CHALMERS

FAST TRIANGLE RASTERIZATION USING IRREGULAR Z-BUFFER ON CUDA

Master of Science Thesis in the Programme of Integrated Electronic System Design

Wei Zhang
Ivan Majdandzic

CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Department of Computer Science and Engineering
Göteborg, Sweden. June 2010

Fast Triangle Rasterization Using Irregular Z-Buffer on CUDA

Wei Zhang,
Ivan Majdandzic.

Examiner: Ulf Assarsson.

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46(0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden June 2010

Abstract

In 3D rendering, shadows provide valuable visual information to viewers, and increase the level of realism in the rendering outcome. Therefore, shadow generation become a fundamental task in modern real-time rendering. Shadow mapping, one of the most common shadow generation techniques in real-time rendering, due to its inherent flaw is not capable of generating aliasing-free shadows. The irregular Z-Buffer algorithm revealed in 2004 eliminates the resolution mismatch problem in conventional shadow mapping, but it is not directly supported by the existing graphics hardware and lack of efficient software implementations. With the appearance of CUDA, the programmability of current graphics hardware has been drastically improved. It allows developers to leverage the enormous computation horsepower that resides in the current graphics hardware in a more flexible way.

We provide a complete description to our irregular Z-Buffer based shadow mapping software implementation on CUDA. The rendering system we built is running completely on GPUs. It is capable of generating aliasing-free shadows at a throughput of dozens of million triangles per second.

CONTENTS

1 INTRODUCTION

1.1 Shadows

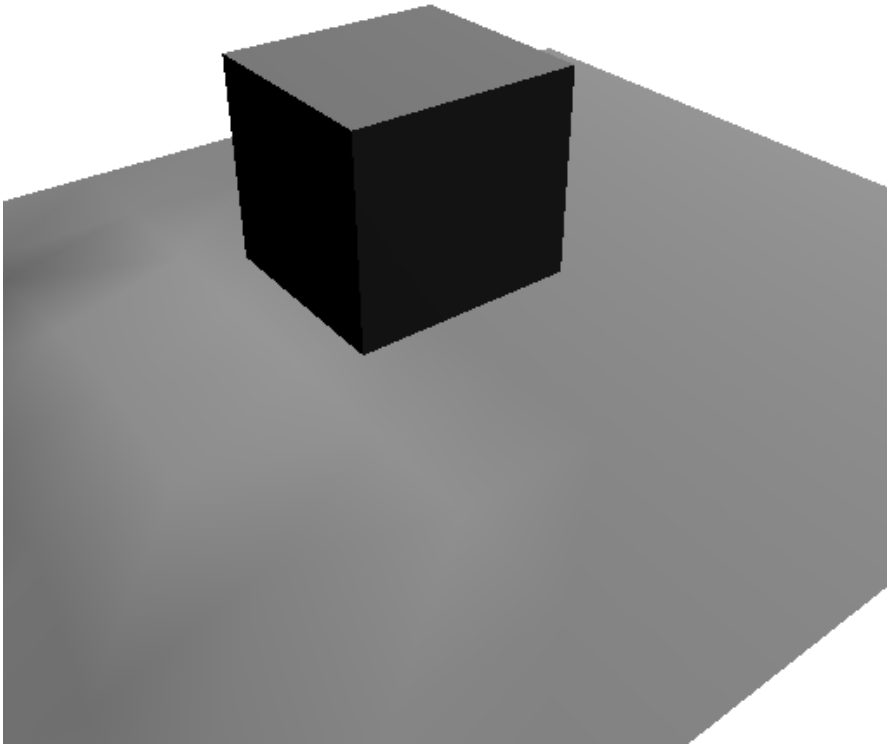


Figure 1.1. A cube with no shadow

By looking at the illustration above (Figure 1.1), please ask yourself a question:" How high is the cube from the ground?" Apparently, there is no simple answer, because it is not possible to tell whether the cube is close to viewers hanging above the ground or it is actually big and lying on the ground. This is due to the lack of visual information in the illustration.

Shadows provide visual information such as: shape of the receiver, shape of the surface of the blocker, relative object position, and light position, which lead to a more realistic 3D scene. The illustration below (Figure 1.2) would probably make it easier to answer the question we had.

Therefore, shadow rendering is a very important aspect of 3D rendering. In this thesis work, we focus on shadow generations in real-time rendering.

