in one's data. These reasons sparked a new approach to collecting cache. Cache is now collected by specifying a patch id and an added depth value. The patch id can be retrieved by turning on indices in the GUI (see Figure 4.4). Once a patch is specified, the depth at which that patch appears is calculated and the added depth parameter is added in. If added depth is zero, then only the patch itself is cached. If added depth is one, then the patch's four child patches are cached, since with each new depth, four child patches are created. In general the number of patches cached are 4^{added_depth} and the final depth of all cached patches is the depth of the specified patch plus the added depth. Each patch is cached by using the SCM framework on the CPU side and simulating increasing the depth so as to subdivide that patch into 256x256 sub patches. Each sub patch is sampled at the center to retrieve a normal vector for that sub patch. In total 256x256 (or 65,536) normal vectors are calculated for each patch. This number is the result of the fact that the SCM TIFF files used to generate the moon's topology are each 256x256 for each patch. Therefore the sampling directly matches the resolution. Since the calculation happens on the CPU and is focused only on the specific patch, we can be sure to gather all the vertices of the

patch and nothing else. This approach can also be faster since using OpenGL's transform feedback is costly and this new approach completely does away with the need for transform feedback. The result of this new approach is a single cache.txt file generated for each patch. Later on when creating the final SCM TIFFs , we will use the mipmapping option to create information for each more shallow deep automatically without the need to create separate cache for them. The following section on horizon angle calculation assumes the older caching method with transform feedback, so will refer to files generated by this older method. The process is the same however, as this new caching method ultimately only affects what vertices are cached. Therefore the only real change to the horizon angle calculation phase is the fact that there exists only a single file called cache.txt and not multiple cache files. This is only a minor change and doesn't affect the overall angle calculation process.



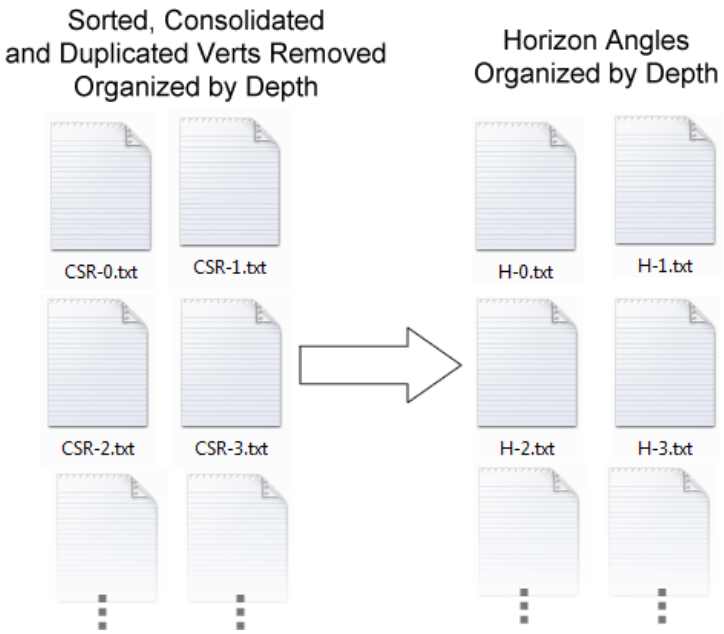
Moon with patch indices being displayed. The patches displayed from left to right are 6541, 6542, 6543 and 6573, 6574 and 6575. This is used to identify the specific region we are interested in and specify that to the cache generation process. Patch 6574 has been processed and is displaying shadows. The faded gray display of the number 1638 is an artifact of the shadow algorithm and is displaying the parent patch's index. This artifact only appears when indices are shown, which is allowed to be toggled in the GUI .

Figure 4.4: Patch Indices Shown on Moon

4.3 Horizon Angle Calculation

Now that we have a single file describing each vertex we want to use, organized by depth, we can begin to encode the horizon for each. This is the heart of the algorithm and is where the majority of the work happens. The basic idea is to move the view to the specific location of the vertex, orient it to look forward, then rotate it about the normal of the vertex to capture all 360 degrees. At each degree, the goal is to find the highest point visible and encode the angle of elevation from the vertex to that high point. This is our

horizon angle for that orientation and azimuth degree. Capturing all of these gives us a signal of 360 values and adequately describes the horizon all around that point. Figure 4.5 shows the file conversion from cache files to horizon angle files. This part of the algorithm was changed a couple of times in an attempt to find the fastest implementation. Since this part can be a bottleneck to the entire pre-process algorithm, it was important to improve upon it as much as possible.



Conversion of cache files (sorted and duplicates removed) into horizon angle files.

Figure 4.5: Cache to Horizon Angle Files

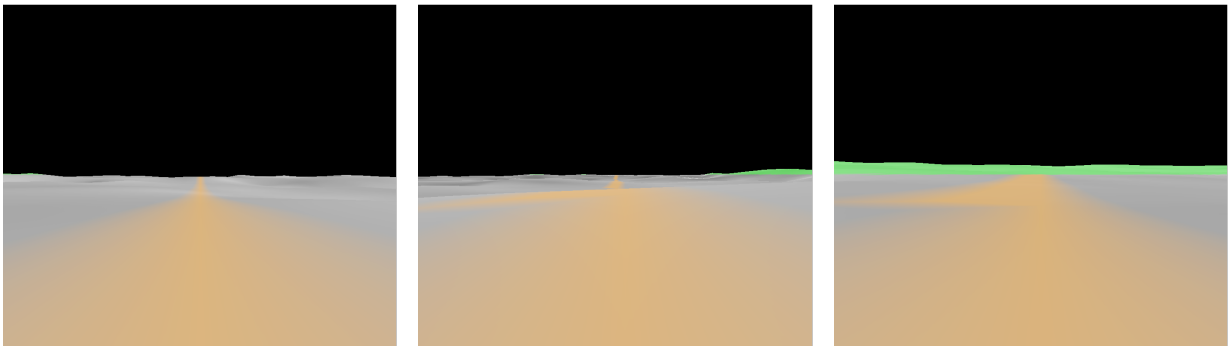
4.3.1 Method 1 - CPU Only

When we first started capturing horizon data, we had the idea of going through each vertex in the file and checking it against all other vertices in the file to see if any of them would be part of that vertex's horizon. A model view matrix was constructed for the each vertex and applied, on the CPU side, to each other vertex in the file, thereby projecting each vertex onto a space where the vertex in check is the origin. We only tested files that could fit in main memory, so we were able to load the entire contents into a data structure. Each vertex was checked against all other vertices 360 times (one check through all vertices per angle). So if N is the number of vertices in the file, we were performing 360*n*n operations on the CPU to process a single file (i.e. $O(n^2)$). If a single file contained 100,000 vertices, this means performing an order of 3,600,000,000,000 matrix operations and comparisons for that file. The process took around 22 mins for a single vertex in a file containing around 5,000 to 6,000 vertices. This would equate to 4.19 years for a file of 100,000 vertices. At this slow speed, it wasn't necessary to add code to handle the case when a file was too big to fit in main memory, as we knew the speed would be even slower. We knew we needed a faster approach.

4.3.2 Method 2 - GPU Object Space

Method 1 was restructured to instead perform a rendering at each vertex and use the results of the render to calculate the horizon angle. This means going from 360*n*n comparisons to 360*n renders (i.e. from a

quadratic number of comparisons to a linear number of renders). Transform feedback was originally used to capture all vertices rendered. We would then look for the vertices that were along the line in front of the camera. We would take the vertex with the greatest y value to be the horizon point and calculate the angle from that. The problem was that very few vertices were actually directly in front of the camera (i.e. they had x values exactly equal to zero). We therefore expanded what the acceptable range of x values were, allowing a vertex to be a little to the left or right of the camera. This was slow and inaccurate. Slow because of having to loop through all vertices rendered to find the right ones and inaccurate because of only looking at vertices, instead of fragments, especially when some vertices weren't even right in front of the camera. Using this approach, it took around 22 minutes to get all horizon angles for a single point. Figure 4.6 shows the process of finding these horizon angles. To increase speed and accuracy, this was changed to grabbing data in image space, which became Method 3.



The orange line traveling outwards from the camera denotes the current angle we are examining. The green areas are possible peaks that could form our horizon line. When the orange line meets a vertex colored green, it is checked to see if it is the highest vertex seen so far. After checking all green vertices for that azimuth angle, the one with the greatest elevation is stored as the peak and the horizon angle is calculated from there.

Figure 4.6: Horizon Search Line

4.3.3 Method 3 - GPU Image Space

Instead of using transform feedback on the vertices, we used read pixels on the fragments (the rough equivalent of transform feedback but with fragments). Now we could find the highest point out of only those points directly in front of the camera. We would go from top to bottom of the returned image looking for the first non-black pixel.

Algorithm 4.3.2 shows the procedure for this method. We basically want to create our fbo and decide our starting and stopping points for our azimuth light angles. We then process caches at all depths and open each cache file. For each cache file, we read one vertex at a time and initialize a zeroed out array of horizon angles for it. We then render our views capturing the eye space positions of each pixel. Algorithm 4.3.1 shows how we setup the render and process the returned information. We use the function in Algorithm 4.3.3 to set up our transformation matrix. We then bind our fbo, render the scene and collect the pixel information from the buffer. Using this information we find the first non-black pixel and use some geometry to calculate the angle given that pixel's eye space location. This is repeated for each column of pixels in the buffer. After this we simply assign our horizon angles to the angle array and write the vertex along with the entire contents of the horizon angle array to our output file.

Algorithm 4.3.1 Calculates Horizon Angles for Given Point

```
function GETHORIZANGLES(d, lightRot, fbo, point)
    VIEWTRANSFORM(point.vertex, point.normal, lightRot)
    BINDFBO(fbo)
    pixels ← RENDERANDREADPIXELS()
    for c ← 0 to fbo.width do
        maxPoint ← MAKEVECTOR3(0,0,0)
        rowFound ← −1
        pts[c] ← 0
        for r ← fbo.height − 1 to 0 step −1 do
            if pixels[r,c] ≠ MAKEVECTOR3(0,0,0) then
                maxPoint ← pixels[r,c]
                rowFound ← r
                break
            end if
        end for
        if rowFound ≠ −1 then
            pts[c] ← ASIN(maxPoint.Y / VECTORMAGNITUDE(maxPoint))
        end if
    end for
    return pts
end function
```

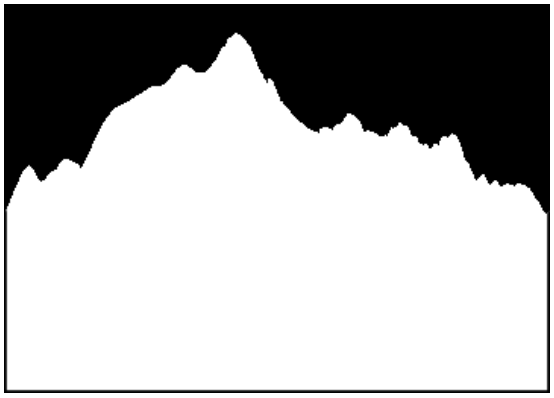
Algorithm 4.3.2 Horizon Encoding Method 3

```
function HORIZANGLEGPUIMG( )
    fbo ← BUILDFBO()
    lightAngleStart ← FLOOR(fbo.width/2.0)
    lightAngleEnd ← 360 + lightAngleStart
    for d ← 0 to maxDepth do
        inputFile ← OPENFILE(REMDUPCACHEFILE(d))
        repeat
            line ← READLINE(inputFile)
            point ← PARSELINE(line)
            normal ← point.normal
            horizAngles ← NEWFLOATARRAY(360)
            rotIter ← 0
            for lightRot ← lightAngleStart to lightAngleEnd step fbo.width do
                angles ← GETHORIZANGLES(d, lightRot, fbo, point)
                for li ← 0 to LENGTH(angles) do
                    horizAngles[rotIter + li] ← angles[li]
                end for
                rotIter ← rotIter + fbo.width
            end for
            WRITETOFILE(HORIZFILEPATH(d),horizAngles)
        until ENDOFFILE(inputFile)
    end for
end function
```

Figure 4.7 shows an example of the results returned from reading the framebuffer pixels. This greatly improved both accuracy and speed, however further improvements had to be made. Due to the scale of the amount of vertices we needed data for, for a small 100,000 vertex patch, each increase of 10 ms in speed per vertex results in over 15 minutes decrease in total time. To further improve our time, the perspective field of view was shortened to a single degree, so only what was necessary for that horizon angle would be rendered. Also, we created a feature that allowed us to turn off all transform feedback since it was no longer necessary for this phase. We performed calculations to allow for a field-of-view of 90 degrees so we could render 4 times instead of 360. We also allowed multiple CPU threads to help in issuing GPU commands and processing results. These changes allowed a reduction in time per vertex. Figure 4.8 shows the results of calculating horizon angles using method 3.

Algorithm 4.3.3 View Transform

function VIEWTRANSFORM($vertex, normal, lightRot$)
$lat \leftarrow$ ASIN$(normal.y) * (180.0/PI)$
$lon \leftarrow$ ATAN2$(-normal.x, normal.z) * (180.0/PI)$
$ground \leftarrow$ VECTORMAGNITUDE$(vertex) + lift$
GLMATRIXMODE($GL\_MODELVIEW$)
GLLOADIDENTITY( )
GLTRANSLATE$(0, -ground, 0)$
GLROTATE$(-90, 1, 0, 0)$
GLROTATE$(lightRot, 0, 0, 1)$
GLROTATE$(lat, 1, 0, 0)$
GLROTATE$(lon, 0, 1, 0)$
end function

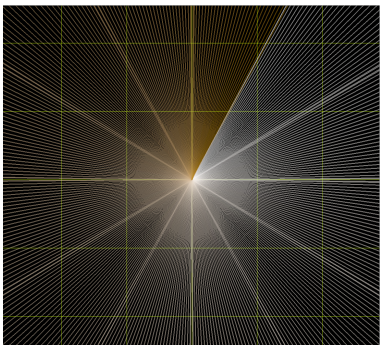


An example of rendering the moon's topology from a vertex. A simple scan of this image can tell us which fragment has the highest object space elevation. Colors have been clamped for rendering this image and therefore appears as black and white even though the actual values held in the pixels have eye space coordinates of each fragment.

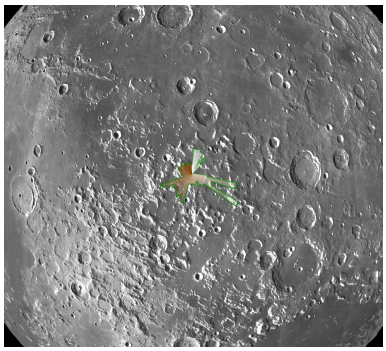
Figure 4.7: Horizon Read Pixels

In Figure 4.8, each line is one azimuth degree rotated from the lines next to it. The lines go from the center of the vertex to the highest peak along that direction. We can see from this figure just how closely the horizon lines match the topology, indicating the strength and accuracy of our calculations. We can,

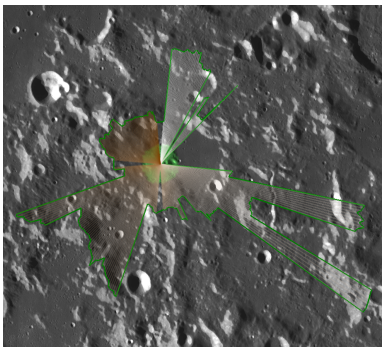
however, be off by a small amount and still have convincing shadows. Some parameters allow us to trade this accuracy for speed. These parameters will be discussed in Section 4.3.4.



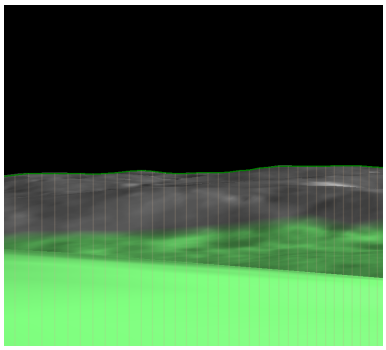
(a) Wireframe Showing Vertices



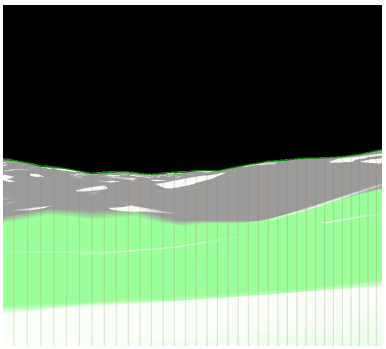
(b) Zoomed Out



(c) Zoomed In



(d) On the Surface



(e) On the Surface

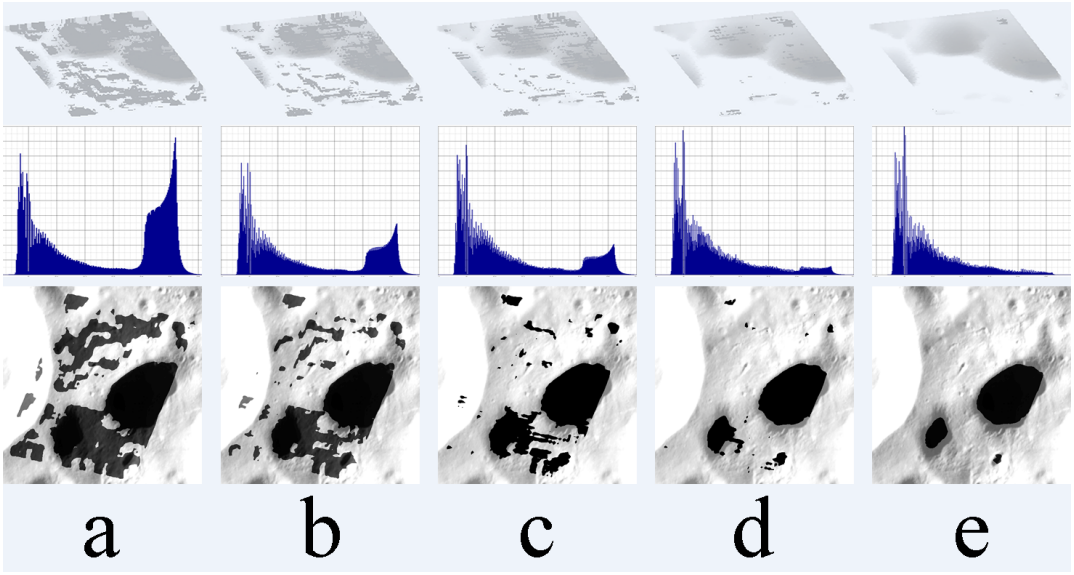
Each line is one azimuth degree. (a) shows a wire frame representation where the intersections denote vertices. (b) and (c) shows the vertex location relative to the entire moon and to the local surrounding area, respectively. (d) and (e) show the view when standing on the surface at the vertex.

Figure 4.8: Viewing Horizon Lines

4.3.4 Trade-offs and Optimizations for Method 3

Certain factors can affect the accuracy of the horizon calculation phase. This phase also takes the most time of all the phases. As stated small increases in time per vertex equate to large increases overall. It is therefore important to explain the different accuracy trade-offs and speed optimizations used and considered.

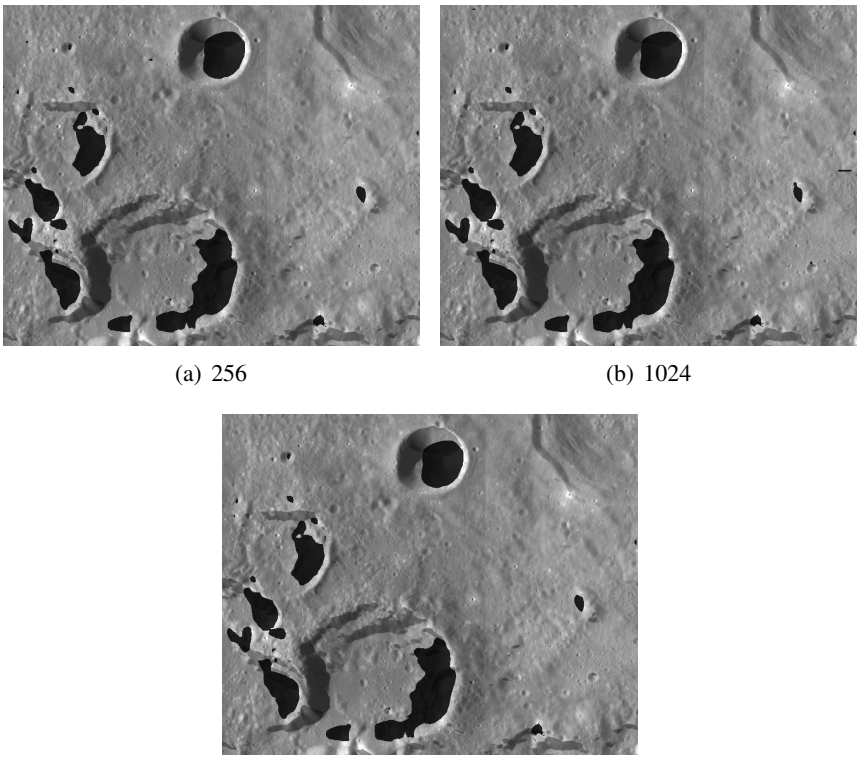
4.3.4.1 Lift from Surface. When adjusting the camera to be at the specific location of a vertex, there are problems that arise. Points on the moon have their radius from the moon's center adjusted according to the depth they are at. At depth 0 vertices are given an initial radius and this radius is refined with each depth. If one creates cache at, say, depth 5, then when calculating the horizon points, the camera will be moved to the exact location at depth 5. SCM 's internal topology cache is then forced to fully build up to the deepest depth that would be displayed for each point on the surface. If the location of the camera, i.e. the location of the vertex at depth 5, changes because of small differences between that location at depth 5 and the same location at a deeper depth, then the camera could potentially be stuck inside of the moon. Small differences added on from the contributions of, say, depths 6, 7 and 8 could raise the vertex's location and leave the camera to be underneath the surface. It is also possible because of near clipping plane distances and highly rocky areas of the terrain, that the camera can find itself in the same situation even if the cache was generated at the deepest depth. It is for these reasons that we must add a small "lift" value to the camera's position, thereby increasing its radius from the moon's center, before calculating horizon angles. In some situations, this is not necessary, however due to the reasons stated, it cannot be guaranteed. Figure 4.9 shows an example of the implications of adding a lift amount to the camera's position. Note that too much of a lift leads to inaccuracies as it misrepresents the true location of the vertices. However, on such a scale as the moon, raising all points in the patch should cause minimum error overall as long as lift values are sensible. Our findings indicate that values between 16 and 20 seem to be ideal.