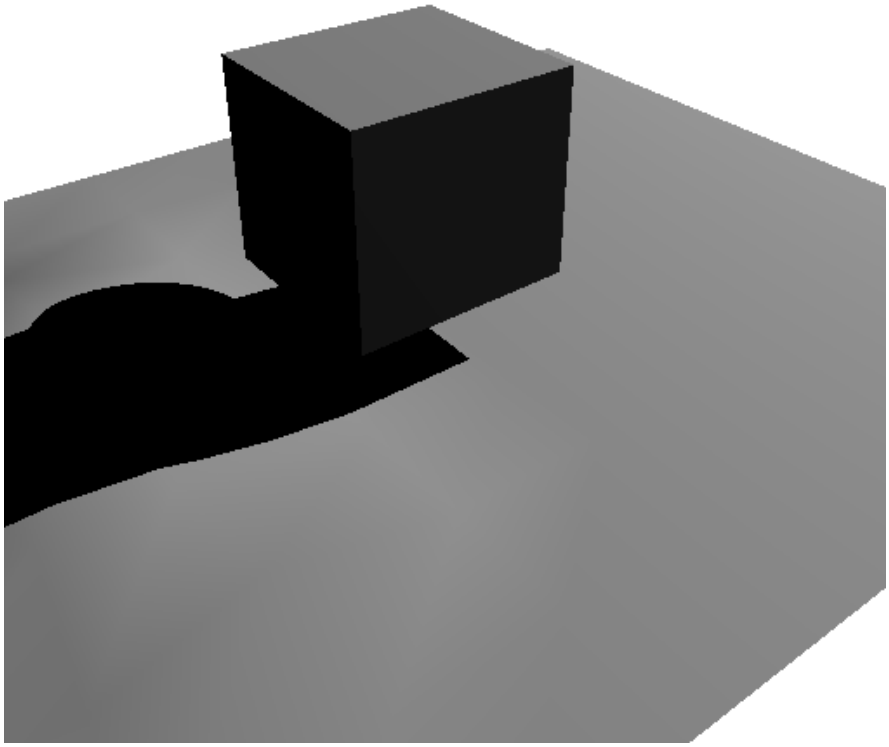


Figure 1.2. A cube with shadow

1.2 Conventional Shadow Mapping

Shadow mapping is one of the most common techniques to generate shadows in real-time rendering. The algorithm was first introduced by Lance Williams [Wil78]. This approach is very simple and requires two rendering passes. In the first pass, the depth of the scene is rendered from the light position into a depth buffer (the shadow map), and then, in the second pass the scene is rendered from the eye position and compared with the shadow map. When rendering from the eye position, each point needs to be projected onto the light image plane and the x- and y- coordinate are used for lookup in the shadow map. The comparison of the point's depth with the depth in the shadow map gives a solution to tell if the specific point is in shadow or not.

1.2.1 Artifacts

Figure 1.3 illustrates how a point is projected onto the light-view image plane and the depth value used to compare with from the light's point of view. One can easily

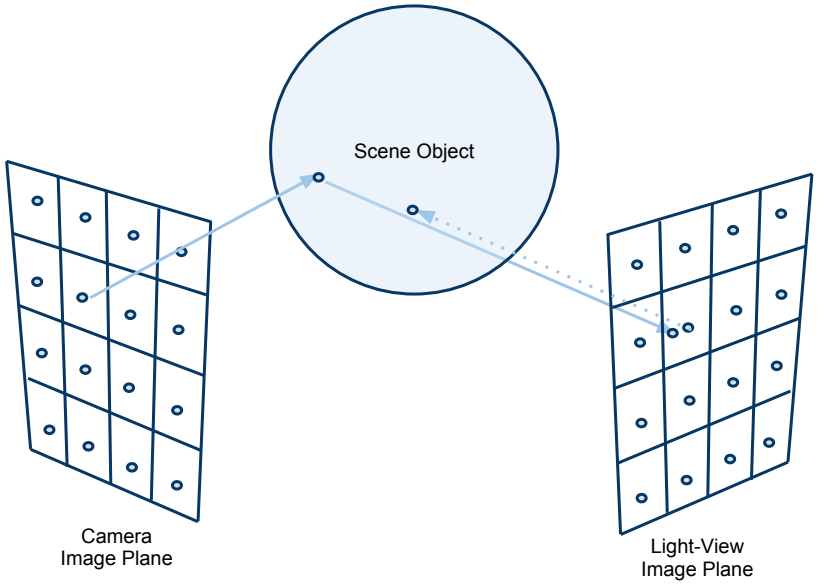


Figure 1.3. Shadow mapping

tell that the actual point that we want to compare the depth with is not aligned with the pixel seen from the light position. Thus, the closest pixel position is chosen. The deviation between the desired sample point position and the regular pixel position causes resolution mismatch in the generated shadow map, which is visualized as unwanted artifacts in the later stage (Figure 1.4).

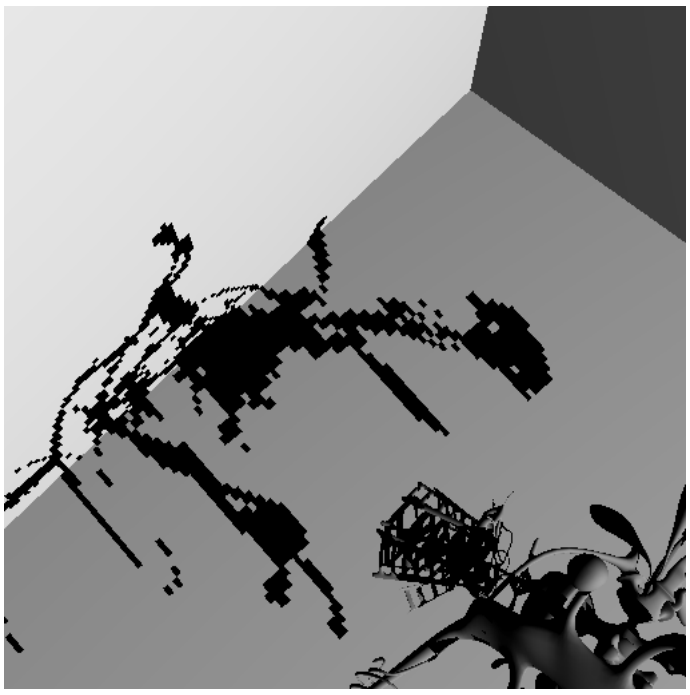


Figure 1.4. Shadow mapping artifacts

1.3 Artifact Free Shadow Mapping

One attempt to remove artifacts from shadow mapping is to increase the shadow map resolution. Other approaches have been proposed as well, such as Cascaded Shadow Maps (CSMs) [Eng06], Perspective Shadow Maps (PSMs) [SD02], and Logarithmic Perspective Shadow Maps (LogPSMs) [LGQ+08]. All of them tend to minimize the artifacts, instead of removing the artifacts completely.

To eliminate artifacts, there must be an exact point lookup for each point seen from the eye space. One possible solution to eliminate the artifacts in shadow mapping is to find a way to produce a 1:1 map such that each pixel point in the eye space corresponds to the exact sample point in the light space. Figure 1.5 illustrates that the pixel points in the light space which need to be rendered are view and geometry dependent, and thus, not longer distributed uniformly. Because the fixed function unit on the GPU (i.e., the rasterizer which handles the conversion from vertex data into pixel data) is not capable of handling irregular points, there is a need to have software implementations for the irregular approach.

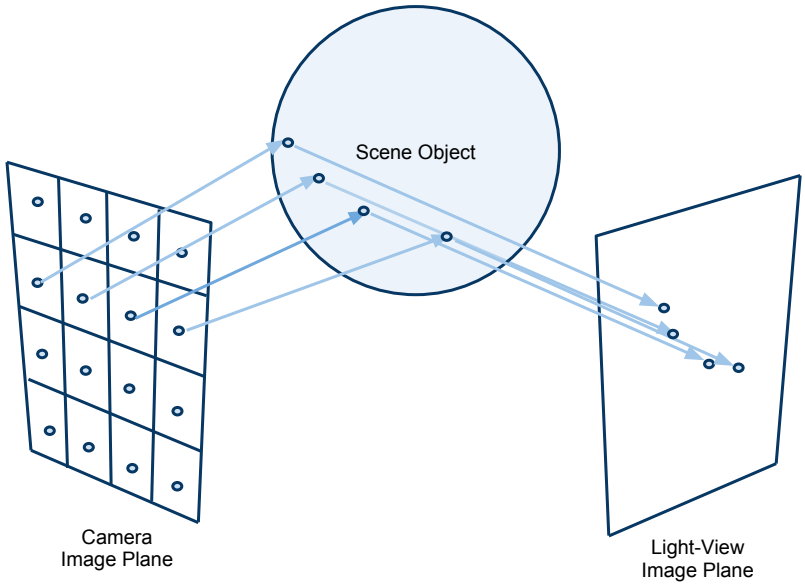


Figure 1.5. Irregular shadow mapping

The new approach to generate artifact-free shadow mapping is somewhat different. First, the scene is rendered from the eye position to extract the information of all the sample points. The sample points are then used to render the scene from the light position, and the irregular shadow map is generated. The last pass renders the scene from the eye position and utilizes the irregular shadow map to perform exact lookups for each corresponding point. This algorithm is known as the Irregular Z-buffer [JMB04] or Alias-Free Shadow Maps [AL04].