Comparison of different values for the lift variable. Lift has values of (a) 0, (b) 2, (c) 4, (d) 8, and (e) 16. At the top we have the input TIFF generated from the horizon angles. One can see the artifacts created by low lift values. In those areas the camera was too low to the ground or even underneath the moon's surface, causing a high horizon angle value. In the middle of the figure, we see a histogram of the horizon angles over the patch. The first peak represents values close to zero, which should be the majority of angles. The second peak indicates values close to 0.5 radians. This second peak shows the frequency of artifacts as angles in practice should rarely be this large. Notice how higher lift values lower the frequency of these artifacts. The bottom row shows the shadows created by each. Discussed in greater depth later on in the paper, the faded dark regions that are part of the underlying texture represent actual shadows apparent in the data. These are used as a form of ground truth. Examining the HES shadows, we see that artifacts erroneously show fragments in shadow.

Figure 4.9: Comparison of Lift Values

4.3.4.2 FBO. The off screen framebuffer is rendered to and pixels are read from it. By increasing the height of the framebuffer, greater resolution is used to better figure out the exact location of the highest peak we can see at each angle. For smaller values, e.g. 256, several world space locations will be rendered to the same image space location, and the fragment that writes last may not be the highest point in world space. For larger values, e.g. 4096, less world space locations will collide with the same image space point, and therefore the fragment that gets written is closer to the actual peak. This, however, also means there is more information to transfer from the GPU to the CPU during a read pixels operation. Therefore there exists a trade-off with the framebuffer height between accuracy and time, with higher values taking more time but with more accurate results. Figure 4.10 shows the shadows produce by an fbo height of 256, 1024 and 4096. Figure 4.11 shows the comparison among these heights using the technique in Figure 4.8 to view the horizon angles for a single vertex.

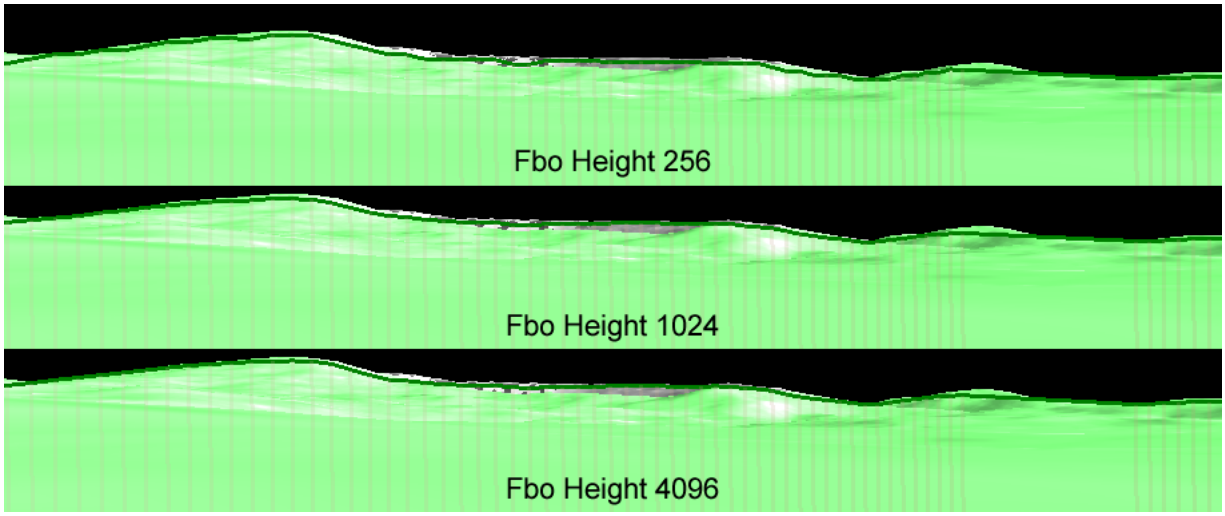


(a) 256 (b) 1024

(c) 4096

Figure 4.10: Effects of changing fbo height during horizon calculation. Fovx set to 90

4.3.4.3 Fovx. Since rendering is the bottleneck, we want to reduce the total number of renders needed per point. Originally there were 360 renders each capturing information about a single angle. The view would be rotated a single degree each time. If we reduce the perspective viewing angle to one degree along the x-axis, each render doesn't take too much time. We can however get the most out of a single render by increasing the number of angles we gather data on. If we increase our perspective viewing angle to 90 degrees and rotate our view 90 degrees each time, we can capture information on 90 angles at a time, requiring only four renders per point. This increases the time per rendering since there is more work for the GPU to do as well as more pixels to read back to the CPU , however since we drastically reduce the total number of renders by a factor of 90, the total time per point drops significantly.



Horizon lines drawn for an fbo height of 256 (top), 1024 (middle), and 4096 (bottom). The 4096 height lines match the true horizon closer than the 256 height lines, but not by a significant amount, showing that fbo height is not a huge factor in accuracy.

Figure 4.11: Comparison of Horizon Lines for Different Fbo Heights

4.3.4.4 Threading. Each vertex must be handled mostly in series since in the fastest method, Method 3, the bottleneck is GPU rendering. However the commands sent to the GPU and the work to process the results sent back can be made parallel. On the CPU side, each pass is independent and therefore multi-threading can be used. Mutex locks are used to ensure overwriting of data doesn't occur, e.g. overwriting pixel data with current processed frame while previous frame's contents are being analyzed. We do see some small speed gains overall by increasing the amount of CPU threading and with so many vertices to process, even small speed increases per vertex can amount to huge savings overall. A further optimization was later applied. Originally threading was performed by creating a new thread, giving it a vertex to process, starting the thread, having it run and produce data, then acquire a lock for writing to the output file before writing its data. The main thread was halted when all threads were spawned and started, and would resume when all threads finished and were destroyed. CPU thread creation and destruction does come with a cost and is not as negligible as GPU threads are. This along with having each thread acquire a lock to write to the output file, was costing a lot of time. Instead of this approach, we redesigned the system to allow threads to cover multiple vertices. Now each thread is created and given an array of vertices to process. Also, threads no longer write directly to the output file, but instead write to a buffer. Each thread has its own buffer so no locking mechanism is required. Buffers are flushed and written asynchronously to the file system after all threads complete their task and before the new batch of threads are created. Under the previous approach, timings per vertex for a fovx of 90 and fbo height of 256, would vary wildly and on average could be anywhere from 800 ms to 1,300 ms. After implementing these changes, timing deviations have become more restricted and now average between 450 ms to 600 ms.

4.3.4.5 Summary. Table 4.1 shows the average time per vertex when adjusting fbo height, fovx angle, and CPU threading using the older approach to threading. We can see by the table that fovx angle matters a great deal as this can dramatically reduce the number of renders which is the bottleneck in horizon calculation. CPU threading does help out some, although at low fovx angles, the amount of work done by a single thread does not outweigh the cost of thread allocation/deallocation, leading sometimes to worse results. Around a fovx of 60 degrees or higher, enough work becomes available on the CPU side to warrant threading, and

the addition of threads starts to pay off. Also around a fovx of 60 degrees or higher, fbo height shows a slow down with increased size. With fovx's of 1, 10, and 30, the total amount of pixels being read from the GPU and sent to the CPU each frame might not be enough to be affected noticeably (and consistently) by a change in fbo height. With 60 or more, the color buffer's width becomes large enough such that sufficient increases in height lead to a large enough increase in total buffer area, thereby causing an increase in the average time per vertex. At this point increases in fbo height reliably cause an increase in time. It is important to note that although these times where taken on the CPU side, due to the usage of OpenGL's readpixels function call, the GPU is forced to finish its render before allowing the CPU thread to continue (i.e. blocking the thread until the data is available to be read). Because of this the times should accurately reflect the amount of total work done per vertex on both the CPU and GPU side.

Table 4.1: Average time, in seconds, per vertex when executed over 50 vertices.

		Fbo Height		
Fovx	**Threading**	256	1024	4096
1	1	256.92	242.64	242.45
	2	241.61	265.78	286.60
	4	277.79	258.32	250.33
	8	265.18	269.59	271.89
10	1	26.95	19.41	19.51
	2	19.03	19.12	18.77
	4	19.31	19.28	18.86
	8	19.51	19.48	18.99
30	1	4.48	4.53	4.63
	2	4.48	4.32	4.49
	4	4.52	4.43	4.58
	8	4.46	4.43	4.53
60	1	1.82	1.85	1.90
	2	1.80	1.82	1.87
	4	1.77	1.79	1.86
	8	1.63	1.65	1.73
90	1	1.14	1.18	1.23
	2	1.11	1.14	1.22
	4	1.08	1.11	1.19
	8	0.98	1.02	1.09

4.4 Horizon Angle Storage

4.4.1 Generating Input Images

A latitude and longitude range is specified that covers just the patch that data has been calculated for in the horizon calculation phase. Each vertex's normal is transformed into a latitude and longitude location and is then converted into a pixel location in the input image. The full latitude range works with the height of the image as well as the longitude range with the image width to place vertex normals into pixel locations. The final input image is an equirectangular projection that covers the latitude and longitude range of the moon. The horizon angles are stored in the pixel color channels, 4 float values per pixel. Since each image only contains 4 channels (restrictions in GLSL texture access enforces this), we need multiple input images

to fully preserve the signal. This stage takes files of the form "H-*depth*.txt" and outputs files of the form "t*image_num*.tiff ". This is explained in Algorithm 4.4.1 and the file IO is shown in Figure 4.12.

Algorithm 4.4.1 Make Input Images

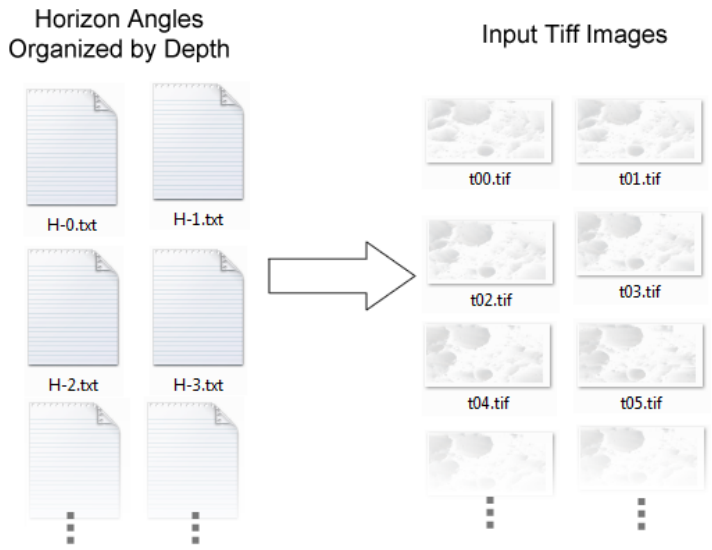
```
function MAKEINPUTIMAGES( )
    encodeType ← GETENCODINGTYPE()
    numScmTifs ← encodeType.numScmTifs
    scmTifs ← NEWARRAYSCMTIF(numScmTifs)
    fft ← PREPAREFFT()
    for d ← 0 to maxDepth do
        inputFile ← OPENFILE(HORIZFILEPATH(d))
        repeat
            line ← READLINE(inputFile)
            point ← PARSELINE(line)
            pos ← NORMALTOLATLON(point.normal)
            values ← GETEVERYXTHVALUE(point, encodeType.stride)
            if encodeType.useFft then
                values ← FFT(values)
                if encodeType.useKernel then
                    values ← APPLYKERNEL(values)
                end if
            end if
            for s ← 0 to numScmTifs do
                r ← values[s∗4+0]
                g ← values[s∗4+1]
                b ← values[s∗4+2]
                a ← values[s∗4+3]
                pixelColor ← NEWVECTOR4(r,g,b,a)
                ADDPIXEL(scmTifs[s], pixelColor, pos)
            end for
        until ENDOFFILE(inputFile)
    end for
    for i ← 0 to numScmTifs do
        WRITETOFILE("t" + i, scmTifs[i])
    end for
end function
```

4.4.2 Generating SCM TIFFs

After the input images have been created, a Windows batch file is also created that runs the scmtiff tool on all input TIFF images. This creates the output SCM TIFFs and then copies them over to the file location where they can be used. As an example of the commands run on one of the moon's horizon encoded patches, Algorithm 4.4.2. The file input/output process is shown in Figure 4.13.

4.4.3 Types of Storage Encodings

Storing every horizon angle would create too many output images. Therefore every x^{th} angle is used, creating a scaled down sample of the original signal. Interpolation is then used to reconstruct the missing



Conversion of horizon angle files to input TIFF image files

Figure 4.12: Input Images to SCM TIFFs

Algorithm 4.4.2 SCM TIFF Image Generation

```
index ← 0
for all inputFile in TIFFINPUTFILES( ) do
    scmtiff -pconvert -n256 -d4 -E14,17.8,-4.8,-1.8 -ocon.tiff inputFile
    scmtiff -pborder -obor.tiff con.tiff
    scmtiff -pfinish bor.tiff
    RENAMEFILE(bor.tiff , shadow$index$.tiff )
    index ← index + 1
end for
for all shadowFile in SHADOWINDEXFILES( ) do
    COPYFILE(shadowFile, publish-location/shadowFile)
end for
```

Batch file for generating SCM TIFF images from input TIFF images.

data. The following are different ways that data is arranged and encoded.

4.4.3.1 3 Stride. In this type every 3rd horizon angle is taken. We have 360 original values in our signal. Since we are taking every 3rd value, we only encode 120 values. Since each SCM can hold 4 values in each pixel, this means we need 30 SCM TIFFs for this type of encoding. Since each SCM requires a separate active texture unit, and since some cards have only 32 texture units available, this type of encoding can create a problem. Figure 4.14 shows what these input images look like.

4.4.3.2 4 Stride. For 4 Stride we encode every 4th value, requiring 23 SCM TIFFs . This gives us more room to use our texture units for other purposes and speeds up initial application startup by requiring less SCM TIFFs to load. The difference in accuracy between 3 Stride and 4 Stride seems empirically to be minimal in the input images and mostly unnoticeable in side-by-side comparisons (see Figure 4.18, which will be discussed later). Figure 4.15 shows 4 Stride images. Note how the direction of the light is evident in

Conversion of input TIFF images into SCM TIFF images

Figure 4.13: Input Images to SCM TIFFs

Input TIFF images for 3 stride encoding

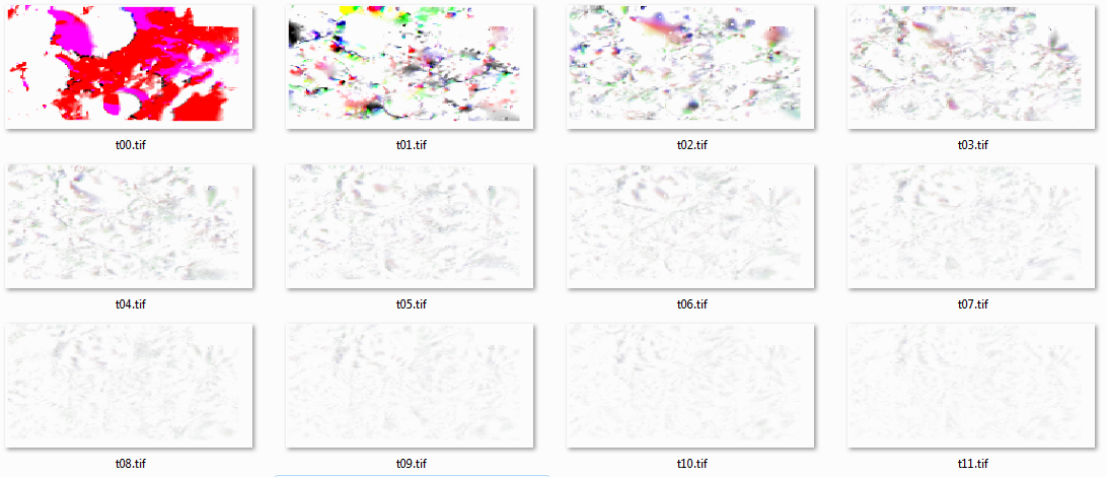
Figure 4.14: 3 Stride Input Images

these as well as the 3 Stride images. If one were to view all the images in order, almost as if it were frames in an animated gif, one would see the light's azimuth angle go from 0 degrees to 360 as the light's direction spins around the patch.

4.4.3.3 FFT 4 Stride. This is similar to 4 Stride in that every 4th value is taken. The difference being that the 90 value signal is then projected into frequency domain using Fourier analysis [37] [38]. This is done so the signal can be further compressed by eliminating some high frequency information. Visually inspecting the images this creates, as in Figure 4.16, we can see the coefficients trending towards zero, meaning each sequential input image stores less important information than the previous. At some point the data stored within an image can be safely discarded, along with all subsequent images. The signal is recreated by zero padding the eliminated data.

Input TIFF images for 4 stride encoding

Figure 4.15: 4 Stride Input Images



Input TIFF images for 4 stride, with Fourier analysis, encoding

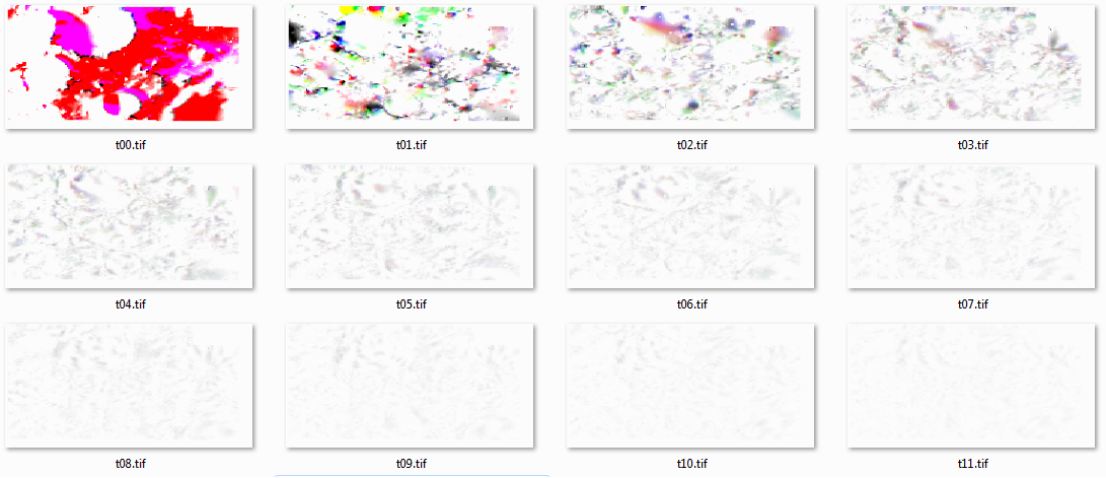
Figure 4.16: FFT 4 Stride Input Images

4.4.3.4 FFT 4 Stride with Kernel. Since cutting off a signal in the frequency domain can lead to artifacts when the signal is synthesized, a better approach is to taper the signal by applying a kernel. This can be seen in Figure 4.17. A Euclidean kernel is applied in this encoding to the result from FFT 4 Stride.

4.4.4 Storage Encoding Comparison

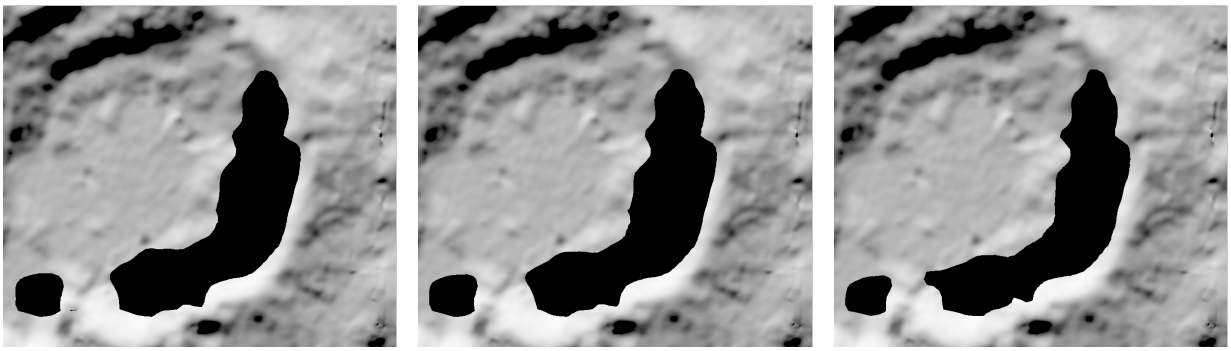
Figure 4.18 shows a visual comparison of the different types of storage encoding. Notice how there isn't a great amount of difference among them. As for storage space, the FFT types allow us to use the least amount of SCM TIFF files. Speed differences among the types will be discussed in Section 6.1.

4.4.4.1 Test Formats. Sometimes getting just the right input format for a tool is difficult. So to test if the input format was correct, testing formats were created. These include formats that only encode a single value and one that encodes the first 10 values. This was used during the initial stages to encode a series of



Input TIFF images for 4 stride, with Fourier analysis and kernel applied, encoding

Figure 4.17: FFT 4 Stride with Kernel Input Images



(a) 3 Stride (b) 4 Stride (c) 4 Stride FFT

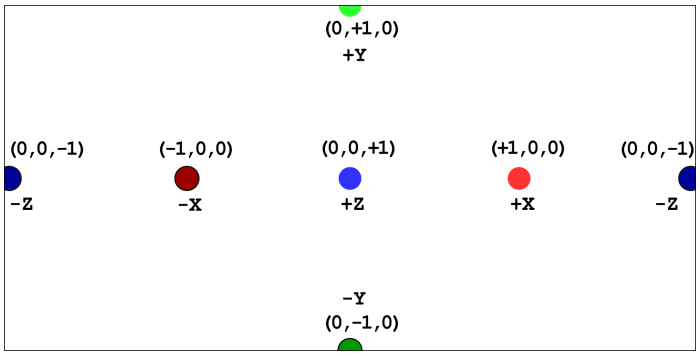
Images taken over a zoomed in area of patch from Figure 4.2. This shows the results of using (a) 3 Stride, (b) 4 Stride, and (c) FFT 4 Stride. Using FFT 4 Stride with Kernel would give the same results as in (c) when sampling from all SCM TIFF files.

Figure 4.18: Comparison of Storage Encoding Types

values, convert the input image into an SCM TIFF , then read back the value from the same vertex normal to ensure the value encoded was the same as the value read. We used the image found in Figure 4.19 along with the test formats to figure out how to properly encode our input images. The equirectangular projection sample also helped us figure out certain aspects such as the correct orientation of the x and z axes that we needed to have in our input images in order for the sampling to be correct. We then created more complex test cases.