Johnson et al. proposed a hardware architectural solution, while Aila et al. outlined

a hierarchical software implementation on CPUs and extended the basic algorithm for semi-transparent shadow casters and receivers. In 2008, a GPU implementation was presented by Sintorn et al. [SEA08] including the functionality of generating soft shadows. So far a more efficient software implementation of irregular shadow mapping on GPUs, which can be directly equipped in real-time rendering, is still anticipated.

1.4 Triangle Rasterization

In real-time rendering, triangles are treated as the basic geometrical primitives, and our shadow rendering system is based on a triangle rasterization process. Therefore, it is necessary to put up a brief introduction to triangle rasterization [AM07].

1.4.1 Edge Functions

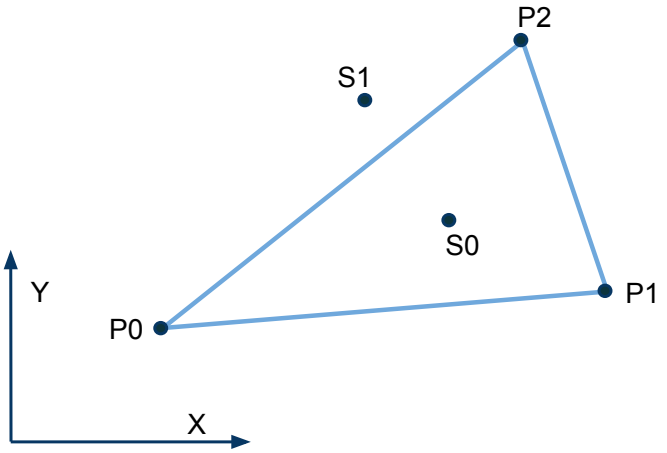


Figure 1.6. A triangle, $\triangle \mathbf{P}^0\mathbf{P}^1\mathbf{P}^2$, and two sample points, $\mathbf{S}^0$ and $\mathbf{S}^1$, in view image plane

Since in the scope of this thesis work we are dealing with rasterizations for irregular sample points, the regular pixels on the screen are replaced with random sample points on the view image plane (Figure 1.6).

In Figure 1.6, the triangle, $\triangle \mathbf{P}^0\mathbf{P}^1\mathbf{P}^2$ is defined by three vertices, $\mathbf{P}^0$, $\mathbf{P}^1$, and $\mathbf{P}^2$, where $\mathbf{P}^i = (p_x^i, p_y^i)$. The edge function through $\mathbf{P}^0$ and $\mathbf{P}^1$ can be described as:

$$e(x, y) = -(p_y^1 - p_y^0)(x - p_x^0) + (p_x^1 - p_x^0)(y - p_y^0) = ax + by + c. \tag{1.1}$$

Therefore, for each triangle, all the three edge functions are:

$$
\begin{aligned}
e_0(x,y) &= -(p_y^2 - p_y^1)(x - p_x^1) + (p_x^2 - p_x^1)(y - p_y^1) = a_0x + b_0y + c_0 \\
e_1(x,y) &= -(p_y^0 - p_y^2)(x - p_x^2) + (p_x^0 - p_x^2)(y - p_y^2) = a_1x + b_1y + c_1 \\
e_2(x,y) &= -(p_y^1 - p_y^0)(x - p_x^0) + (p_x^1 - p_x^0)(y - p_y^0) = a_2x + b_2y + c_2. \quad (1.2)
\end{aligned}
$$

By using Equation 1.2, we can perform visibility tests on sample points against the triangle. For example, there are two sample points in Figure 1.6, $\mathbf{S}^0$ and $\mathbf{S}^1$, where $\mathbf{S}^i = (s_x^i, s_y^i)$. We can put $\mathbf{S}^i$ into Equation 1.2 and evaluate all edge functions. If the values of all edge functions are bigger than zero, the sample point is covered by the triangle; otherwise it is not.