4.4.5 Fourier Transform Choices

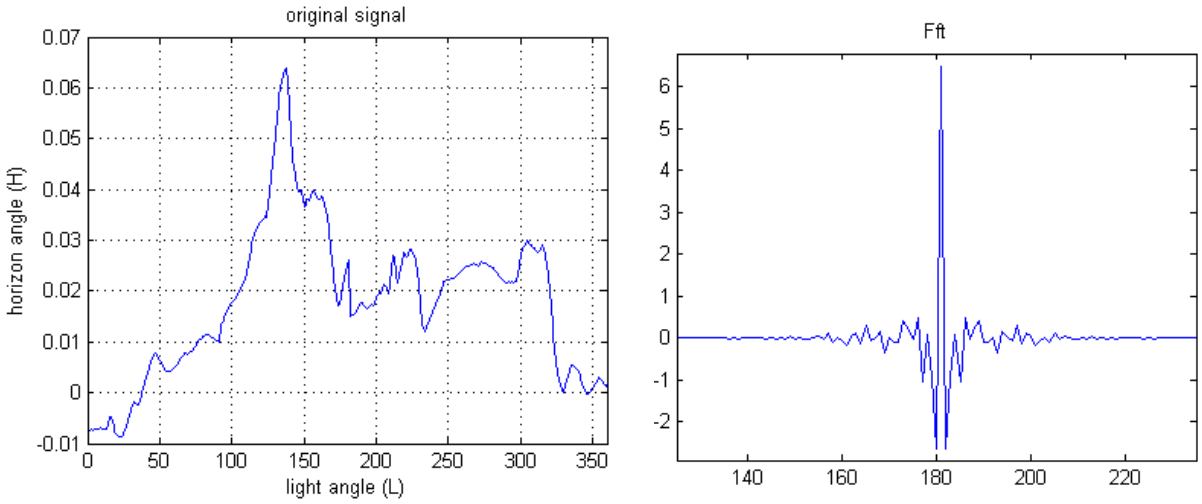
There are several choices when it comes to using the Fourier transform on the horizon angle data. One of the choices is what kernel to use to help compression without introducing so called ringing artifacts. This is described below. Another is the amount of bits needed to encode the data and which distribution to use, which is discussed in the Further Work section.



During the initial stages of implementation, this guideline was used as a reference for correctly encoding an equirectangular projection input image from the horizon angles calculated.

Figure 4.19: Guideline used for Equirectangular Projection Input Images

4.4.5.1 Kernels. Figure 4.20 shows a horizon angle signal and its Fourier domain representation. Kernels help to gradually taper off the frequency data so we can smoothly remove the last k values without suffering the effects of a sudden drop to zero. Given this k, if we were to apply a square or box kernel that is equal to one from zero to k and zero elsewhere (Figure 4.21), this would be equivalent to not having a kernel and simply setting to zero all frequency values from k+1 to the end of the signal. A better choice would be applying a linear kernel, Figure 4.22. This kernel linearly decreases from one to zero (at indices zero up to k), then remains zero from k+1 up to the end of the signal. This successfully tapers the signal but does so in a way that doesn't always give us the best results. We want to preserve the first several coefficients as much as possible then quickly reduce the values afterwards up to k. A linear kernel starts reducing right away and doesn't take into account the non-linear important of the signal values as we go from sample 0 to the end. One of the best kernels we can apply is a Gaussian kernel, Figure 4.23. This reduces the signal in a more desirable way. Figure 4.24 shows this kernel applied to our signal.

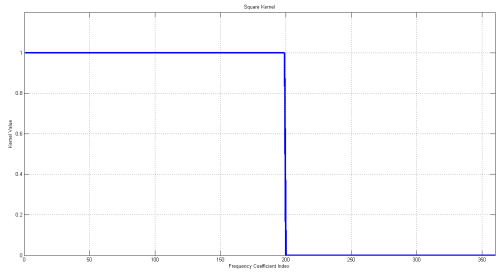


(a) Original Signal

(b) Frequency Representation

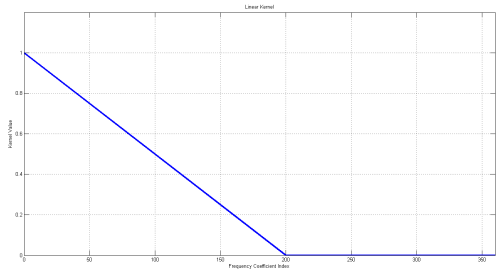
This shows (a) one of our actual 360 sample horizon angle signals and (b) its corresponding representation in Fourier domain (after a phase shift to center it at zero).

Figure 4.20: Example of Horizon Angle Signal and Fourier Representation



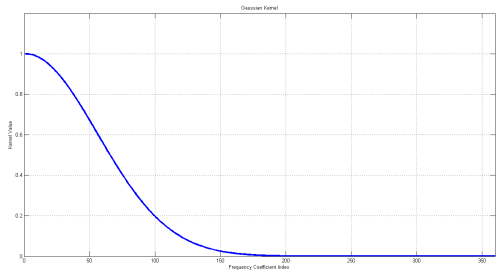
A square kernel stopping at index 200.

Figure 4.21: Square Kernel



A linear kernel stopping at index 200.

Figure 4.22: Linear Kernel

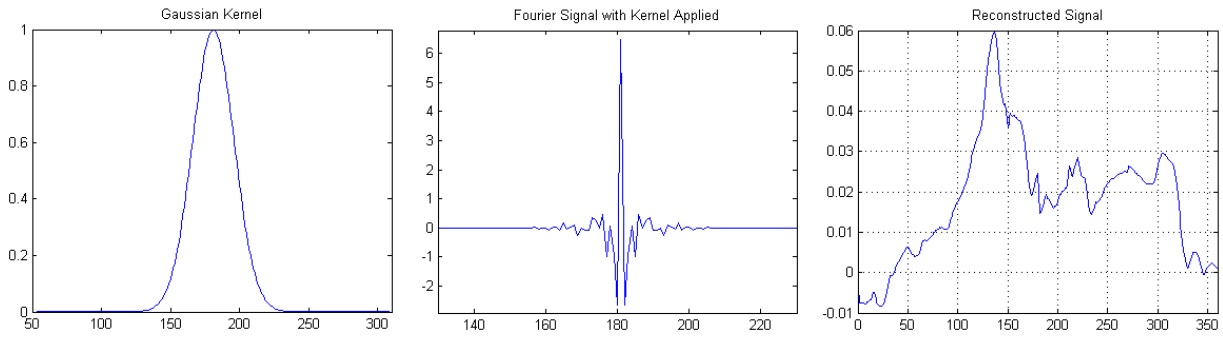


A Gaussian kernel stopping at index 200.

Figure 4.23: Gaussian Kernel

4.4.6 Compressing the Data

We have seen already that although choice of fbo height may technically increase/decrease the horizon angle accuracy, in practice this doesn't affect much. The shadows created by heights of 256, 1024 and 4096 do not differ much (Figure 4.10). This leads us to believe that a decent amount of accuracy can be lost without sacrificing visual integrity. Therefore, instead of saving a 32-bit floating point value as an indication of the horizon level, we could get away with much less. If we consider that an fbo with height equal to 256 can still produce quality shadows rivaling that of a height of 4096, then we only need 8 bits of precision to encode

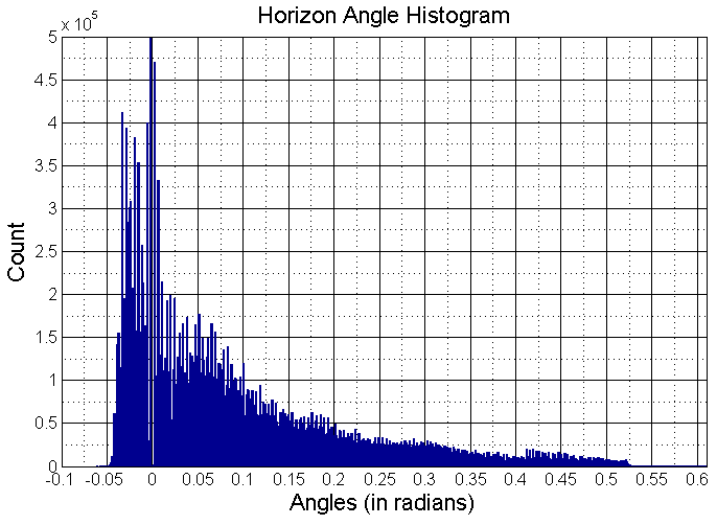


(a) Gaussian Kernel (b) Fourier Signal with Kernel Applied (c) Signal after Synthesis

A Gaussian kernel applied to our signal from Figure 4.20(a).

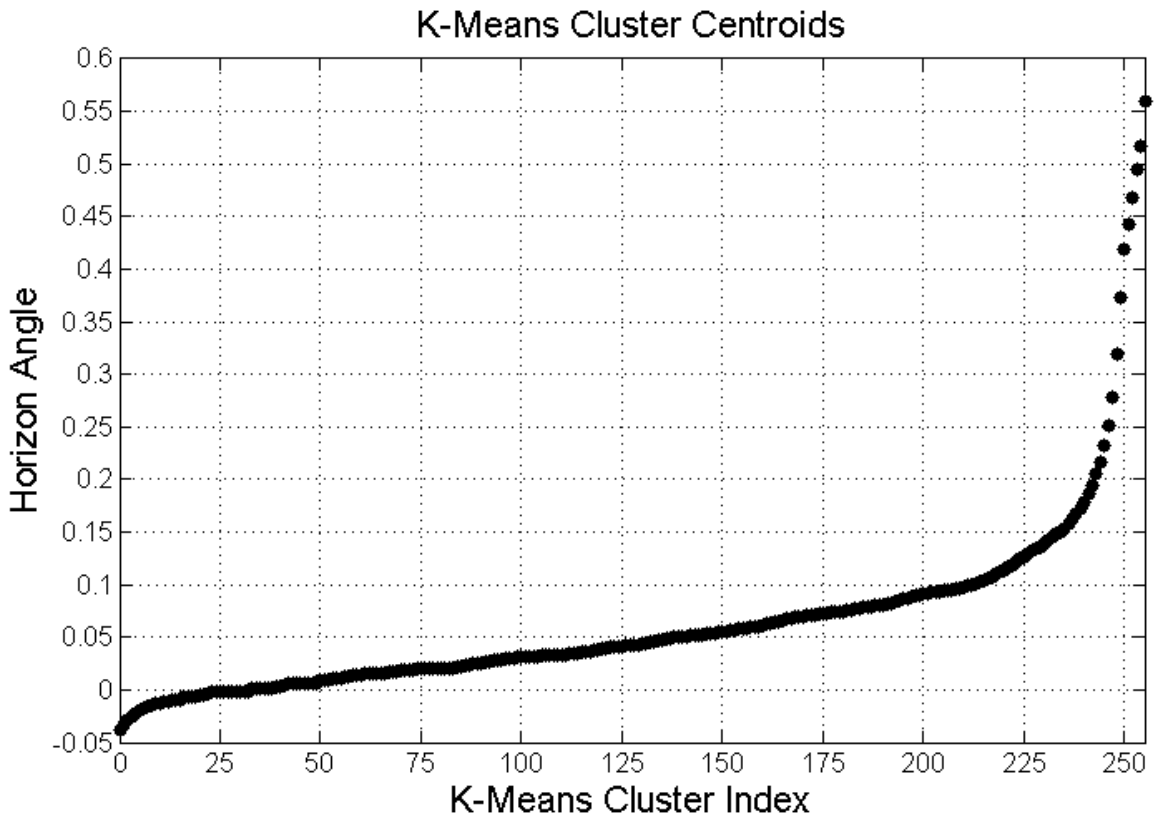
Figure 4.24: Gaussian Kernel Applied

the horizon level. We therefore only need to find a way to encode all horizon angles as one of 256 unique values. To find these values, we study a histogram of one of the moon's patches (see Figure 4.25). We can see that a simple discrete linear mapping from -PI/2 to +PI/2 is insufficient. The histogram clearly shows favoritism to certain values. To exploit this favoritism, we use the K-Means algorithm over all the horizon angles of the patch with K set to 256. This algorithm produces 256 cluster centroids that were iteratively found from an initial guess and should represent the distribution of the data. Areas of high frequency in the histogram would be allotted the most clusters whereas low frequency areas would receive far fewer. Figure 4.26 shows the 256 centroids. Evident in the figure is the large amount of clusters around 0 radians, matching the distribution found in the histogram. We now merely compare each horizon angle to each of the cluster centroids to find which centroid is closest, storing the index of that centroid (from 0 to 255) in our input TIFF . This reduces the TIFF 's channels from 32 bit floats to 8 bit integers, allowing for smaller file sizes.



This shows the histogram of horizon angles over a sample patch. We see that the distribution of this angles is not linear and there is a large clustering near an angle of zero. We can exploit this distribution to better decide what are representative 256 values will be for compressing our data into 8 bits.

Figure 4.25: Histogram of Horizon Angles



After performing the K-Means clustering algorithm over the horizon angles in a sample patch, we can see a lot of the clusters around a value of zero. Each black circle in the figure represents a single cluster centroid and the y-axis value is the horizon angle it represents.

Figure 4.26: K-Means Clustering of Horizon Angles

4.5 Usage

After we have our SCM TIFF images, we need to probe them to generate shadows. There is no extra work done on the CPU . The scene is rendered as normal with no setup. Instead all work is done in shaders on the GPU .

4.5.1 Shaders

We have two modes implemented for comparison, vertex and fragment. The difference lies in where the main calculations take place. The general procedure is to get both the position of the light and the planet normal into world space and use these values to calculate the light's azimuth and elevation angles. Since we haven't encoded every horizon angle into the SCM TIFF images, we need to interpolate to get our horizon angle. This means finding the lower and upper indices into our shadow SCMs , retrieving the value from them, then interpolating to get an approximation of our actual horizon angle. After figuring out the lower and upper SCM indices, we see if our values are actually in frequency domain and need to by synthesized. If so we perform the synthesis, but only on those indices, to retrieve the values in spatial domain. If they are not in frequency domain, we simply sample the SCMs and retrieve the values as is. We then perform the interpolation and set the results as our horizon angle. To account for possible index encoding, we

have a function that applies a regression model based on user preferred regression mode. This mode is set to 'none' if index encoding was not used. Based upon the drawing mode we are in, these calculations are either performed in the vertex or fragment shader. Since the calculations are the same, we only show the fragment shader, Algorithm 4.5.3, as the vertex shader would look very similar. There are several functions that both the fragment and the vertex shader have in common. These are shown in Algorithms 4.5.1 and 4.5.2.

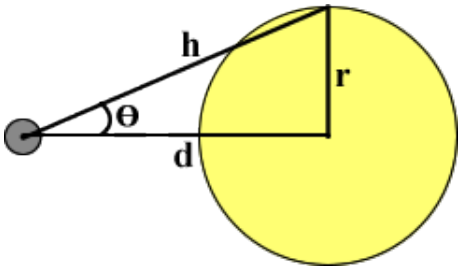
4.5.2 Soft Shadows

For soft shadow generation, we originally used a heuristic that compares the horizon angle to the light elevation. The light's elevation is measured as the angle from the point on the moon to the center of the sun. If the difference between these values are within a threshold, we apply simple hard shadowing by setting the penumbra value to either completely lit or completely shadowed. If the difference is beyond the threshold, we use that difference to create a gradient representing a penumbra region. The last part of the fragment shader renders the shadows using the penumbra value. We have since changed this to be more mathematically accurate. The original heuristic used a linear scale to adjust the penumbra value based on the difference of the light elevation and horizon angle. This implies a cube-shaped light source. Instead this was changed to use the sin function to better mimic the spherical shape of the sun. Since the elevation angle is measured from the center of the sun, we needed to find out how large is the angle that corresponds to the sun's radius. This allows us to figure out at what angle difference would the sun become either fully visible or fully occluded. By taking the sun's radius and the distance between the sun and the moon, we were able to figure out this value.

We start out by using some constant values gathered online from Universe Today [39]:

$$\begin{aligned} r &= 695,842km \\ d_f &= 152,503,397km \\ d_m &= 149,597,888km \\ d_c &= 146,692,378km \end{aligned} \tag{4.3}$$

Where r is the sun's radius, d_f is the farthest distance the moon can be from the sun's center (given that it rotates around the earth which in turn rotates around the sun) while d_m is the average distance and d_c is the closest distance. Given our radius, r, and our distances, we now need to figure out the hypotenuse of the triangle they create (refer to Figure 4.27). We can simply use Pythagorean Theorem with our constants to figure this out as shown in Equation 4.4.



The gray circle represents the moon whereas the yellow circle represents the sun. This simple diagram shows the mathematical idea with the soft shadow part of the algorithm.

Figure 4.27: Geometry behind Soft Shadow Approach

Algorithm 4.5.1 These uniforms and methods are common to both the vertex and fragment shaders

```
uniform uniStride                          ▷ amount of stride. Prefix "uni" used for uniforms
uniform uniNeedFFTSynth                    ▷ true if FFT synthesis is required
uniform uniMaxScmTIFFs                     ▷ maximum number of TIFFs to use (only for FFT )
uniform uniRegressionMode                  ▷ regression mode for index encoding

function APPLYREGRESSION(index)
    if regressionMode = REGMODE_NONE then
        result ← index
    else if regressionMode = REGMODE_EXP then
        result ← EXPREGRESSION(index)
    else if regressionMode = REGMODE_LIN then
        result ← LINEARREGRESSION(index)
    else if regressionMode = REGMODE_POLY then
        result ← POLYREGRESSION(index)
    end if
    return result
end function

function SHADOWSAMPLE(scmIndex)
    texel ← vec4(0)
    for scmDepth ← 0 to 10 do
        samp ← TEXTURE2D(scmTiff[scmIndex], TEXLOC(scmInfo[scmIndex], scmDepth))
        if LENGTH(samp) > 0.001 then
            texel ← samp
        end if
    end for
    return texel
end function

function FFTSYNTHSINGLEVAL(index, F)
    c ← 0
    for k ← 0 to 45 do
        re ← F[k*2+0]
        c ← c + (re * COS((2*PI*k*index)/90))
    end for
    c ← c*(1.0/45)
    s ← 0
    for k ← 0 to 45 do
        im ← F[k*2+1]
        s ← s + (im * SIN((2*PI*k*index)/90))
    end for
    s ← s*(1.0/45)
    return c − s
end function
```

Algorithm 4.5.2 These uniforms and methods are common to both the vertex and fragment shaders

```
function FFTSYNTH(lowerIndex, upperIndex)
    for i ← 0 to maxScms do
        F[i*4+0 : i*4+3] ← APPLYREGRESSION(SHADOWSAMPLE(i))
    end for
    for i ← maxScms*4 to 90 do
        F[i] ← 0
    end for
    return vec2(FFTSYNTHSINGLEVAL(lowerIndex, F), FFTSYNTHSINGLEVAL(upperIndex, F))
end function

function PENUMBRAFUNC(x)
    return SIN(x)/2.0
end function

function GETPENUMBRA(horizonAngle, lightAngle)
    θ_m ← 0.004630                ▷ this value is discussed in the soft shadows section
    if lightAngle + θ_m < horizonAngle then
        penumbraVal ← SHADOW
    else if lightAngle − θ_m > horizonAngle then
        penumbraVal ← LIT
    else if lightAngle ≥ horizonAngle then
        penumbraVal ← 0.5 + PENUMBRAFUNC((lightAngle − horizonAngle)/θ_m)
    else
        penumbraVal ← 0.5 − PENUMBRAFUNC((horizonAngle − lightAngle)/θ_m)
    end if
    return penumbraVal
end function
```

$$\begin{aligned} h_f &= \sqrt{(d_f^2 + r^2)} = 153,200,819km \\ h_m &= \sqrt{(d_m^2 + r^2)} = 150,295,341km \\ h_c &= \sqrt{(d_c^2 + r^2)} = 147,389,862km \end{aligned} \tag{4.4}$$

Where h_f is the hypotenuse using the farthest distance from the sun, h_m uses the mean distance and h_c uses the closest. Now that we have the distance from the moon to the top of the sun, we must convert this into an angle. Using the Law of Sines to solve for θ (in radians) we have Equation 4.5.

$$\begin{aligned} \theta_f &= \arcsin(r/h_f) = 0.004542 \\ \theta_m &= \arcsin(r/h_m) = 0.004630 \\ \theta_c &= \arcsin(r/h_c) = 0.004721 \end{aligned} \tag{4.5}$$

Where θ_f uses h_f, θ_m uses h_m and θ_c uses h_c. Since these values are relatively close to one another, they won't make a significant difference visually and therefore we can just choose one of them; we use θ_m since it is the mean value. To save on the total number of texture probes, we assume that the value of the horizon angle (found by referencing the azimuth angle towards the sun's center) remains constant for a single point on the moon. In reality the horizon angle can change slightly as we look left or right, however this change

Algorithm 4.5.3 Fragment shader for HES shadow algorithm

```
in inHorizonAngle                                   ▷ horizon angle from the vertex shader
in inLightAngle                                     ▷ light angle from the vertex shader

function MAIN( )
    if drawMode = VERTEX then
        penumbraVal ← GETPENUMBRA(APPLYREGRESSION(inHorizonAngle), inLightAngle)
    else
        oldLightDir ← TRANSPOSE(mat3(glModelMatrix)) ∗ lightDir
        newX ← NORMALIZE(CROSS(vec3(1,0,0), normal))
        newY ← CROSS(newX, normal)
        newZ ← normal
        changeOfBasisMat ← mat3(newX, newY, newZ)
        newLightDir ← oldLightDir ∗ changeOfBasisMat
        azimuth ← DEGREES(ATAN(−newLightDir.x, newLightDir.y))      ▷ light's azimuth angle
        lightAngle ← ASIN(newLightDir.z)                            ▷ light's elevation angle
        lowerIndex ← FLOOR(azimuth/uniStride)
        upperIndex ← CEIL(azimuth/uniStride)
        if uniNeedFFTSynth then
            luVal ← FFTSYNTH(lowerIndex, upperIndex)
            lowerVal ← luVal[0]
            upperVal ← luVal[1]
        else
            lowerVals ← SHADOWSAMPLE(lowerIndex/4)
            upperVals ← SHADOWSAMPLE(upperIndex/4)
            lowerVal ← lowerVals[lowerIndex mod 4]
            upperVal ← upperVals[upperIndex mod 4]
        end if
        t ← (int(azimuth) − lowerIndex ∗ uniStride)/uniStride
        horizonAngle ← MIX(lowerVal, upperVal, t)
        penumbraVal ← GETPENUMBRA(APPLYREGRESSION(horizonAngle), lightAngle)
    end if
end function
```

would only introduce or take away relatively small portions of sun light. To consider and therefore alleviate the error introduced by this would cause significant overhead and in most cases would result in little to no change in the final pixel color.

4.5.3 Debugging

There are several viewing modes that were created to help solve issues that we encountered. The following is a list of those modes.

4.5.3.1 SCM TIFF Value PI Range. This mode colors the fragment with the actual value found in the red channel of the first SCM shadow TIFF . It assumes the value found in the TIFF will be in a range of -PI/2 to PI/2 which is the range for floating precision horizon angles. This allows us to see which parts of the moon we have information on in our SCM TIFF . If there is no information in the TIFF at that particular

location, the fragment color is white. A gradient value from black to red indicates information is stored for that fragment.

4.5.3.2 SCM TIFF Value Normalized Range. This is similar to the previous viewing mode exception it assumes the values stored in the SCM TIFF are in a normalized range from zero to one. If the TIFF was generated by converting it into a 16 bit integer, the OpenGL texture's internal format would be set to integer and, when read from the sampler on the GPU , the value would be normalized from zero to one (if using a standard sampler and not an integer or unsigned integer sampler). This viewing mode was created for this case.

4.5.3.3 Show Light Elevation. To make sure the elevation formula is correct, this mode sets the fragment's color to the result of the elevation calculation. A black color indicates the lowest elevation while a red color indicates the highest elevation the light source can be at. The light's elevation is compared against the horizon angle to determine whether a fragment is in shadow or not.

4.5.3.4 Show Light Azimuth. This is similar to the light elevation view mode, except for the light's azimuth angle. The light's azimuth determines which two SCM shadow samplers are tapped and which channels in each are used.

4.5.3.5 Show Horizon Angle. This view mode shows the actual horizon angle retrieved from interpolation after tapping into the appropriate SCM TIFFs and possible applying Fourier synthesis [37] [38]. The horizon angle's range is from $-PI/2$ to $+PI/2$ and so this value is remapped using linear interpolation to a normalized range and is displayed in the red channel of each fragment.

4.5.3.6 Show Light Direction. In the main app, when the light source tells OpenGL its position, it is careful not to include any view transformations. In OpenGL when you specify a light's position, that position is automatically projected into eye space by applying the current modelview matrix. By not specifying any view transformations, OpenGL will project the light's position into world space instead. In the shader the light's direction is then retrieved and back projected into the moon's object space. This viewing mode colors all fragments using the x, y, and z orientation of the light source in the moon's object space. This is used to check if our projection calculations are accurate.

4.5.3.7 Show Normals. The surface normal is used to color the fragments in this mode. One of the features of the application is to rotate the moon to the exact location of a vertex we are examining. This mode was used to find a bug where the vertex's information would be drawn on an area of the moon different from where the vertex's normal was at. By drawing the color of the fragment's normal, we were able to spot this and make the proper fixes.

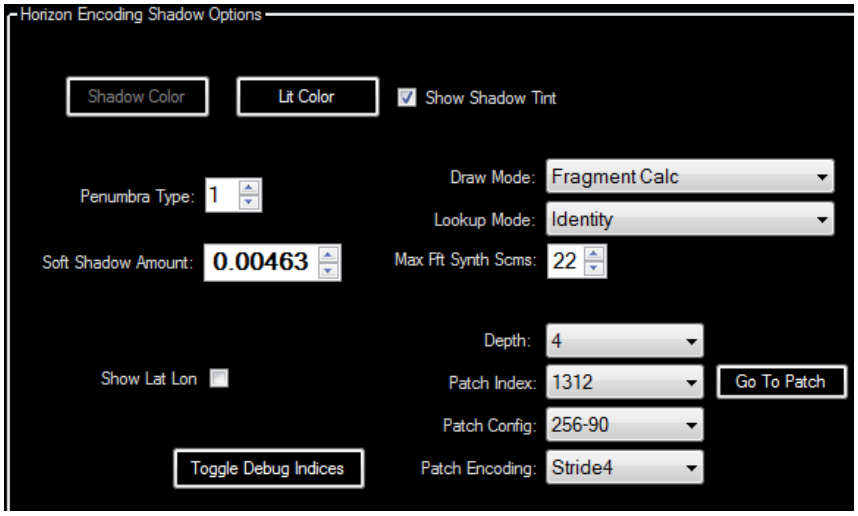
4.5.3.8 Height Map. This mode gives us a simple height map for the moon. This helps us to examine good patches to run the algorithm on, as we can better inspect where there are high peaks that may cast shadows into surrounding lower elevation areas.

4.5.4 Comparing

There are several ways we can compare using the HES algorithm. These ways were implemented into the GUI to test various aspects of the HES algorithm.

4.5.4.1 View Modes. The view mode used can be changed during runtime. It can either be changed to the vertex or fragment computation mode where we can examine which mode is more accurate and visually see the difference running calculations in the vertex shader compares to running them in the fragment shader, or we can change it to one of the debugging modes to better inspect our data set and solve any issues we are currently having.

4.5.4.2 Parameters. As discussed earlier, during certain stages of the preprocessing pipeline, there are parameters that can be set. We have ran horizon calculations and created SCM TIFFs for multiple parameter settings so that we may test these parameter values during runtime. Parameters such as type of storage encoding, horizon fbo height and horizon fovx can be changed to preset values and the correct set of SCM TIFFs is retrieved and applied. For the Fourier compression storage encoding types, we can change the maximum number of SCM TIFFs used when recreating the signal using Fourier synthesis. This is so that we can see the effects of reducing the number of coefficients on the shadow quality and overall accuracy. We can then compare the visual results each parameter has on the shadow output both separately and in combination with other parameter values. Figure 4.28 shows these options in the GUI .

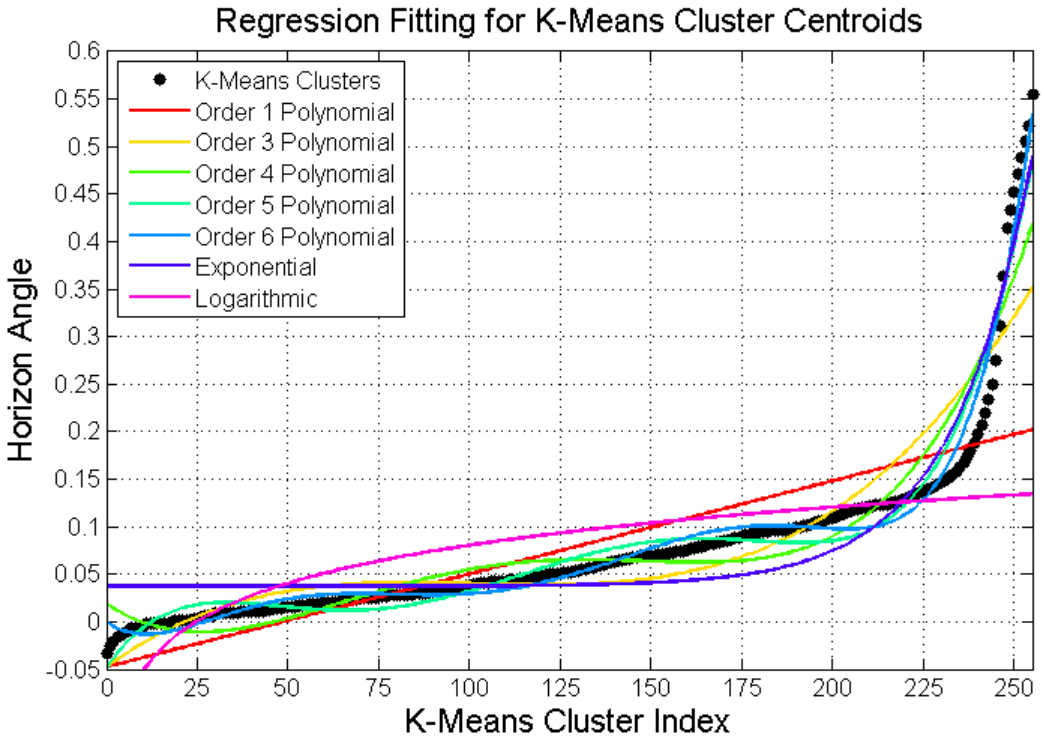


GUI for changing different parameters in the HES shadow algorithm.

Figure 4.28: HES Light Options

4.5.5 Compressing the Data

If we have compressed our data into cluster indices in the storage phase, then we have to have a way to turn the indices back into horizon angles. There are several ways of accomplishing this. One way is to store the lookup table matching cluster index to represented angle in a texture. We can then send this texture to the GPU to sample from. The texel index would denote the cluster index and the texel color would denote the angle value. While this is not a poor approach, this would increase the number of texture lookups performed in the shader. An alternative to this would be to estimate a function that fits the curve created by the K-Means cluster centroids. Given this function in the shader, we simply plug in the cluster index as the independent variable and allow the result to be our angle. Figure 4.29 shows the results of using polynomial, exponential, and logarithmic regression to fit our K-Means curve. Although this way produces an error over the texture lookup technique, it is faster for the GPU and a carefully chosen function can reduce this error. It is important to note that both of these approaches allow for interpolation between cluster centroids.



Polynomial, exponential and logarithmic regression used to fit the curve created by the K-Means cluster centroids. The y-axis shows the centroid location, or the angle it represents, and the x-axis is the corresponding cluster index. The polynomial functions appear to fit the curve the best.

Figure 4.29: Regression for K-Means Clusters

4.5.6 Selecting an SCM Set

The GUI is set to detect which SCM sets have been generated. The first option selects the depth from the depths that have at least one patch. The next option selects the patches that exist for that depth. Selecting -1 means using the depth-wide set of SCM TIFFs , that is the SCM set that was built from all patches in that depth. The next option selects the configuration for fbo height and fovx. The last option selects the encoding type. There is also a button for going to the currently selected patch. Figure 4.30 shows these options.

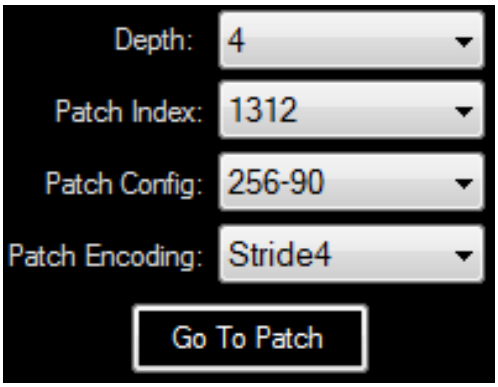
4.5.7 GUI Integration

The HES algorithm is integrated into the application just like any other shadow algorithm. This means it can be selected and used just like other algorithms and therefore we can compare it to our other approaches. This goes well with the split screen feature.

4.6 Summary

An overview of the entire process is described below in Algorithm 4.6.1. In short the entire process can be summed up in the follow steps:

- Capture all vertices at the different depths



Options for choosing which SCM shadow set to use. Here we can choose the depth and patch as well as the configuration and encoding. Choosing -1 for the patch index makes the application use the depth-wide SCM set.

Figure 4.30: HES Usage GUI

- Consolidate them into a single file for each depth
- Go to each vertex in each file, calculate and save the horizon angles
- Optionally, compress these horizon angles using Fourier analysis
- Write the compressed horizon lines into a TIFF image using the scmtiff utility
- Read the TIFF image, reverse the compression (if need be), retrieve the correct horizon angle and use that to calculate whether a point is in shadow or not

Algorithm 4.6.1 Entire HES Algorithm

```
function EntireAlgorithm( )
    CreateForEntireMoon( )
    CombineCache( )
    SortCache( )
    RemoveDuplicateCache( )
    HorizAngleGpuImg( )
    MakeInputImages( )
    RunScmTifBatchScript( )
    RenderWithShadowScmTifs( )
end function
```

4.6.1 Automating the Process

The pre-processing tools were originally setup to process a single set of cache files which covered a rendered portion(s) of the moon at all rendered depths. This was changed to allow a specific patch to be sampled, thereby creating more control over capturing the region that was to be pre-processed. This approach used the mipmapping feature of the scmtiff tool to create data for all depths above that of the processed patch. However, if one wanted to sample a patch at a higher resolution, one would have to manually create the necessary data for each of the four child patches separately. If one wanted to sample several distinct patches,

again the patches must be handled separately. There was also no mechanism currently in place to combine these patches into a complete set of SCM TIFFs , even though the scmtiff tool allows for it. For these reasons we added depth-wide controls to the GUI to help automate the processing of several patches. We also modified the generated batch files to utilize the combine feature of scmtiff.

4.6.1.1 Cache. As mentioned previously we added controls to allow a patch to be sampled at a deeper depth than it currently resides. This means large patches can have its child patches sampled to create higher resolution shadows over that patch. Sampling is done at a resolution of 256x256 for each child patch so as to match the resolution of the topology SCM TIFF image. A cache.txt file is generated for each patch and placed in that patch's directory.

4.6.1.2 Horizon Angles. One can specify a certain depth as opposed to a particular patch, and all patches in that depth's directory on disk that have a cache.txt file will have horizon angles calculated for it. Patches are serially processed and the results are written into the patch's directory. Like caching, we need only a single horiz.txt file to hold the horizon angle information, as mipmapping is used in the final phase to generate data for the more shallow depths.

4.6.1.3 Storage. Similar to horizon angles, one can generate input TIFFs for an entire depth and all patches in that depth will have their input images generated. When this happens a new directory called "Build" gets created and a build.bat file is put into this directory. This build file will convert all the input TIFFs from all patches in that depth into SCM TIFFs , combine them and create a single set of shadow SCM TIFFs .

4.6.2 Plans for Sampling the Entire Moon

Since we can pick any patch on the moon and sample it as deep as we like, thereby creating information for all the subpatches under the root patch for that depth, we can apply this approach for the entire moon. If we have 6 machines each working on a patch at level 0 (i.e. patches 0, 1, 2, 3, 4, and 5), with an added sampling depth of N, we can cover the entire moon generating all information for all patches at depth N. With the automation tools, we can then combine all the input TIFFs to create one set of SCM shadow TIFFs to cover the entire depth. Using the mipmap feature of the scmtiff command line tool to create bilinearly sampled pages from the given data, we can have shadow data for levels 0 through N-1 as well. This means the entire moon will have shadow data for depths up to N. We could also have 24 machines sampling the patches at level 1 up to a depth of N.

4.6.3 HES Problems Encountered

During the process of building the idea and working on the implementation of our main algorithm, there were several issues found along the way. The following discusses a few of those issues encountered.

4.6.3.1 Batch File. A Windows batch file is created by our implementation and used to convert the input TIFF images into SCM TIFFs . The choice was either to use this or create a makefile, and not being too proficient in makefiles due to lack of using them for a long time, the choice was made to use a batch file. Shell scripting in general is not a personal strong point but it wasn't too hard to get this to work. The main issue, however, wasn't noticed right away and only found out after running some debugging values. When inspecting not only the exact values that were sampled from the shadow SCMs but also the order of values, it was apparent that the order was incorrect. This was due to a foreach style loop in the batch script that grabbed all input TIFF files matching the *.tiff pattern and converting them into SCM TIFFs . The foreach

should process the files in lexicographical order and so the order one sees in Windows Explorer (if sort is set to name) should be the order the files were processed. Unfortunately the order in the batch script seemed to mirror that of older Windows versions which had a problem with file names that mixed letters with numbers. So when Windows Explorer was updated with the new sorting system sometime around Vista, the cmd shell was left alone leading to a disparity in the order one sees in the GUI and the order retrieved in a cmd script. File names were changed to remedy this. File names of the form ”t0”, ”t1”, ”t2”, etc. were changed to ”t00”, ”t01”, ”t02”.

4.6.3.2 Latitude and Longitude. Given a normal on the moon's surface it was imperative that the code figure out the correct latitude and longitude. This was especially important to correctly encode the input TIFF images. Equations for these online make several assumptions including positive z-axis direction and default model orientation. None of the equations found were working. The tactic that did work was to think through the math with pencil and paper and come up with equations that seemed to work intuitively. These equations ended up being close to what was found online, but with key differences that fitted our particular situation. This helped out greatly with getting the view transformations correct for calculating the horizon angles.

4.6.3.3 Horizon Calculation View Transform. This function was particularly troublesome to get right. The problem to be solved was, given a vertex's position and normal and an azimuth light angle, come up with the proper view transformations that place the camera at the vertex's position, with the up vector as the vertex's normal, and facing the azimuth direction. Several times the equations were worked out on paper and while some of them seemed to work for some vertices, none of them worked for all vertices. It wasn't until the latitude and longitude issue discussed previously was fixed and a feature was added to rotate the moon to any latitude/longitude point, that a realization happened. The moon rotation feature was added to automatically show any horizon point we were examining with the technique in Figure 4.8. Since we could rotate the moon such that a specific point was right in front of the camera, then we could use this to have the vertex's position right in front of the camera. We use the vertex's normal and convert that into the correct latitude/longitude location. We could then move the camera forward until we were at the ground position of the vertex. Finally, we rotate about the x-axis so as to look forward into the horizon. From here we simply rotate using the azimuth angle. Algorithm 4.3.3 has the pseudocode version of this.

4.6.3.4 SCM Depth. One of the reasons the initial production of shadows was not realized for quite some time with the HES algorithm was due to SCM depth. Initially information about the entire moon was trying to be captured and processed to be used. The idea was to capture all vertices, or at least enough, at depth zero and then use this to produce a rough approximation of shadows across the entire surface of the moon. The problem with this is that with the small size of each image stored within an SCM TIFF , even an approximation is not possible with the way we are using the information. To actually test the algorithm and ensure it is working, instead of going for breadth of information over the entire moon, we needed to look into vertices of a deeper depth in a small area. It was therefore important to pick a small area of approximately 3 to 4 degrees range in both latitude and longitude. This would allow the resolution of the resulting SCM TIFF to adequately cover the target area and produce fragment colorations that actually looked like shadows. The patch we decided on is shown in Figure 4.2. Now we needed to figure out the exact latitude and longitude range of this patch. A visualization was added to display latitude and longitude lines so as to create a rough idea of where the patch lies. The empty white border around the input images helped to tweak these ranges to a more exact estimate. We basically kept decreasing the range until only a very thin border remained in the input images. The small amount of border was kept to ensure all information was indeed included in the TIFF and nothing was being cut off at the edges. Shadows still weren't appearing

until the realization that the scmtiff tool needed the depth that the vertices were captured at, in the form of a command line argument. The default being used at the time was a depth of zero. After changing this to match the capture depth, as well as altering the shader to sample at this depth, we saw our first shadow. Section 6.2.2.4 in the Appendix shows the progression as we go from trying to cover the entire moon, to just covering a small patch and getting the depth issue corrected.

CHAPTER 5. SHADER WORKFLOW AND GUI

5.1 Shaders

Shader organization is not a trivial task. Graphics drivers do not include certain functionality such as a #include directive or any other means of combining code fragments. Fortunately OpenGL has built-in support for specifying multiple strings when compiling shaders but this only gets us so far. Therefore it was important for us to develop our own system of shader organization to prevent rewriting code. This was especially important before the SCM framework was integrated, as before then, each shadow algorithm had its own set of shaders that it used. If a shadow algorithm wanted proper lighting and texturing, it had to include the code for these features. Since there are several algorithms implemented, an approach had to be made to reuse code. The follow section describes this approach. It is important to note that when the lunar rendering framework was integrated in, all the separate shader programs had to become one. The SCM framework controls the shader program and it was easier to just allow other scenes to render using that same program, where certain uniforms were set to distinguish between a moon rendering and a normal scene. At this point, the module approach discussed below wasn't as effective, however for writing and debugging shaders, it was still nice to have different effects put into different files.

5.1.1 Modules

Shaders are shared among the GLshadow and GLstandard dlls as well as the Client application. We have developed a system where shader "modules" are used to store common shader code. These are simply shader files with no main function and that only contains functions to be used by other shaders. To prevent compile errors from some drivers; function prototypes, uniform, varying, attribute, and constant variables needed by the module are put into a comment at the beginning of the file. Any shaders that want to use a module simply needs the commented part put before the main function. Therefore all that needs to be done is to copy it from the commented part of the module and paste it (without comments) into the top of the shader being used. There is code that grabs the contents of the module and inserts it after the main function in the shader. Using this paradigm we are able to have new shaders have the ability to add per-pixel illumination, spherical terrain displacement mapping, texture mapping, and more by simply appending the module's code and calling the right functions. In fact, the fragment shader modules all have a function that takes the current fragment color as an "inout" parameter, similar to copy in / copy out, and uses this to grab the current fragment color, multiply it by the contribution of the module (like the color from lighting calculations), and passing it back to the shader using the modules. Each of these modules then has a uniform boolean that can be used to toggle the module's contribution. Algorithm 5.1.1 shows an example of a fragment shader utilizing this paradigm. The fragcolor starts out as white and gets modified by adding

Algorithm 5.1.1 Example usage of modules

```
[module function prototypes and variables go here]
function MAIN( )
    fragcolor ← VEC4(1)
    TEXTURE(fragcolor)
    LIGHTING(fragcolor)
    CASCADE(fragcolor)
    gl_FragColor ← fragcolor
end function
```

texture, lighting calculations, and cascaded shadow maps to produce the final fragcolor which is set as the

fragment's color. Each of these functions check a boolean uniform value that tells it whether it should apply the effect or leave the fragment color as is. These can then be toggled in the client code and leads to even more dynamic shaders. As stated beforehand, when the SCM framework was introduced, all the modules were loaded in code into one big shader program along with all shadow algorithms. Shader modules and shadow algorithms are still kept in separate files. Uniforms are used to switch to a different active shadow algorithm. Two large files are generated, when the app runs; one for the vertex and one for the fragment shader containing all shader code. This is done since SharpGL appears to have a bug in its implementation of glGetShaderInfoLog, where an error will not properly show up in the returned text. For this reason a separate C++ application was written where one simply drags and drops the full shader code on top of the executable file and native C++ code compiles and attempts the run the shader, reporting any and all errors in either the vertex, fragment, or geometry shader if one is present.

5.1.2 Transform Feedback

Transform feedback is a built-in mechanism of OpenGL and a way to access a part of the pipeline that feeds back vertex information after the vertex shader has ran. Any output variables (such as those with the *out* storage qualifier) from one's vertex shader can be gathered by OpenGL and sent back to the CPU . These can be sent either in an interleaved buffer or each variable can have its own buffer. This mechanism was very useful for capturing some properties of each vertex of the moon after they had been displaced with the SCM TIFF displacement map. Information such as its displaced position, its depth in the SCM TIFF , and its non-displaced normal vector were captured. Transform feedback is used in several steps in the HES pre-processing and we were constantly having to add/modify the values that were being captured. This is not an easy process as several lines of code must be changed and any mistakes results in transform feedback not working at all. For transform feedback to work, OpenGL must know after the shaders have been compiled but before the program is linked the exact names of all variables needing to be captured. When the rendering is complete, a buffer whose size is the product of the amount of bytes needing to be captured per vertex multiplied by the total number of vertices rendered must be allocated and used to retrieve the information. If the buffer is too small, you lose information; too big and you are wasting memory. Every time a variable that needs to be captured changes name, that new name must be updated in the OpenGL code. If a variable changes data type, there are several places to change to ensure the total buffer size for retrieving the values is correct. To automate this process, thereby making it both easier to modify captured variables and ensure no mistakes are made (i.e. bugs are introduced), some new custom commands were introduced into the GLSL files. These commands required some pre-processing steps to take place. The shader source was fed through a method that stripped out these custom commands and made the necessary modifications to output valid GLSL code, inserting the hooks for feedback to work. These custom commands were used like functions and were *__debugPrint* and *__debugCollect* . The commands receive arguments specifying the type of variable and the identifier needing to be captured (see Figures 5.1 and 5.2). The name and type of each variable is remembered so that the total size in bytes of the feedback can be calculated and sent to OpenGL . After rendering, the names of all captured variables are saved in an associative array, along with their captured value. One can then retrieve the value from transform feedback by using the variable name. The reason for the naming convention was to prevent any conflicts with pre-existing identifiers. GLSL reserves any identifiers that begin with two underscores. A user cannot declare a variable that starts with two underscores as GLSL would report this as an error. This means the identifiers *__debugPrint* and *__debugCollect* can not already exist in a shader. The pre-process phase can then convert these commands into valid GLSL before giving them to the graphic driver's compiler. The two commands have similar purposes. *__debugPrint* simply displays the value of the variable using the printing method specified by the user. For example, one can set the output to go to the stdout console, in which case all *__debugPrint* statements would display their results to stdout upon rendering (in the form of "key = value" pairs). *__debugCollect* simply

collects the variable into an associate array indexed by the variable name. After rendering, a collection of associate arrays is generated (one for each vertex). One can then iterate through this array to retrieve the values captured for each vertex. This makes the process of capturing data from the vertex shader and sending it to the CPU very easy. It is as simple as adding a line of code to the shader, then adding code on the client side to retrieve and handle the value that gets returned. This process was later improved by setting variables to be allowed/disallowed by name. Certain stages in the pre-processing of the moon's data needed certain values from the vertex shader, while other stages needed different values, and still others didn't require any values at all. Sending data from the GPU to the CPU is expensive, and therefore sending more data than what is required can slow down the processing of a stage. It was this reason that we added in the ability to restrict or allow certain variables to be captured. We can then have a single shader that contains all the *__debug** commands and selectively allow the ones that we want. Also, if no variables are allowed, then transform feedback is never turned on and therefore no speed penalties are incurred. Due to the way the app works, we found it hard to get this to work while the app is running and therefore the allowed feedback variables are specified when the app starts up and can only be changed in code prior to firing up the app. This was still easier then commenting out the corresponding *__debug** commands in the shader every time we wanted to change which variables were captured.

Input

```
void main()
{
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

  // display the vertex's x position in object space
  __debugPrint(float, gl_Vertex.x);

  // save the vertex's position in clip space
  vec3 vertex_position = gl_Position;
  __debugCollect(vec3, vertex_position);
}
```

Output

```
out float dpOut_gl_Vertex_x;
out vec3 dcOut_vertex_position;

void main()
{
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

  // display the vertex's x position in object space
  dpOut_gl_Vertex_x = gl_Vertex.x;

  // save the vertex's position in clip space
  vec3 vertex_position = gl_Position;
  dcOut_vertex_position = vertex_position;
}
```

At the top of the figure, *__debugPrint* and *__debugCollect* are used. This is what the shader would look like while editing. At the bottom, the result of running the input shader through the pre-process phase. The second argument to the commands are the variable names. Any special characters are replaced with an _ and a new variable is formed by appending either "dp" for *__debugPrint* or "dc" for *__debugCollect* , followed by the text "Out" specifying it is an output variable, followed by an underscore, followed by the new name of the variable. The output is generated by creating this new variable at global scope with the out qualifier and assigning to it at the point of the original *__debug** command.

Figure 5.1: GLSL Transform Feedback Custom Commands Example

Input

```
void main()
{
  __debugPrint(float, gl_Vertex.x);
}
```

```
type = float
name = gl_Vertex.x
newname = name.replace('.', '_') // gl_Vertex_x
identifier = "dpOut_" + newname // dpOut_gl_Vertex_x
declaration = "out " + type + " " + identifier
assignment = identifier + " = " + name
```

Output

```
out float dpOut_gl_Vertex_x; // declaration

void main()
{
  dpOut_gl_Vertex_x = gl_Vertex.x; // assignment
}
```

This shows the process of converting a single *__debugPrint* command and how that command is parsed to create the output. In the middle we see the parsing of the input command, with the value of certain values shown in comments to the right. At the bottom, the output generated with comments showing which variables were used from the processing phase.

Figure 5.2: GLSL Transform Feedback Custom Commands Process

5.2 GUI - Consoles

5.2.1 Shadow Console

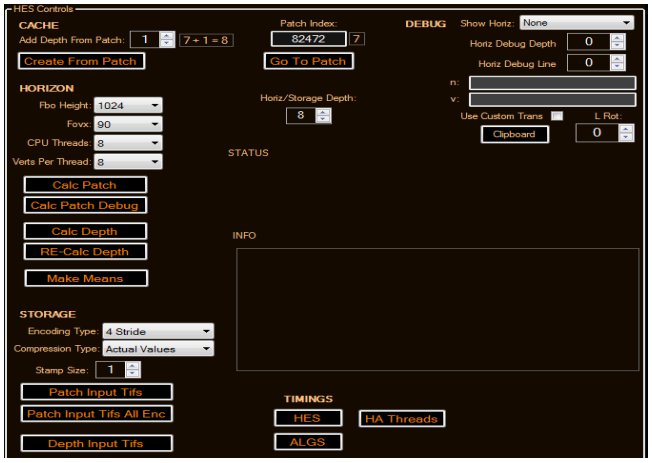
In the shadow console, we can change the algorithm used to calculate shadows. Each algorithm has settings specific to it. To see the full list of settings for each algorithm, go to Figure C.1 in the appendix. We can also split the screen to either 2 or 4 views so as to compare algorithms side-by-side. Each view can have its own shadow algorithm as well as its own settings for that algorithm. This is especially important for testing different options for the same algorithm. Finally, for the shadow map algorithms, we can control the orthographic bounds for generating the shadow map from the point of view of the light source. This console is shown in Figure 5.3.

5.2.2 HES Console

This is where the entire pre-processing takes place for the HES shadow algorithm. Here we can go through each of the steps: cache collection, horizon calculation, input image generation and output image generation. We also have progress text that displays the estimated time left for certain operations as well as the exact vertex/vertices we are processing for the horizon calculation phase. We have settings for various phases such as maximum SCM depth, and controls such as pausing or stopping execution of a particular phase. Figure 5.4 shows this console.

In this console one can choose the shadow algorithm used in the scene as well as any options that go along with that algorithm. One can also split the view to better compare 2 or more algorithms at a time. The boundary of the light's orthographic projection can also be controlled for the shadow map algorithms.

Figure 5.3: Shadow Console



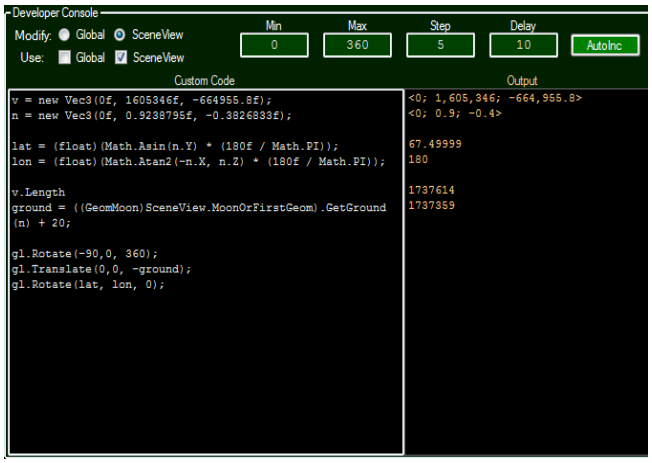
The HES console is where all the controls are for the pre-process stage of the Fourier shadow algorithm. One can capture vertex information, calculate horizon angles, generate input TIFF images, and run the batch file to convert them into SCM TIFFs . This console also has a lot of options to control the output of the pre-process phase.

Figure 5.4: HES Console

5.2.3 Developer Console

The inspiration for this console came about when we were trying to figure out the steps to position the camera when collecting horizon angles. It wasn't trivial to go to an arbitrary vertex location, have the camera look forward at azimuth angle 0 degrees, and rotate about the normal of the point. To help with this, the developer console was created. In this console it is possible to write C# code that gets interpreted on the fly and ran on the OpenGL context. Using this we were able to quickly test several OpenGL commands to get the right series of commands that explained what we were trying to accomplish. On the console if the "SceneView" option is selected, the code is injected into the setup for the scene and any transformation

commands become part of the view matrix. If the "Global" option is selected, code is executed before calculations for that frame are done. All this is accomplished via a parser that was written back in 2009. We had created a custom grammar for C# that parsed and executed several types of expressions and allowed arbitrary strings to be interpreted, during runtime, as C# code. This library was written from scratch as part of a personal project and is very helpful for debugging issues and figuring out correct code for the moon application. Figure 5.5 shows the developer console.



This console is used for finding problems in the application and for figuring out solutions to various issues. One can write custom code and have it evaluated immediately in the scene. There are two modes you can switch from, one that affects the global state of the application and the other that affects the view transformations for the scene. There are separate text fields for each and each can be turned on/off. There is also an output box to display the results of each expression typed in. The output displayed is on the same line as the input, as shown in the figure in orange.

Figure 5.5: Developer Console

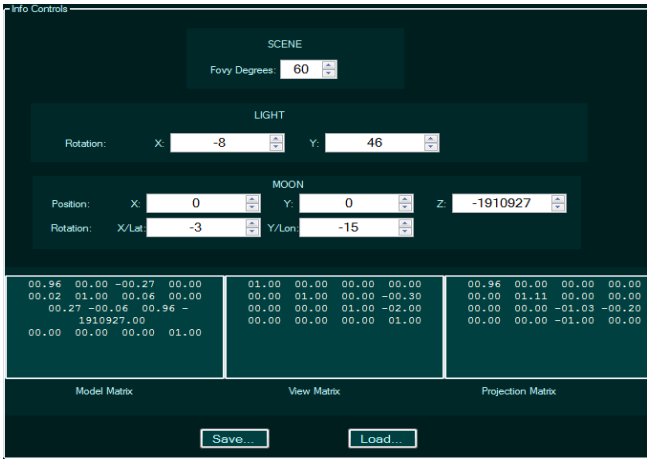
5.2.4 Information Console

The information console contains several controls for various things such as: rotating the moon to a specific latitude and longitude, rotating the light source, controlling zoom level, and saving and loading these settings. The saving and loading feature is useful for going back to the patch on the moon that data was collected for, or for remember all the settings used to take a particular screen shot. The information console can be seen in Figure 5.6.

5.3 Settings

To control various settings in the application, XML files are used. They have grown to incorporate more values over the course of designing the application. An XML file is specified in the code and cannot be changed at runtime, however any values within that XML file can change and the most current values will be used with the app is ran, not requiring a re-compilation. Some of the values one can specify are:

- The background color of the scene
- Default shadow map resolution
- Which variables are allowed to be captured using transform feedback



Here we can control moon and light rotation, zoom level, and save and load scene settings.

Figure 5.6: Information Console

- Which monitor to run the app on
- The default settings for the projection matrix

As well as several other options (a complete list of settings can be found in the appendix in Section 6.2.2.4).

CHAPTER 6. EXPERIMENTS AND CONCLUSION

6.1 Experiments

All code was run on a Windows 7 machine with 8GB of main memory and an nVidia GeForce GT 630M card with 2GB of GPU memory.

6.1.1 Time Comparisons

Tables 6.1, 6.2 and 6.3 show the raw frame timings calculated in milliseconds using the moon scene and rendering the shadows found in Figures 6.2 and 6.3. All timings done using OpenGL queries so as to get the actual time spent on the GPU . In Table 6.1 we see the quickest time is the NSM algorithm. Since all the calculations are done in the vertex shader and values are simply interpolated and used as-is in the fragment shader, the total amount of work is the least among the other algorithms (although as the size of a triangle approaches that of a pixel, this would result in more work being done). For CSM, VSM and NSM with the fragment mask, the times are roughly the same. This is because these algorithms do approximately the same amount of work. ESM and PCF both apply a kernel and therefore they both make multiple taps into the shadow map, thereby increasing their render time. As can be expected, a 5x5 kernel is slower than a 3x3 kernel, however not always by much.

Table 6.1: GPU Timings for Shadow Map Algorithms on the Moon Scene

Frame render times for shadow map algorithms. All algorithms use a 4096x4096 shadow map. (3x3) and (5x5) indicate a kernel size of 3x3 and 5x5, respectively. NSM (frag) indicates the NSM algorithm with the fragment mask feature applied.

Algorithm	Render Time
CSM	39
ESM (3x3)	48
ESM (5x5)	49
NSM	28
NSM (frag)	38
PCF (3x3)	55
PCF (5x5)	61
VSM	37

In Table 6.2 we examine the different approaches to our MPG algorithms as well as the many different parameters for our multi-pass approach. We examine the use of the optional dilate in pass 3, which increases time when used; use of Poisson Disk distribution to sample our kernel, which increases time when not used; number of passes to make, where more passes means more time; and our kernel size, where bigger sizes increase our time. On average, adding the dilate pass increases frame time by about 2ms. Using Poisson distribution appears to cut 1ms per pass on average. An increase from five to ten passes, without Poisson distribution, increases frame time by about 10ms. With Poisson distribution, this increase is lessened to about 6ms. Increasing kernel size from 3x3 to 5x5 has a similar effect as it does in ESM, in that it increases time by about 1ms. Since with every pass we decrease the value indicating the occluder distance and since occluder distances of zero are immune to kernel application, kernel size matters most in the first few passes. If we were to apply a variable sized kernel, then kernel size would merely indicate a maximum and would also not affect render times as much when increased. With all this in mind, high kernel sizes with high

number of passes and without Poisson distribution will still have a noticeable, negative effect on time. This is the reason Poisson distribution was added to the MPG algorithm.

Table 6.2: GPU Timings for MPG Algorithm on the Moon Scene

Value of Dp set to 1.0 and value of Dm set to 1,000. No variable kernel size used in MPG MP.

Approach	Dilation Used	Poisson Dist. Used	Number of Passes	Static Kernel Size	Render Time
PCF	-	-	-	-	54
Single Pass	-	-	-	-	53
Multi-Pass	no	no	10	5x5	61
Multi-Pass	no	no	10	3x3	60
Multi-Pass	no	no	5	5x5	51
Multi-Pass	no	no	5	3x3	50
Multi-Pass	no	yes	10	5x5	53
Multi-Pass	no	yes	10	3x3	50
Multi-Pass	no	yes	5	5x5	46
Multi-Pass	no	yes	5	3x3	45
Multi-Pass	yes	no	10	5x5	63
Multi-Pass	yes	no	10	3x3	62
Multi-Pass	yes	no	5	5x5	53
Multi-Pass	yes	no	5	3x3	52
Multi-Pass	yes	yes	10	5x5	54
Multi-Pass	yes	yes	10	3x3	53
Multi-Pass	yes	yes	5	5x5	48
Multi-Pass	yes	yes	5	3x3	47

Table 6.3 shows frame times for the 3 Stride and 4 Stride HES encoding types. If we compare these times to that of Tables 6.1 and 6.2, we find they are by far the lowest. This is mainly due to the fact that the HES algorithm applies shadows during the normal rendering pass and doesn't require any extra passes. It is faster than the kernel approaches because we calculate ahead of time which SCM TIFFs we need to sample from and only make at most two total taps, retrieving our values and then interpolating between them to get our horizon angle. Clearly the fragment time is slower than the vertex time in the cases where there are more vertices than fragments, however the fragment time is still below even that of the NSM approach without the fragment mask applied. This shows the importance of not having to render a scene twice as well as making the minimum number of taps into a texture.

Table 6.3: GPU Timings for HES Algorithm on the Moon Scene

3 and 4 stride types are tested with calculations done in vertex or fragment shader. Fourier encoding types are tested more throughly in Table 6.4.

Encoding Type	Shader	Render Time
3 Stride	Vertex	22
3 Stride	Fragment	25
4 Stride	Vertex	19
4 Stride	Fragment	26

When one of the Fourier transform encoding types is used, Fourier synthesis occurs on the GPU .

SCM TIFFs are sampled contiguously from the first TIFF containing the first four frequency coefficients up to the last TIFF containing the remaining coefficients. The last TIFF sampled can be adjusted in the GUI , thereby changing to total amount of TIFFs sampled. Zero padding is used if the last TIFF sampled is less than the total number of TIFFs . Since this can affect performance, we tested the different values to gauge its overall effect. Table 6.4 shows the frame times for different values of this parameter. Figure 6.1 shows a plot of the data. Examining the plot we can see a linear trend between the frame time and number of SCM TIFFs we sample. Note that with a reasonable number of TIFFs sampled in the vertex shader, our frame times are not too far from the 3 Stride and 4 Stride encoding types. Sampling more from the textures and performing Fourier synthesis does not affect performance too much. In the fragment shader, however, this is a different story. Frame times there are prohibitively bad and prevent smooth interaction. This is not too much of an issue since performing the calculations in the vertex shader leads to results that are on par with performing the same calculations per fragment, as can be seen later in Section 6.1.2.

Table 6.4: GPU Timings for Different Maximum SCMs used in Fourier Synthesis

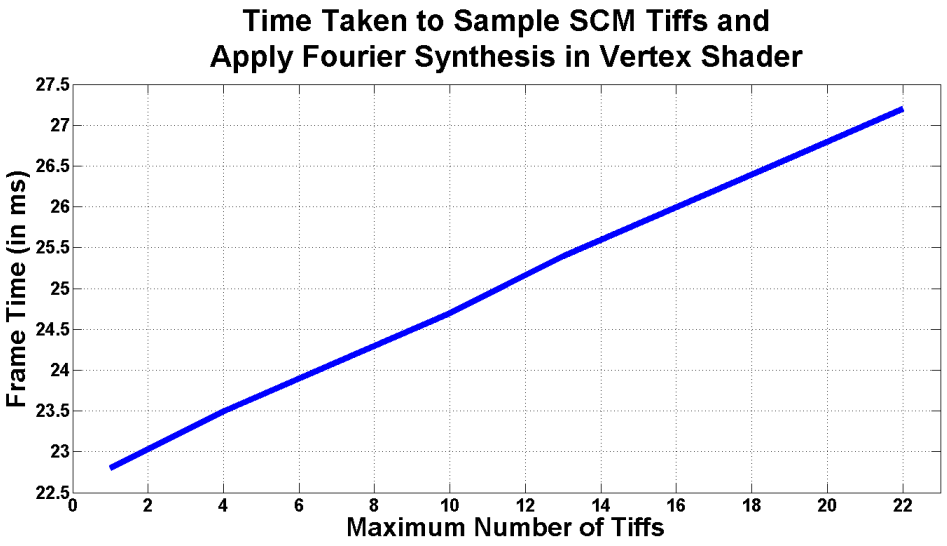
Calculations are done in the vertex shader.

Max SCMs	Vertex Time	Fragment Time
22	27.2	151
19	26.6	143
16	26.0	134
13	25.4	125
10	24.7	117
7	24.1	108
4	23.5	98
1	22.8	89

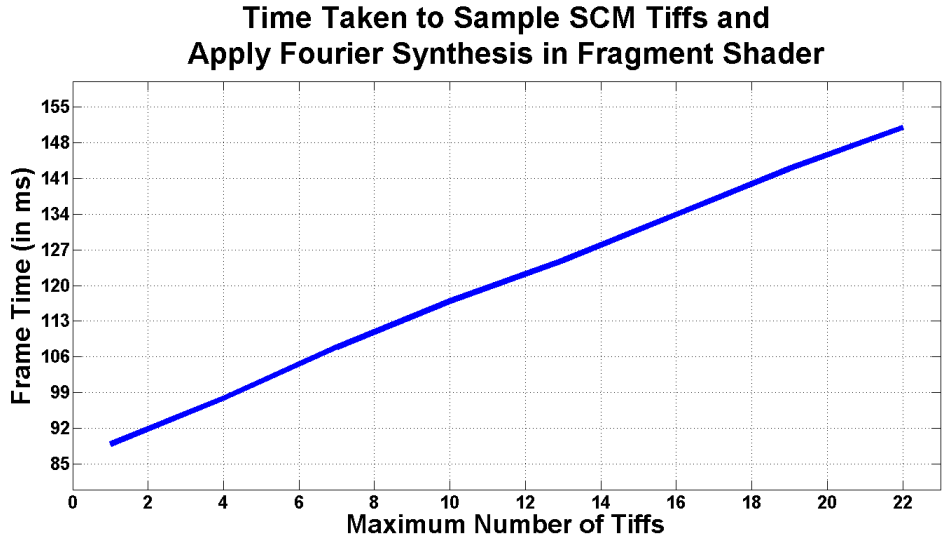
6.1.2 Visual Comparisons

The following figures show all algorithms executed on the patch found in Figure 4.2. For some figures we separate algorithms into two categories, hard and soft shadows. Our MPG algorithm belongs to the soft shadow category. Our HES algorithm is shown in both, activating its soft shadow heuristics in the soft shadow category. The ray traced images were generated by converting the captured vertices over the patch into an .obj file, importing that file into Maya 2015 Student Edition, and recreating the material properties and lighting conditions from the app. The vertices outside the patch area including a very small portion of the patch itself were not captured, and were therefore replaced with a solid gray color. For comparison the ray traced image is placed in both categories.

6.1.2.1 Patch Comparison. Figures 6.2 and 6.3 show all algorithms used on the full patch. Shadow map algorithms used a 4096x4096 shadow map. CSM uses only a single cascade and is therefore just a basic shadow map implementation. Looking at the hard shadow comparisons, Figure 6.2, we see the HES images come close to the shadow map implementations, but are free from aliasing and the false shadowing they cause. The fragment version of HES is only slightly better than the vertex version. Combining these visual results with the timing results discussed earlier, one can make a decision as to whether to use the vertex or the fragment version. Without Fourier synthesis, the added accuracy in shadow calculations may become worth the time hit. With Fourier synthesis, the difference is not nearly substantial enough to warrant calculating in the fragment shader. When compared to the ray traced result, we see all algorithms have correctly detected the casted shadows near the bottom, middle crater, as well as having similar results for the craters to the left. The small crater at the top is casted more outward to the right in the ray traced image.



(a) Vertex Shader



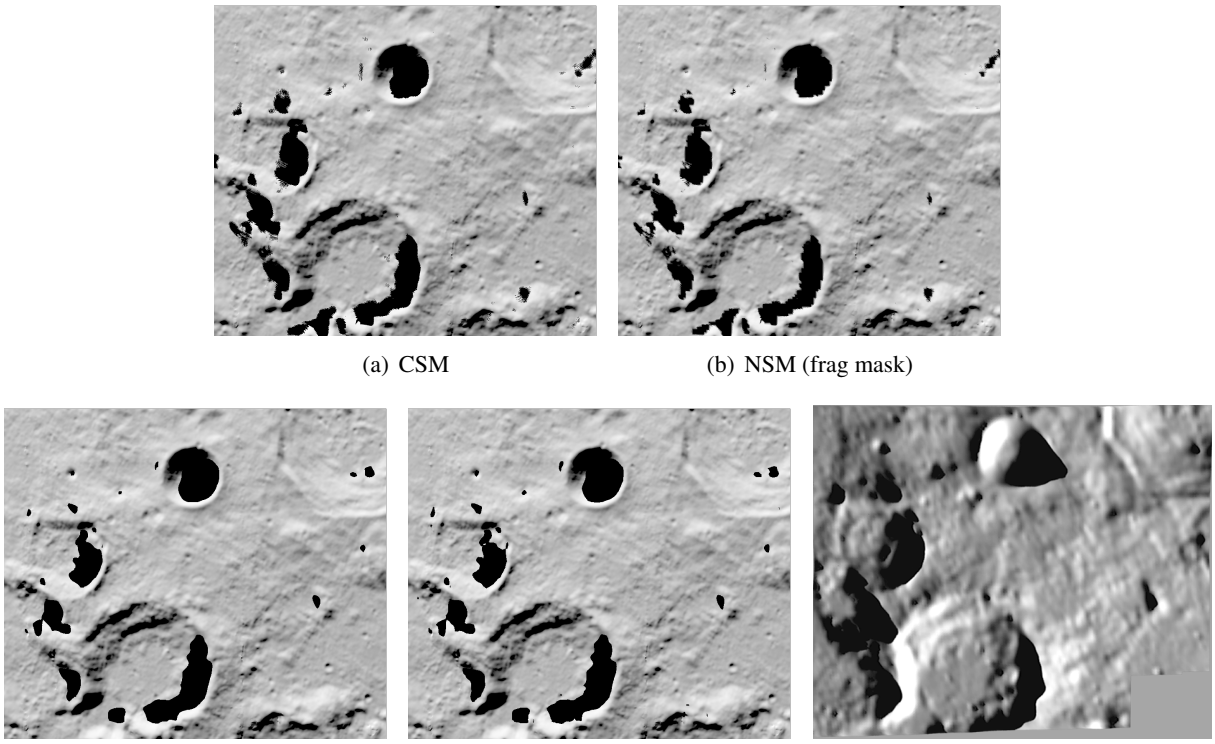
(b) Fragment Shader

Plot showing the linear relationship between frame time and the reconstruction of horizon signal using Fourier synthesis while altering the total number of SCM TIFF images sampled. Time shown for both (a) the vertex shader and (b) the fragment shader. This is a visualization from the data in Table 6.4.

Figure 6.1: Plot of Frame Time v. Maximum SCM TIFFs Sampled and Fourier Synthesis Applied

6.1.2.2 Close Up. Figures 6.4 and 6.5 show all algorithms used on a small portion of the patch, zoomed in for better detailed comparisons.

6.1.2.3 Fourier Synthesis. Figures 6.6 and 6.7 show the effects of using the Fourier encoding types and adjusting the maximum number of TIFFs sampled, both without a kernel and with a kernel, respectively.

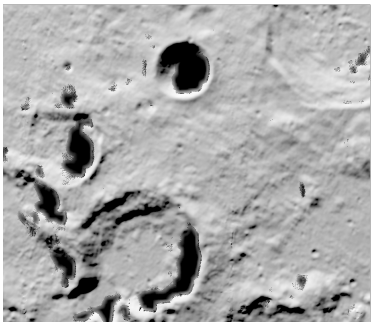


(a) CSM (b) NSM (frag mask)

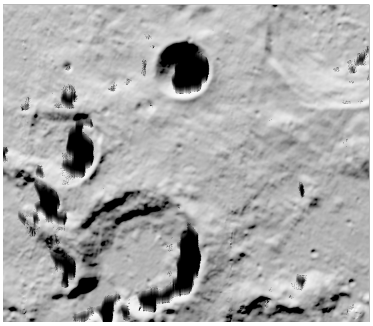
(c) HES (vert) (d) HES (frag) (e) Ray trace

A comparison of implemented hard shadow algorithms. Shadow map algorithms used a 4096x4096 shadow map. For (b) NSM (frag mask), we used the fragment mask feature. HES image uses the four stride encoding type with no Fourier transform (for comparison with Fourier transform, see Figure 6.6 and 6.7. (e) Ray trace , was created in Maya 2015 Student Edition.

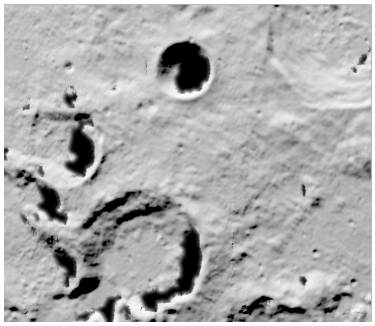
Figure 6.2: Hard Shadow Comparison



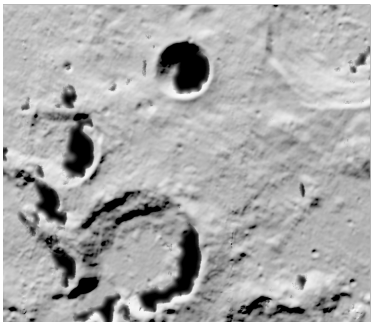
(a) MPG PCF



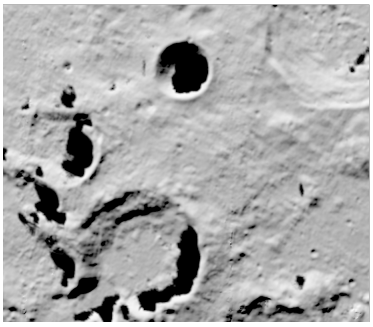
(b) MPG SP



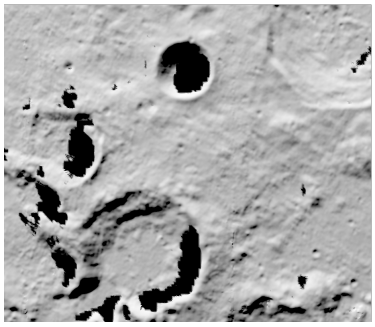
(c) MPG MP



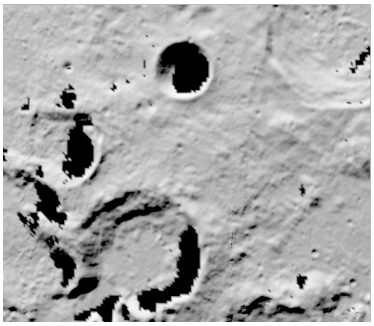
(d) MPG MP (dilate)



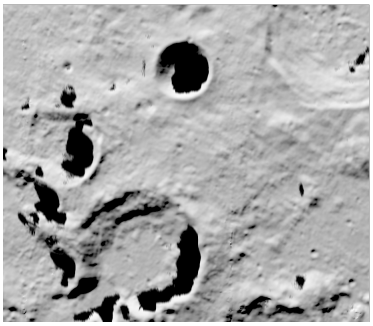
(e) PCF



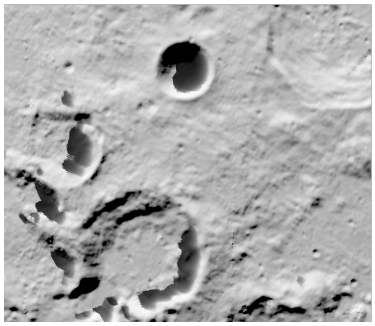
(f) NSM (frag mask)



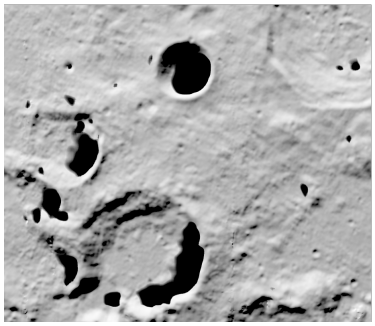
(g) NSM



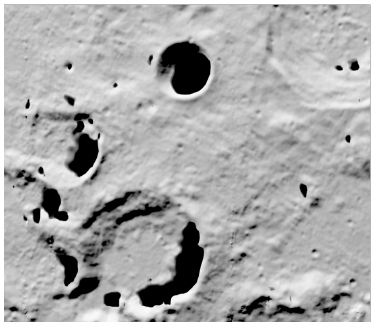
(h) ESM



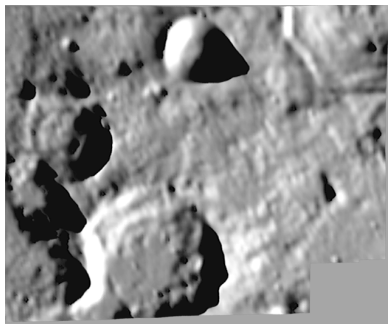
(i) VSM



(j) HES (vert)



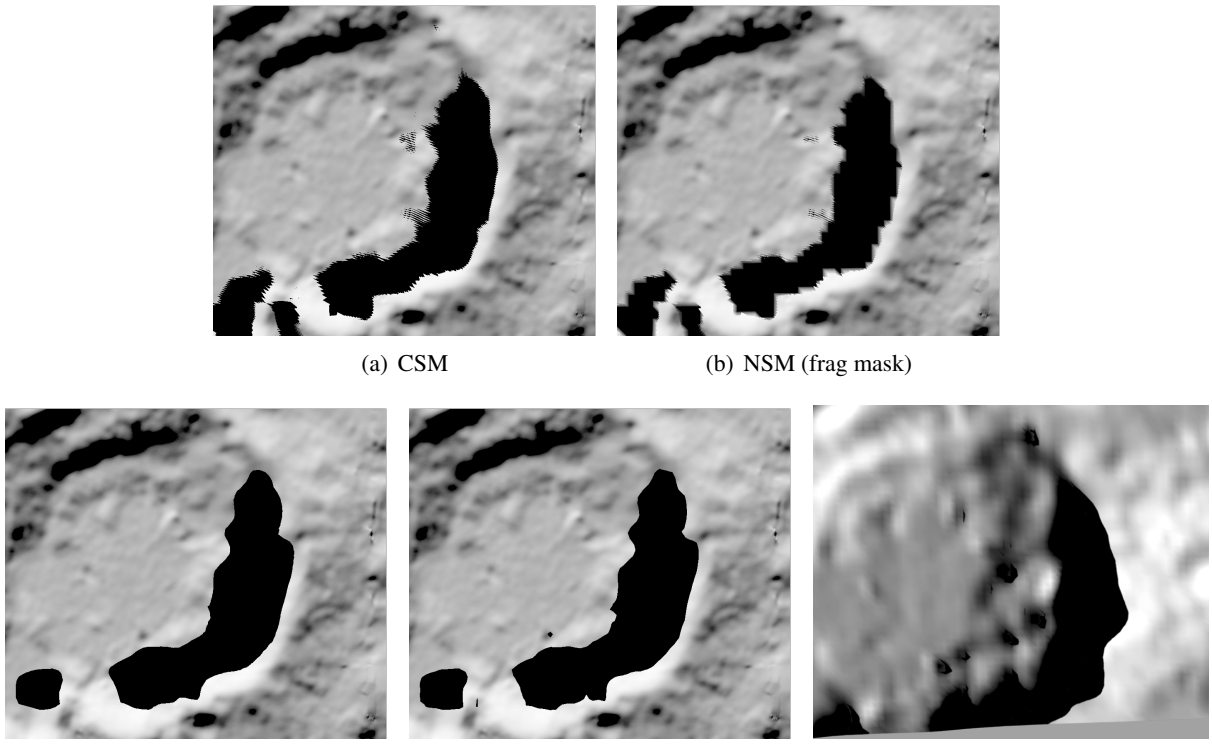
(k) HES (frag)



(l) Ray trace

A comparison of implemented soft shadow algorithms. Shadow map algorithms used a 4096x4096 shadow map. Kernel-based algorithms used a 5x5 kernel. (f) NSM (frag mask), uses the fragment mask feature. HES image uses the four stride encoding type with no Fourier transform (for comparison with Fourier transform, see Figure 6.6 and 6.7. (l) Ray trace , was created in Maya 2015 Student Edition.

Figure 6.3: Soft Shadow Comparison

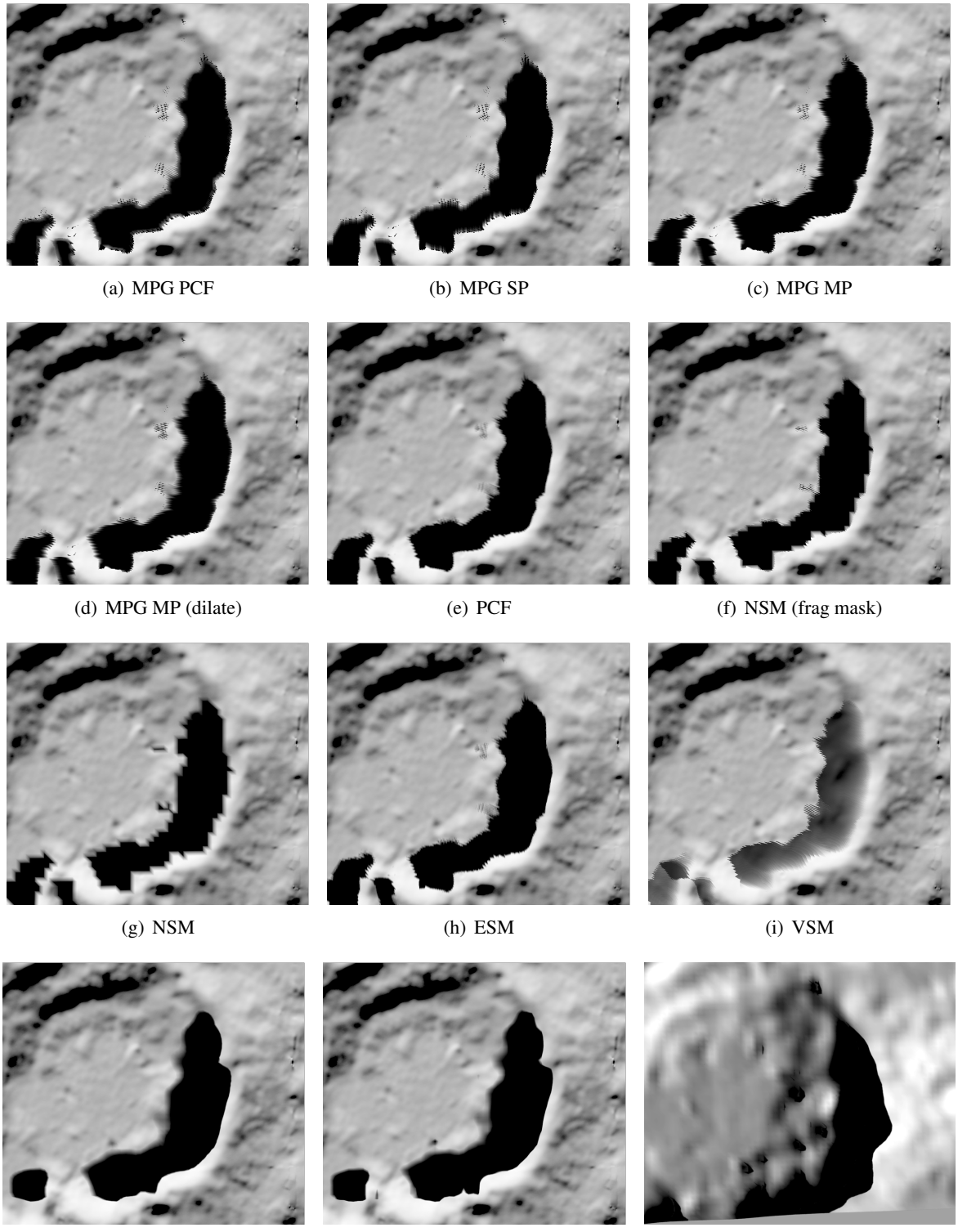


(a) CSM (b) NSM (frag mask)

(c) HES (vert) (d) HES (frag) (e) Ray trace

A comparison of implemented hard shadow algorithms. Shadow map algorithms used a 4096x4096 shadow map. For (b) NSM (frag mask), we used the fragment mask feature. HES image uses the four stride encoding type with no Fourier transform (for comparison with Fourier transform, see Figure 6.6 and 6.7. (e) Ray trace , was created in Maya 2015 Student Edition.

Figure 6.4: Hard Shadow Comparison



(a) MPG PCF (b) MPG SP (c) MPG MP

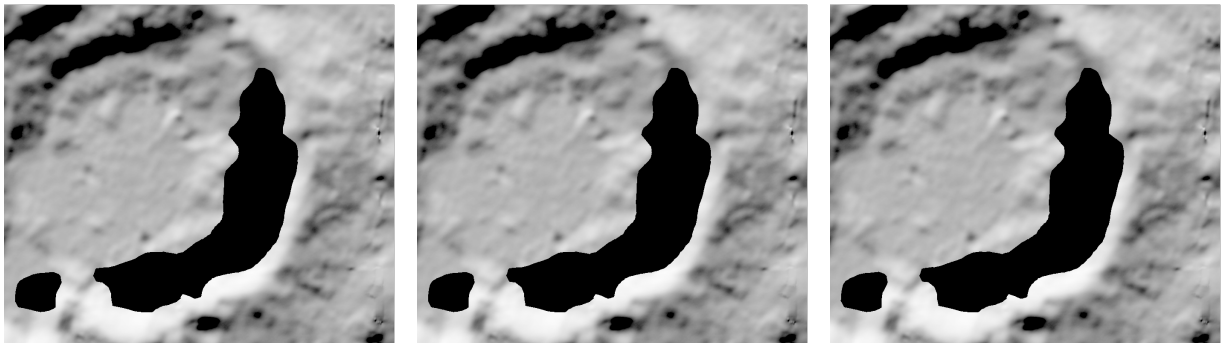
(d) MPG MP (dilate) (e) PCF (f) NSM (frag mask)

(g) NSM (h) ESM (i) VSM

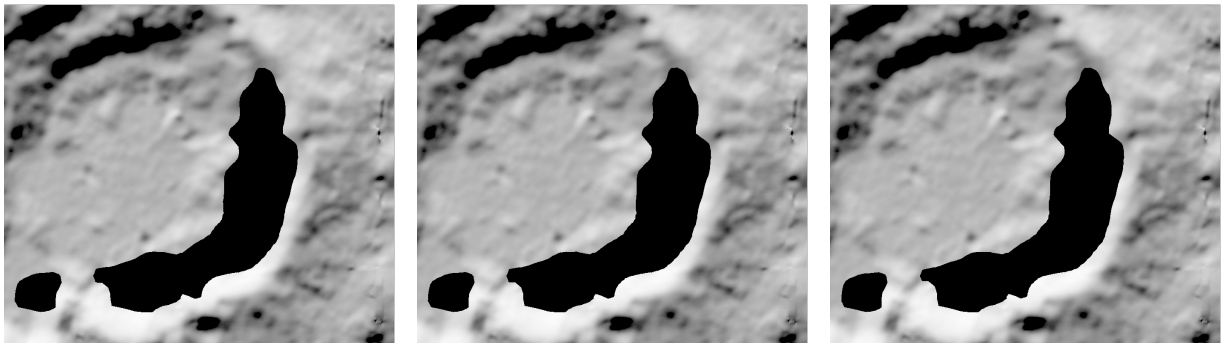
(j) HES (vert) (k) HES (frag) (l) Ray trace

A comparison of implemented soft shadow algorithms. Shadow map algorithms used a 4096x4096 shadow map. Kernel-based algorithms used a 5x5 kernel. (f) NSM (frag mask), uses the fragment mask feature. HES image uses the four stride encoding type with no Fourier transform (for comparison with Fourier transform, see Figure 6.6 and 6.7. (l) Ray trace , was created in Maya 2015 Student Edition.

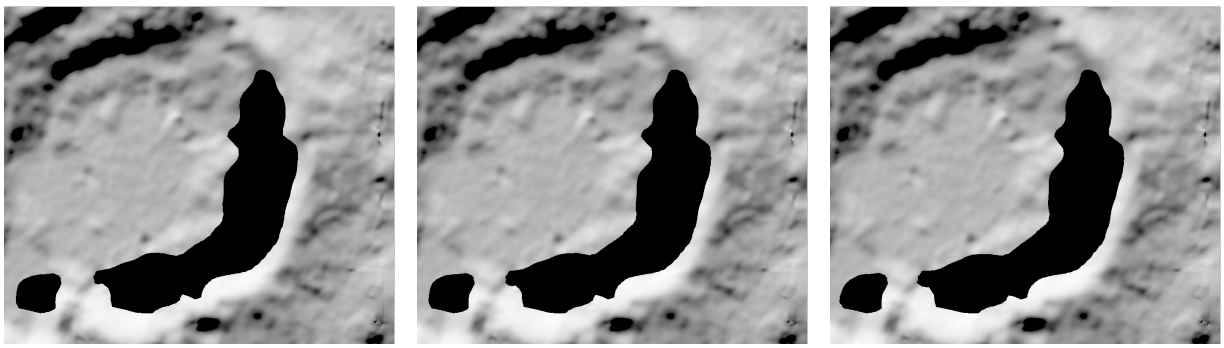
Figure 6.5: Soft Shadow Comparison



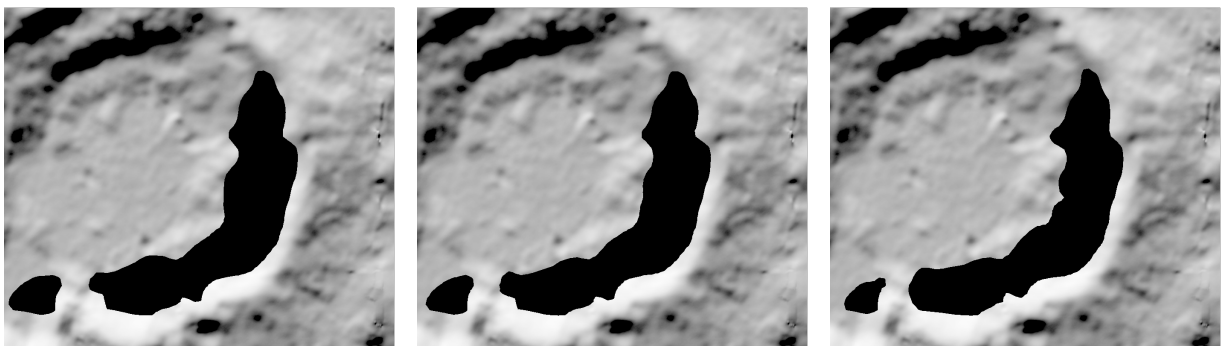
(a) 22 TIFFs (b) 16 TIFFs (c) 10 TIFFs



(d) 9 TIFFs (e) 8 TIFFs (f) 7 TIFFs



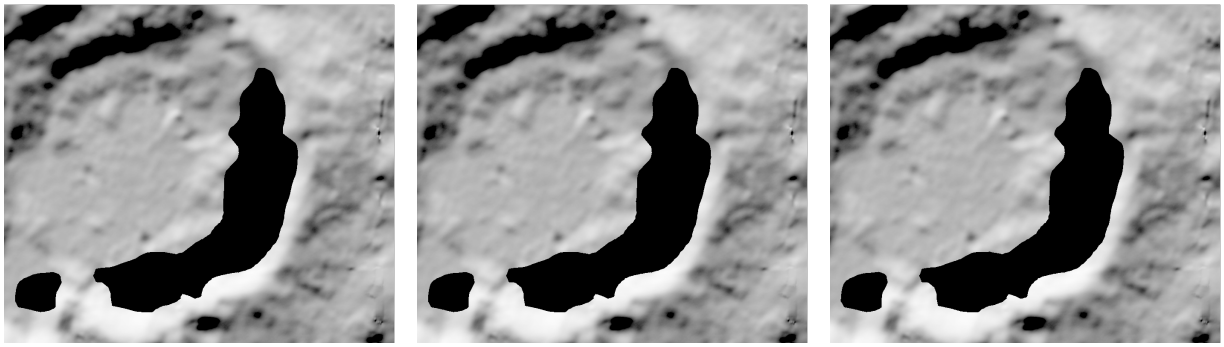
(g) 6 TIFFs (h) 5 TIFFs (i) 4 TIFFs



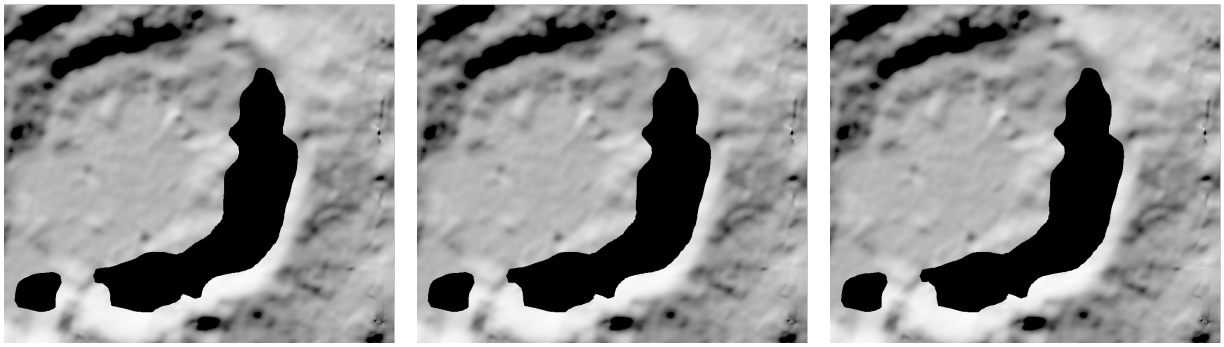
(j) 3 TIFFs (k) 2 TIFFs (l) 1 TIFFs

The effects of changing the total number of SCM TIFFs sampled in the shader before zero padding and performing Fourier synthesis. Four striding was used and no kernel was applied.

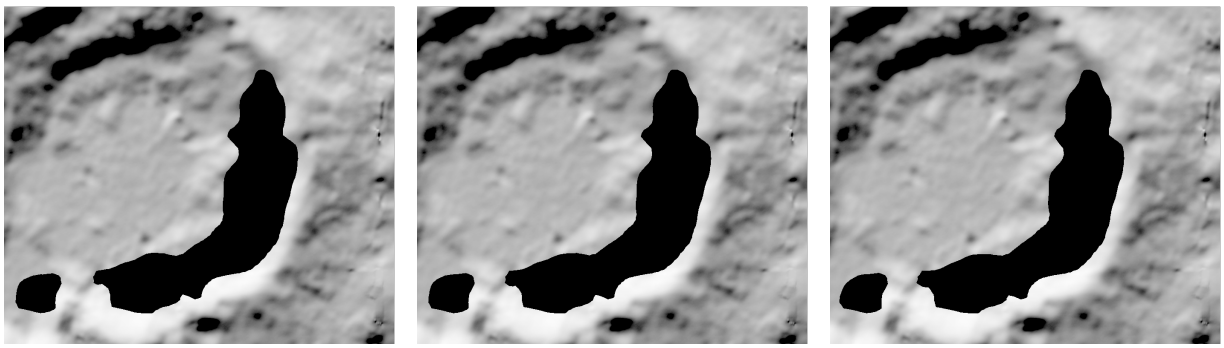
Figure 6.6: Effects of Fourier Synthesis on Different Amounts of TIFF Files (no kernel)



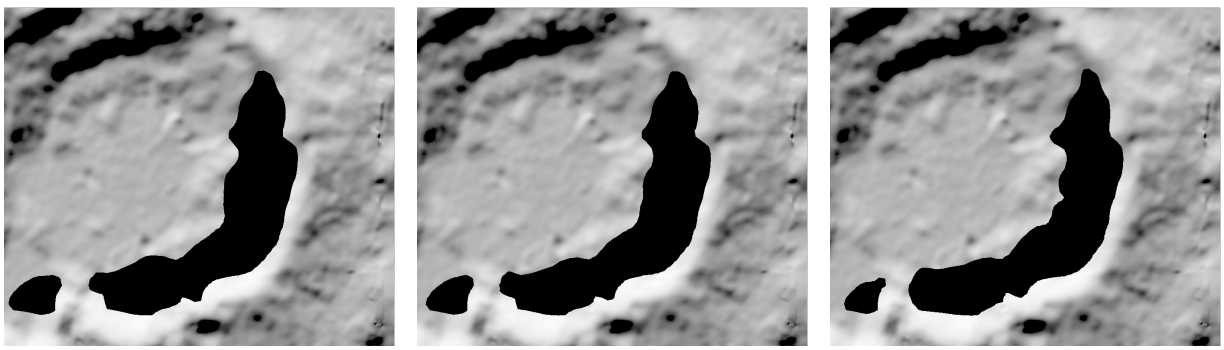
(a) 22 TIFFs (b) 16 TIFFs (c) 10 TIFFs



(d) 9 TIFFs (e) 8 TIFFs (f) 7 TIFFs



(g) 6 TIFFs (h) 5 TIFFs (i) 4 TIFFs

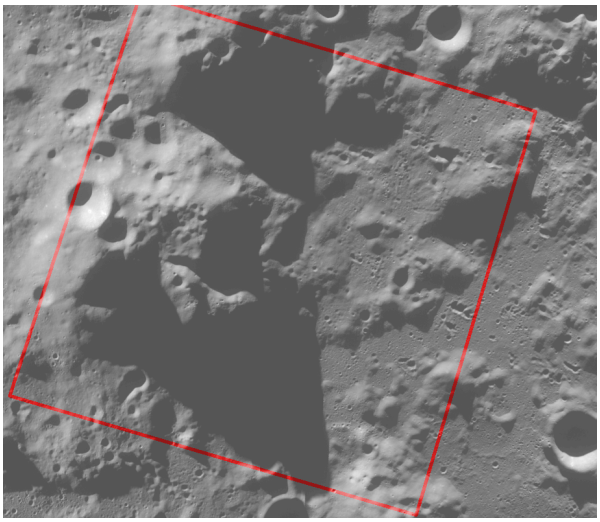


(j) 3 TIFFs (k) 2 TIFFs (l) 1 TIFFs

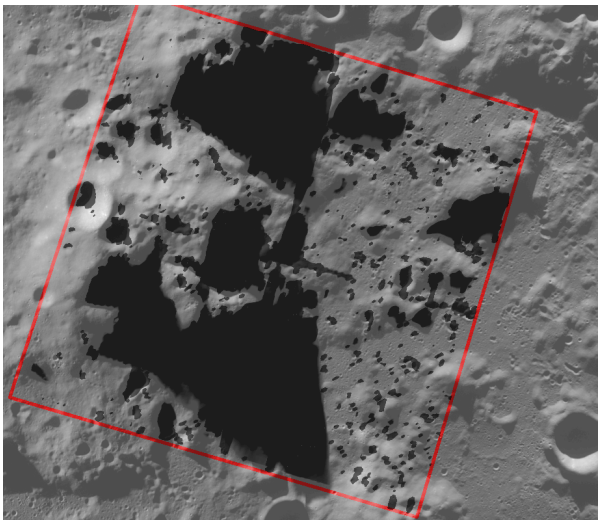
The effects of changing the total number of SCM TIFFs sampled in the shader before zero padding and performing Fourier synthesis. Four striding was used and a kernel was applied.

Figure 6.7: Effects of Fourier Synthesis on Different Amounts of TIFF Files (kernel applied)

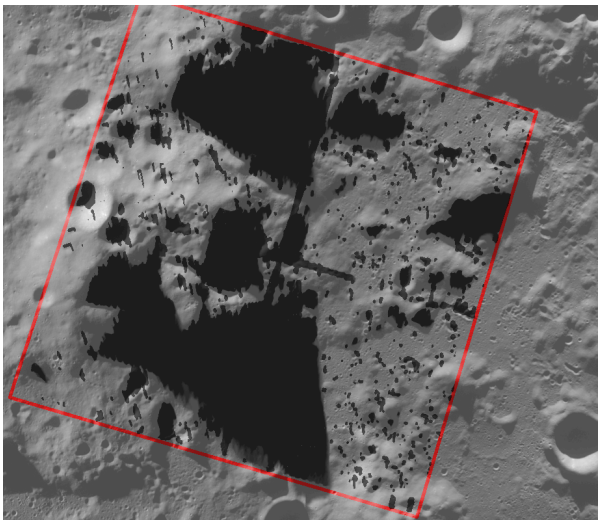
6.1.2.4 Photographic Ground Truth. The SCM TIFF data set that holds the texture information has shadows baked into it indicating the true shadows casted on the moon's surface at the time of gathering the data. This means we have access to actual shadows from the light source of the sun that were present at the time of data collection. This stands as an ultimate ground truth and we can compare our HES algorithm to it. Examining a patch near the south pole, we see in Figure 6.8 a comparison showing ground truth only, HES generating shadows using vertex and fragment shader calculations, and an overlap of HES and the ground truth using vertex and fragment shader calculations.



(a) Ground Truth Only



(b) Overlap Vertex



(c) Overlap Fragment

In (a) we see only the ground truth showing actual shadows on the moon's surface. In (b) and (c) we see the overlap of (a) with the synthetic shadows generated by HES . The red border around the images show the target area and the area we have generated shadow information for.

Figure 6.8: Comparison of HES to Ground Truth

6.2 Further Work

6.2.1 MPG

The following are ways we can improve our MPG algorithm.

6.2.1.1 Kernel Choices. Other low-pass filters can be used in place of a Gaussian kernel. We have not explored as to whether there are benefits of using these other types of kernels. Also, the kernel shape may benefit from taking into account the shape of the light source. While the kernel size is dynamic according to occluder distance for two of the stated approaches, light size can still perhaps influence the initial kernel size or the rate of kernel size growth. The light's size can also have an effect on the multi-pass approach. These may lead to more accurate shadows with respect to different types of area lights. This will also allow us to calculate penumbra width based on Equation 1.1 or Equation 1.2.

6.2.1.2 Multiple Occluders. Although we do not believe this to have a huge impact on accuracy, the case of having more than one occluder has not been extensively tested. We could incorporate and test an average or min occluder depth algorithm.

6.2.2 HES

The following paragraphs describe ways in which we can improve upon the HES algorithm.

6.2.2.1 Reading Pixels in Horizon Angle Calculation. The current way that horizon calculation is performed is to write the position of each fragment into its color, read back the buffer, and perform geometry to calculate the horizon angle given the read back position. This may not be the most efficient solution. Consider that the horizontal resolution of the frame buffer we use to write these values has a great influence on the number of distinct values we can have, then the number of distinct values we can read from a given pixel is limited. In other words the discretization of the data is tied to the height of the frame and within each pixel there is a limited range of world space heights that can be written into it. We have already discussed that the precision needed for our horizon angles can be lowered without sacrificing visual integrity, therefore the range of values that can fall within a pixel can be changed to just a single representative value. We can take the row the first non-black pixel appears in and use the representative value for that row, calculating a horizon angle with that value. This means instead of reading three color channels from the frame buffer, we can read a single binary value of black or white for each pixel. This should increase the speed of horizon angle calculation by reducing traffic coming from the GPU . If we combine this with our 8 bit index approach, we will need the frame buffer to have a height greater than 256 so as to sample at a greater resolution before reducing down into 256 possibilities, akin to how supersampling works. A height of 1024 or greater should suffice.

6.2.2.2 Packing 8 Bit TIFF Channels. Since using the K-Means index encoding reduces the data held in our SCM TIFFs from 32 to 8 bits, we can use those extra 24 bits to store more values. Instead of just reducing the size of each TIFF , we can pack four values in a single channel, causing each 4-channel TIFF to hold 16 values per pixel. Using our 4 Stride encoding, we only need 90 total values and since each TIFF holds 16 of these values, only 6 TIFFs are needed. The only complications that come with this is in that of interpolation. If a single channel holds four indices, these indices will be seen as a single value and interpolated as such, changing the indices in unpredictable ways. To remedy this we simply need to turn off interpolation and instead perform interpolation ourselves. In the shader we simply sample the four neighboring texels, split apart their individual indices, and perform linear interpolation on each index separately.

6.2.2.3 Per Patch Parameter Encoding. The downside to both the K-Means lookup texture and fitted function is that a representative patch must be used to configure the parameters and then be applied to all patches over the moon's surface. If we could instead configure the parameters per patch and save each patch's parameters, then the overall shadow error over all patches would be minimized. This can be accomplished by either using a K-Means lookup table texture for each patch, or using a texture to encode the coefficients of the fitted regression model. This textures could then be saved into an SCM TIFF where each patch would have its own texture. During shadow generation we would merely have to load in one additional SCM TIFF and sample from that to get the corresponding patch's parameters and apply the mapping to convert the index back into the horizon angle. Since there exists restrictions on the number of samplers allowed in the vertex and fragments shaders and since this would introduce an extra sampler when our normal approach already requires 23 samplers for shadow generation, this approach may need to be used with the 8 bit packing approach discussed previously.

6.2.2.4 SCM Depth and Retrieving Fourier Coefficients. The first several Fourier coefficients cover low frequency information. The ones after gradually cover higher and higher frequency data. Since the topology of the moon is generated based on the concept of using general displacement amounts at shallow depth and finer tuning the actual displaced position at deeper depths, it would make sense to combine these efforts and have shadows work the same way. At shallow depths, only the initial Fourier coefficients would be read and synthesized so as to provide broad ideas of where shadows lie. At deeper depths more coefficients would be read and synthesized so as to more accurately pinpoint where shadows are. As the user zooms in on an area, that area's topology along with the shadow calculations would become finer-grained at the same time; leading to a more harmonious and pleasant visual experience.

REFERENCES

[1] R. Kooima, "Spherical cube maps," 2012.

[2] L. Williams, "Casting curved shadows on curved surfaces," in *ACM Siggraph Computer Graphics*, vol. 12, pp. 270–274, ACM, 1978.

[3] F. C. Crow, "Shadow algorithms for computer graphics," in *ACM SIGGRAPH Computer Graphics*, vol. 11, pp. 242–248, ACM, 1977.

[4] R. Fernando, S. Fernandez, K. Bala, and D. P. Greenberg, "Adaptive shadow maps," in *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pp. 387–390, ACM, 2001.

[5] M. Wimmer, D. Scherzer, and W. Purgathofer, "Light space perspective shadow maps," in *Proceedings of the Fifteenth Eurographics conference on Rendering Techniques*, pp. 143–151, Eurographics Association, 2004.

[6] J.-M. Hasenfratz, M. Lapierre, N. Holzschuch, F. Sillion, A. GRAVIR, *et al.*, "A survey of real-time soft shadows algorithms," in *Computer Graphics Forum*, vol. 22, pp. 753–774, Wiley Online Library, 2003.

[7] M. Bunnell and F. Pellacini, "Shadow map antialiasing," *GPU Gems: Programming Tech Tricks for Real-Time Graphics*, 2004.

[8] K. Cherry and R. Kooima, "Multi-pass gaussian contact-hardening soft shadows," in *International Conference on Computer Graphics Theory and Applications (GRAPP 2015)*, pp. 274–280, Scitepress, 2015.

[9] U. Assarsson, M. Dougherty, M. Mounier, and T. Akenine-Möller, "An optimized soft shadow volume algorithm with real-time performance," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pp. 33–40, Eurographics Association, 2003.

[10] U. Assarsson and T. Akenine-Möller, "A geometry-based soft shadow volume algorithm using graphics hardware," *ACM Transactions on Graphics (TOG)*, vol. 22, no. 3, pp. 511–520, 2003.

[11] M. Salvi, K. Vidimče, A. Lauritzen, and A. Lefohn, "Adaptive volumetric shadow maps," in *Computer Graphics Forum*, vol. 29, pp. 1289–1296, Wiley Online Library, 2010.

[12] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan, "Ray tracing on programmable graphics hardware," in *ACM Transactions on Graphics (TOG)*, vol. 21, pp. 703–712, ACM, 2002.

[13] "Opengl." https://www.opengl.org/.

[14] "Directx graphics and gaming." https://msdn.microsoft.com/en-us/library/windows/desktop/ee663274

[15] D. Bookout, "Shadow map aliasing," *Navigation*, vol. 11, no. 42, p. 0, 2011.

[16] R. Dimitrov, "Cascaded shadow maps," *Developer Documentation, NVIDIA Corp*, 2007.

[17] A. Lauritzen, M. Salvi, and A. Lefohn, "Sample distribution shadow maps," in *Symposium on Interactive 3D Graphics and Games*, pp. 97–102, ACM, 2011.

[18] "Low pass filter." http://dictionary.reference.com/browse/low–pass–filter.

[19] "The gaussian kernel." http://www.stat.wisc.edu/ mchung/teaching/MIA/reading/diffusion.gaussian.kernel.pdf.pdf.

[20] R. Fernando, "Percentage-closer soft shadows," in *ACM SIGGRAPH 2005 Sketches*, p. 35, ACM, 2005.

[21] A. Klein, A. Nischwitz, and P. Obermeier, "Contact hardening soft shadows using erosion," 2012.

[22] S. Brabec and H.-P. Seidel, "Shadow volumes on programmable graphics hardware," in *Computer Graphics Forum*, vol. 22, pp. 433–440, Wiley Online Library, 2003.

[23] T. Heidmann, "Real shadows, real time. iris universe, 18: 28–31, 1991. silicon graphics," 1991.

[24] U. Assarsson and T. Akenine-Möller, "Occlusion culling and z-fail for soft shadow volume algorithms," *The Visual Computer*, vol. 20, no. 8-9, pp. 601–612, 2004.

[25] W. T. Reeves, D. H. Salesin, and R. L. Cook, "Rendering antialiased shadows with depth maps," in *ACM SIGGRAPH Computer Graphics*, vol. 21, pp. 283–291, ACM, 1987.

[26] W. Donnelly and A. Lauritzen, "Variance shadow maps," in *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pp. 161–165, ACM, 2006.

[27] A. Wagener, "Chebyshev's algebraic inequality and comparative statics under uncertainty," *Mathematical Social Sciences*, vol. 52, no. 2, pp. 217–221, 2006.

[28] M. McCool and E. Fiume, "Hierarchical poisson disk sampling distributions," in *Proceedings of the conference on Graphics interface*, vol. 92, pp. 94–105, 1992.

[29] C. Wyman and C. D. Hansen, "Penumbra maps: Approximate soft shadows in real-time.," in *Rendering Techniques*, pp. 202–207, 2003.

[30] J. Gumbau, M. Chover, and M. Sbert, "Screen space soft shadows," *GPU pro*, pp. 477–490, 2010.

[31] L. Bavoil, "Multi-view soft shadows," *Nvidia, Mar*, 2011.

[32] J. Kautz, W. Heidrich, and K. Daubert, *Bump map shadows for OpenGL rendering*. Max-Planck-Inst. fur Informatik, Bibliothek & Dokumentation, 2000.

[33] P.-P. J. Sloan and M. F. Cohen, "Interactive horizon mapping," in *Rendering Techniques 2000*, pp. 281–286, Springer, 2000.

[34] R. Copsey and dwmkerr, "Sharpgl," Jan 2013.

[35] "Swig." http://www.swig.org/.

[36] F. Zhang, H. Sun, L. Xu, and L. K. Lun, "Parallel-split shadow maps for large-scale virtual environments," in *Proceedings of the 2006 ACM international conference on Virtual reality continuum and its applications*, pp. 311–318, ACM, 2006.

[37] A. Terras, *Fourier analysis on finite groups and applications*. No. 43, Cambridge University Press, 1999.

[38] P. Henrici, *Applied and computational complex analysis, discrete Fourier analysis, Cauchy integrals, construction of conformal maps, univalent functions*, vol. 3. John Wiley & Sons, 1993.

[39] F. Cain, "How far is the moon from the sun?." http://www.universetoday.com/20566/how-far-is-the-moon-from-the-sun/, 2008.

APPENDIX A. SETTINGS.XML

The following tables shows a list of all settings found in the settings.xml file.

Table A.1: Moon Settings

Identifier	Data Type	Range or Restrictions	Default Value	Description
CacheLargeView	int	-	600	Size of SCM cache displayed when using a single viewing split
CacheSmallView	int	-	300	Size of SCM cache displayed when using multiple viewing splits

Table A.2: Projection Settings

Identifier	Data Type	Range or Restrictions	Default Value	Description
BaseFovy	double	-	60	The default fovy value for perspective projection. Can be overridden by HES pre-processing stages
DefaultZFar	double	-	$1x10^6$	The default far clipping plane value for perspective projection in the moon scene. Can be overridden by HES pre-processing stages
DefaultZNear	double	-	1	The default near clipping plane value for perspective projection in the moon scene. Can be overridden by HES pre-processing stages

Table A.3: Scene Settings

Identifier	Data Type	Range or Restrictions	Default Value	Description
BgColor	float[3]	0 to 1 per value	{0.0, 0.0, 0.0}	Clear color for OpenGL
DefaultLight	int	0 to number of shadow algorithms in scene	0	Which initial shadow algorithm to use when the app first opens
DisplayScreenHeightMult	float	-	1.0	Percentage of the monitor's height to use for the OpenGL context
DisplayScreenNum	int	0 to number of screens connected	1	Screen id to use to display app. Used for multiple monitors where zero indicates main display. Will default to zero if value is invalid
DisplayScreenWidthMult	float	-	1	Percentage of the monitor's width to use for the OpenGL context
FBAllowedVarNames	string[]	if name is present, it is added to feedback	null	Array of variable names in the shader that will be allowed to be used for transform feedback
InitLightRot	float[]	-	{-255, 0, 0}	Initial value for the light's rotation
ShowCache	bool	-	false	True to show the moon's SCM cache on screen when app first opens, false to hide it
UseLight	bool	-	true	True to use pixel lighting calculations when app first opens, false otherwise
UseShadow	bool	-	true	True to turn on shadow calculations when app first opens, false otherwise
UseTexture	bool	-	true	True to use texturing when app first opens, false otherwise

Table A.4: Shadow Map Settings

Identifier	Data Type	Range or Restrictions	Default Value	Description
DefaultBounds	float	-	$2x10^5$	Default value for light's orthographic bounds when rendering the shadow map
DefaultRes	int	256 to 8192	1024	Default shadow map resolution
MaxCascades	int	1 to 4	4	Default number of cascades in CSM
MapDataSmallView	int		200	Size to use when displaying map data on screen. This size is used when having multiple viewing splits
MapDataLargeView	int		50	Size to use when displaying map data on screen. This size is used when having a single viewing split

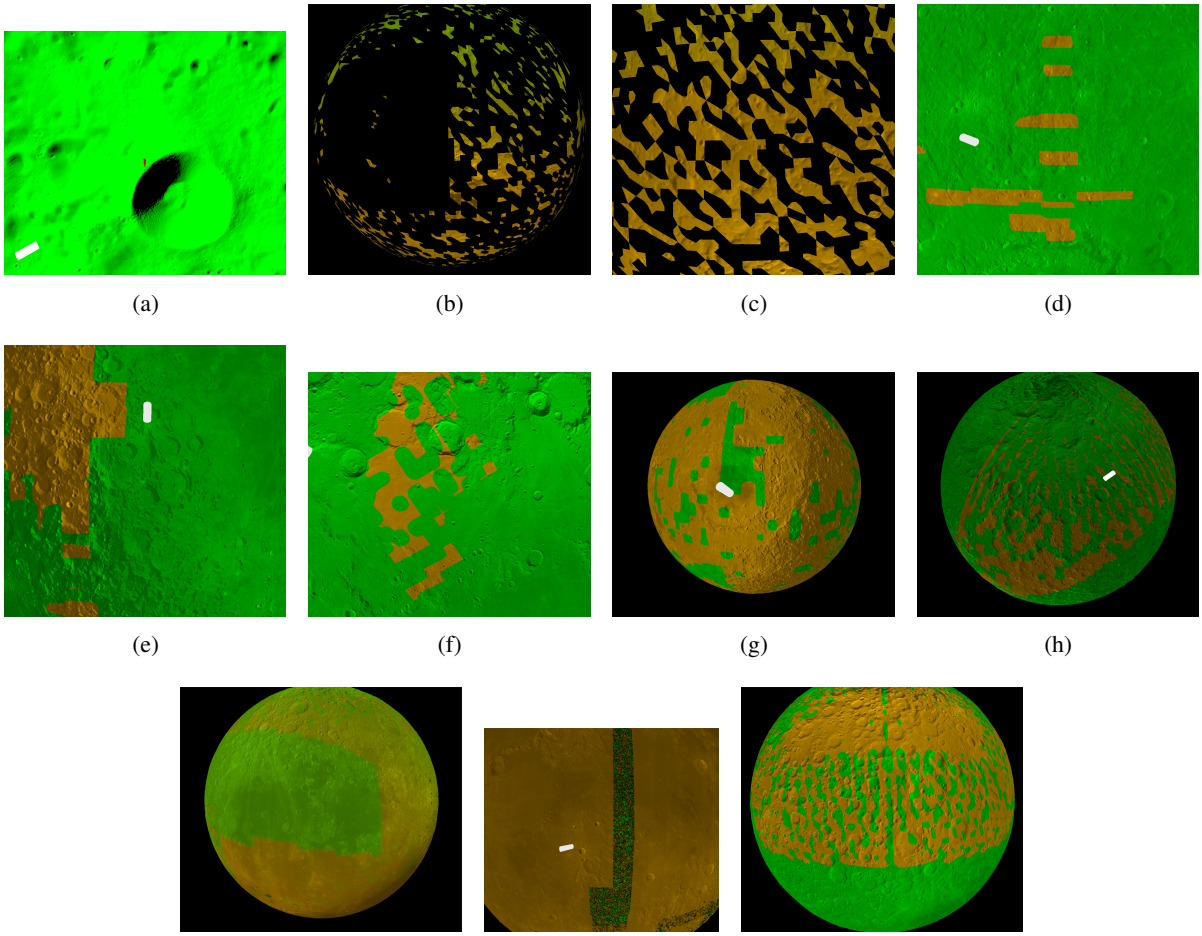
Table A.5: Texture Unit Settings

Identifier	Data Type	Range or Restrictions	Default Value	Description
Diffuse	uint	0 to number of active texture units	0	Texture unit to use for diffuse shading
Ambient	uint	0 to number of active texture units	1	Texture unit to use for ambient shading
Specular	uint	0 to number of active texture units	2	Texture unit to use for specular shading
Emissive	uint	0 to number of active texture units	3	Texture unit to use for emissive shading
Cascade0	uint	0 to number of active texture units	28	Texture unit to use for shadow map for first cascade. This value is set high since the SCM shadow TIFFs take up a lot of active units, and this default value prevents colliding with them
Cascade1	uint	0 to number of active texture units	29	Texture unit to use for shadow map for second cascade
Cascade2	uint	0 to number of active texture units	30	Texture unit to use for shadow map for third cascade
Cascade3	uint	0 to number of active texture units	31	Texture unit to use for shadow map for fourth cascade
Cascade0TexMat	uint	0 to number of active texture units	2	Index of texture matrix for first cascade
Cascade1TexMat	uint	0 to number of active texture units	3	Index of texture matrix for second cascade
Cascade2TexMat	uint	0 to number of active texture units	4	Index of texture matrix for third cascade
Cascade3TexMat	uint	0 to number of active texture units	5	Index of texture matrix for fourth cascade
Special0	uint	0 to number of active texture units	26	Texture unit to use for anything extra in an algorithm, such as ping-pong shading for post-processing effects
Special1	uint	0 to number of active texture units	27	Texture unit to use for anything extra in an algorithm, such as ping-pong shading for post-processing effects

101

APPENDIX B. HES ALGORITHM PROGRESSION

The following are screenshots portraying the journey taken from the time all parts (cache generation, angle calculation, angle storage, and usage) of the HES algorithm were implemented, but not yet working, up until the point where the full algorithm started to work and produce actual shadows. The figures are in proper chronological order and show the progression of the development of the algorithm.



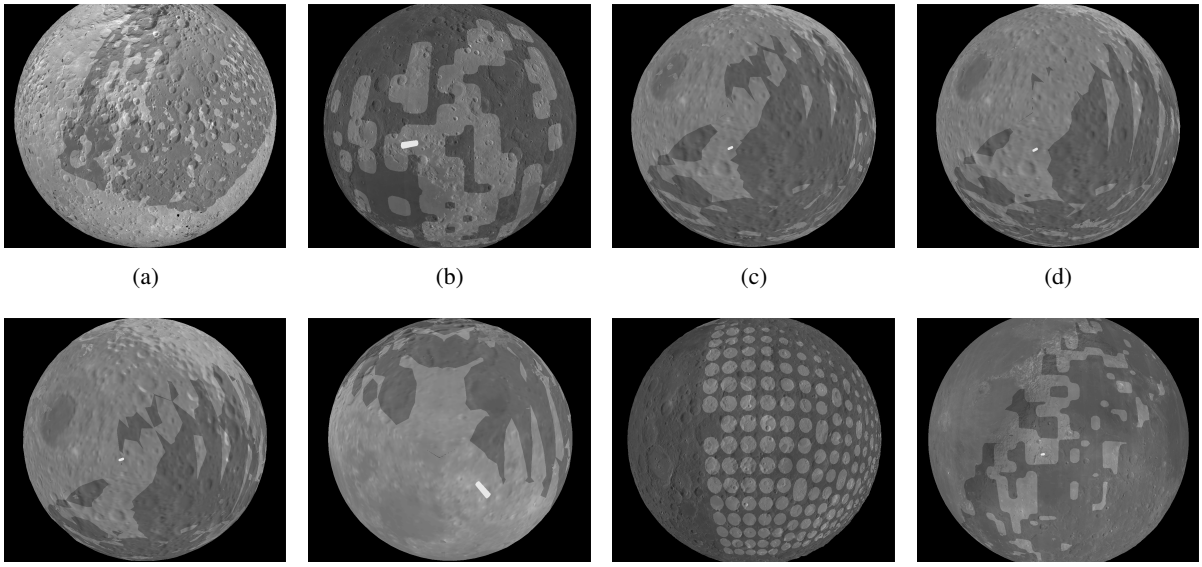
(a) (b) (c) (d)

(e) (f) (g) (h)

(i) (j) (k)

Coloration was used to help indicate casted shadows. Anywhere the algorithm thinks there is a shadow, an orange color is used, otherwise green is used. This shows the initial results when all the parts of the HES algorithm were completed. The images, in order, show the progression of the early stages of the algorithm. During these stages there were several bugs in multiple pre-processing parts that prevented accurate shadow calculations. (a) shows the very first appearance of something resembling shadows, indicated by the small orange spot in the center. (b) and (c) show a problem where vertices were seeming to be clipped. This particular issue went on for awhile before it was fixed. The problem from (d) to (i) and (k) was found to be an issue in the shader during horizon angle calculation where a uniform set for debugging purposes was not set back. This caused the vertex positions read back to the cpu to be in object space as opposed to a space where the vertex we are calculating for is in the center. Since the positions were in object space, the vertex positions read back were always the same regardless of the target vertex and this caused incorrect circles to be generated as shadows. Another problem found at this stage was the way the view and model transformations were handled when calculating horizon angles. The method at the time would not work for all areas of the moon and would sometimes be facing the wrong direction when gathering horizon data. (j) shows some noise which was originally thought to be actual shadows trying to be generated correctly but later was believed to be just a z-fighting issue.

Figure B.1: Progression of HES Algorithm

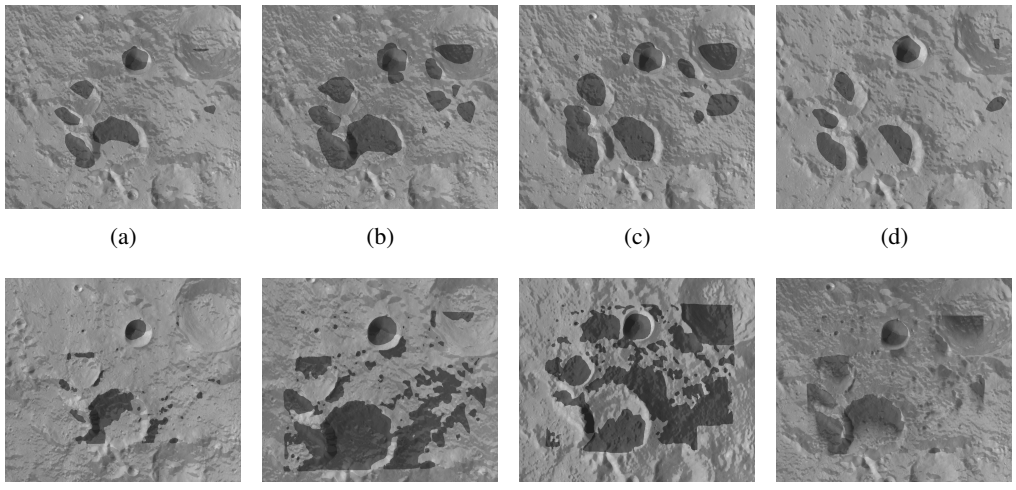


(a) (b) (c) (d)

(e) (f) (g) (h)

This set of images, in order, shows the progression after Figure B.1. False coloration has been removed. Several rounds of testing and bug fixing occurred during this stage, but shadow calculations were still not correct. These images (especially (g)) show the circle problem explained in Figure B.1 as well as other issues such as sampling from the wrong SCM TIFFs , incorrect generation of input TIFF images (via calculating latitude and longitude from a normal vector incorrectly and generating ill-formed equirectangular projections), and incorrectly specifying the depth in the scmtiff conversion tool. (h) shows what could be actual shadows but the range was too large. After toning down the range to just a few degrees in both latitude and longitude, we got the images shown in Figure B.3.

Figure B.2: Progression of HES Algorithm



(a) (b) (c) (d)

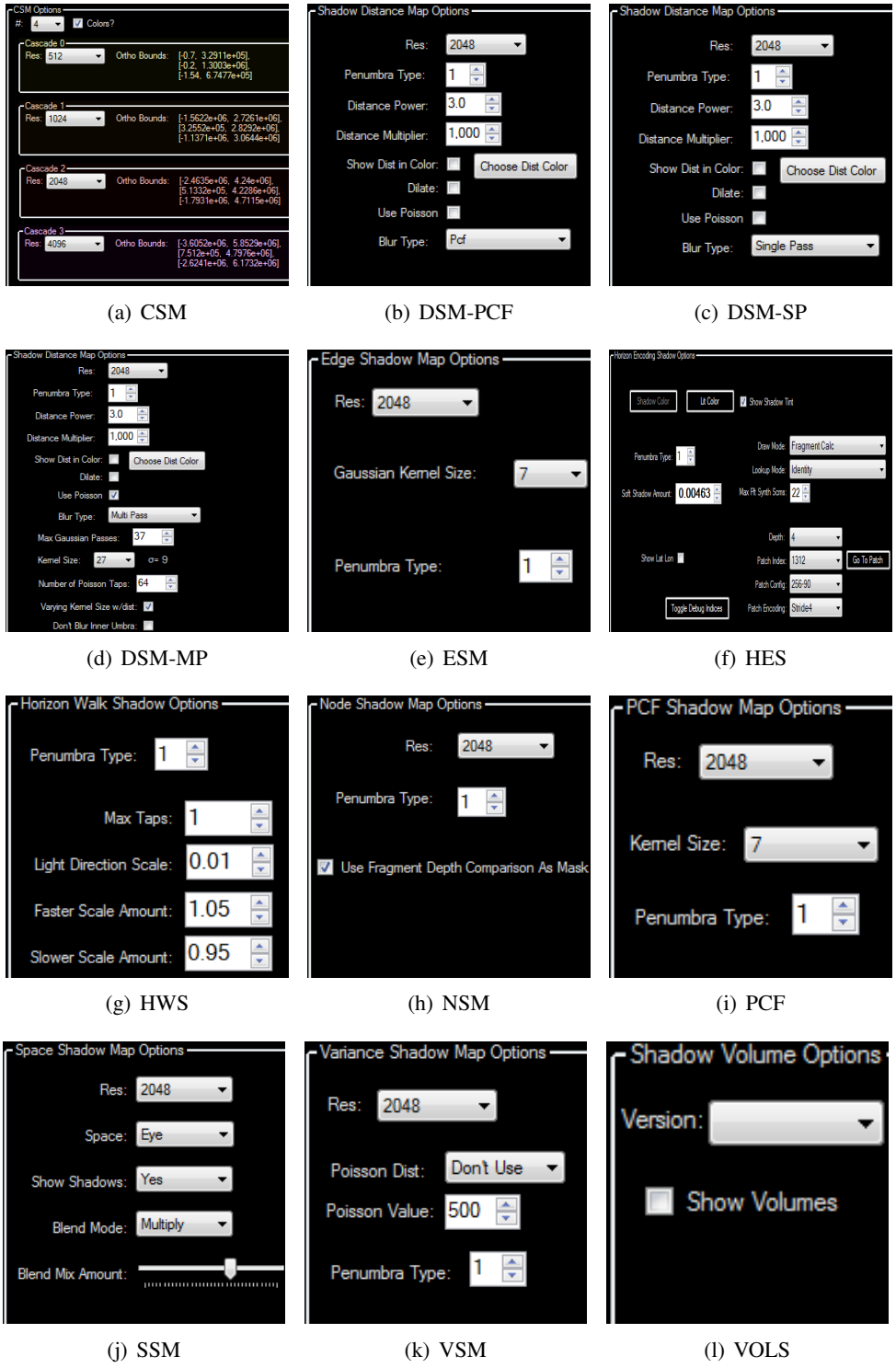
(e) (f) (g) (h)

After Figure B.2 and several more rounds of testing and bug fixing, shadow calculations became a lot more accurate. These images show the final stages of testing with the HES algorithm. These images show shadow calculations at different lighting directions.

Figure B.3: Progression of HES Algorithm

APPENDIX C. GUI SHADOW OPTIONS

Figure C.1 shows all the options available for each shadow algorithm that can be set through the GUI .



(a) CSM (b) DSM-PCF (c) DSM-SP

(d) DSM-MP (e) ESM (f) HES

(g) HWS (h) NSM (i) PCF

(j) SSM (k) VSM (l) VOLS

Figure C.1: Shadow Algorithm GUI Options

APPENDIX D. FULL ALGORITHMS

Shadow Maps

```
// use light's model transformations as the view transformations for the scene
// thereby setting up to render in light space
Apply the inverse of the light's model transformations to the ModelView matrix

// bind the off-screen buffer
Bind light's fbo that has a depth buffer attached to it

// render to depth buffer the scene in light space. the shader for this
// should be simple and should omit visuals such as texturing,
// lighting, etc., since we only need the depth information
Render scene into depth buffer

// Save information to project pixel into light space
// when rendering in camera space
Save light's model transformations and send to shader

// reset ModelView
Clear ModelView matrix

// prepare to render to screen's normal back buffer
Unbind light's fbo

// render in camera space
Apply camera's view transformations to ModelView matrix

// render scene with shadow information
Render scene
  In Shader:
    // project vertex position like normal
    Send vertex from object space to eye space and save the position in a varying

    // project vertex position into light space. this is done by using the
    // vertex's model matrix, the inverse of the light's model matrix as a
    // view matrix, and the scene's projection matrix
    Send vertex from object to light space and save position in a another varying

    // get shadow map value
    Use fragment's light space position as projected texture coordinate
      into shadow/depth map

    // compare fragment's distance from light source to shadow map value
    Compare fragment's Z coordinate to shadow/depth value from texture lookup
```

```
    // if distance is equal or smaller
    If Z is equal or smaller then
      // fragment is lit
      Render fragment with full lighting

    // if distance is larger
    Else
      // fragment is in shadow
      Render fragment with ambient light only

// show what you rendered
Swap buffers
```

Variance Shadow Maps

```
// use light's model transformations as the view transformations for the scene
// thereby setting up to render in light space
Apply the inverse of the light's model transformations to the ModelView matrix

// bind the off-screen buffer
Bind light's fbo that has a color and depth buffer attached to it

// render the scene in light space.
// the depth value and the depth value squared should be written into separate
// channels in the color buffer.
// omit visuals like in normal shadow maps as we only need the depth information
Render scene into color buffer

// Save information to project pixel into light space
// when rendering in camera space
Save light's model transformations and send to shader

// reset ModelView
Clear ModelView matrix

// prepare to render to screen's normal back buffer
Unbind light's fbo

// render in camera space
Apply camera's view transformations to ModelView matrix

// render scene with shadow information
Render scene
  In Shader:
    // project vertex position like normal
    Send vertex from object space to eye space and save the position in a varying
```

```
    // project vertex position into light space. this is done by using the
    // vertex's model matrix, the inverse of the light's model matrix as a
    // view matrix, and the scene's projection matrix
    Send vertex from object to light space and save position in a another varying

    // get shadow map value
    Use fragment's light space position as projected texture coordinate
      into shadow/depth map

    // compare fragment's distance from light source to shadow map value
    Compare fragment's Z coord to depth value (non-squared) from texture lookup

    // if distance is equal or smaller
    If Z is equal or smaller then
      // fragment is lit
      Render fragment with full lighting

    // if distance is larger
    Else
      // fragment is either fully or partially in shadow
      Let depth = shadow map's depth value
      Let depthSquared = shadow map's squared depth value
      Let variance = depthSquared - depth * depth
      Let distDiff = fragment's distance from light - depth
      Let shadowColor = variance / (variance + distDiff * distDiff)
      Multiply fragment's final color by shadowColor
      Add ambient lighting to fragment's color
      Render fragment

// show what you rendered
Swap buffers
```

Shadow Volumes (ZPass Method)

```
// ready our stencil buffer
Enable stencil test and enable writing to stencil buffer
Clear stencil buffer

// don't overwrite values in color or depth buffers
Disable writing to color and depth buffers

// we want to render only front or back faces at a time
Enable face culling

// if shadow geometry goes past near or far clipping plane, clamp it
Enable depth clamp
```

```
// increment the stencil buffer for each intersection with a front face
// of a shadow volume
Set stencil to increment on depth pass
Set culling to cull back faces
Render only shadow volumes

// decrement the stencil buffer for each intersection with a back face
// of a shadow volume
Set stencil to decrement on depth pass
Set culling to cull front faces
Render only shadow volumes

// go back to writing to the color and depth buffers and lock the stencil buffer
// from being written to
Enable writing to the depth and color buffers
Disable writing to the stencil buffer

// no more need to do face culling or depth clamping
Disable face culling
Disable depth clamp

// render everything in ambient light
Set stencil test to always pass
Render scene with only ambient light

// render only fragments not inside a shadow volume with full lighting.
// setting it to render where stencil is zero means that at that fragment,
// we have entered the same number of shadow volumes as we have exited, meaning
// we are currently outside any shadow volumes
// (similar to tracking matching parenthesis in text)
Set stencil to render only where stencil bit is equal to zero
Render scene with full lighting

// show our work
Swap buffers
```

VITA

Kevin Anthony Cherry, a native of Baton Rouge, Louisiana, received his bachelor's and master's degree at Louisiana State University in 2008 and 2011, respectively. Thereafter, he taught video game design to high school and college students around Louisiana and Washington. He has always wanted to receive a doctorate and loves learning about and teaching computer science. He will receive his Ph.D. degree in December 2015 and plans on continuing his research and teaching as a career upon graduation.