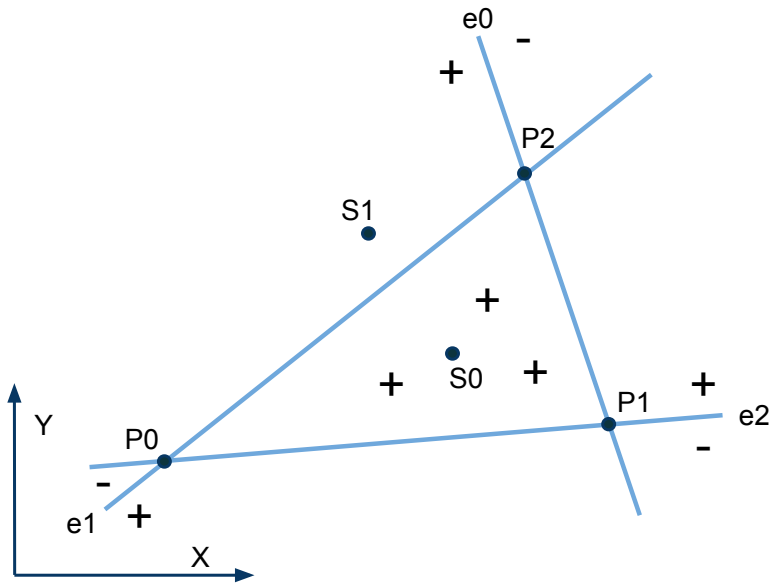


Figure 1.7. Triangle edge function evaluation

As illustrated in Figure 1.7, for $\mathbf{S}^0$ the evaluation results of all three edge functions are positive, so it is located inside the triangle, meaning the triangle is visible at this sample point position. For $\mathbf{S}^1$, it is not the case. Thus, the triangle cannot be seen at the position of $\mathbf{S}^1$.

1.4.2 Depth Interpolation

In order to transfer the attributes of each vertex consisting the triangle into information at the sample point position, we need to perform interpolations so as to obtain the correct depth value (other attributes as color or texture coordinates is not the concern of shadow mapping, and therefore will not be discussed).

Barycentric coordinates [AM07] is commonly employed to perform interpolations, which can be derived from the edge functions we had before:

$$\begin{aligned} \bar{u} &= \frac{e_1(x,y)}{2A_\triangle} \\ \bar{v} &= \frac{e_2(x,y)}{2A_\triangle} \\ \bar{w} &= 1-\bar{u}-\bar{v}, \end{aligned} \tag{1.3}$$

where $A_\triangle$ is the signed area of the triangle which can be calculated as:

$$A_\triangle = \frac{1}{2}((p_x^1 - p_x^0)(p_y^2 - p_y^0) - (p_x^2 - p_x^0)(p_y^1 - p_y^0)). \tag{1.4}$$

Assume that the depth value of vertex $\mathbf{P}^i$ is d_i. The depth value at a desired sample point position, $\mathbf{S} = (s_x, s_y)$, can be calculated with the barycentric coordinates from Equation 1.3 by using the following equation:

$$d(x,y) = (1-\bar{u}-\bar{v})d_0 + \bar{u}d_1 + \bar{v}d_2 = d_0 + \bar{u}(d_1 - d_0) + \bar{v}(d_2 - d_0). \tag{1.5}$$

1.5 Rasterization on CUDA

CUDA was introduced by NVIDIA back in 2007 [NVI10]. It is essentially a heterogeneous computing platform, which provides APIs that can be used to implement general-purpose computations on graphics hardware. CUDA reveals huge opportunities for developers to leverage the GFLOPS computational horsepower and massive amount of parallelism that reside on current graphics hardware. Complex computation problems can now be solved on GPUs in a fraction of the time required on CPUs.

A typical rasterization process consists of a large number of individual visibility tests between a pixel and a triangle. Although rasterization is accelerated by a dedicated hardware unit on existing graphic hardware, the inherent characteristic of rasterization shows its potential to be parallelized and adapted onto CUDA programming model. We intend to seize the opportunity and explore how far our irregular shadow mapping implementation can achieve in terms of performance on CUDA.

2 ALGORITHM OVERVIEW

In this chapter, we provide a brief introduction to our rasterization based irregular shadow mapping implementation on CUDA. The outline of our design including OpenGL rendering passes and various CUDA processing kernels will be unveiled.

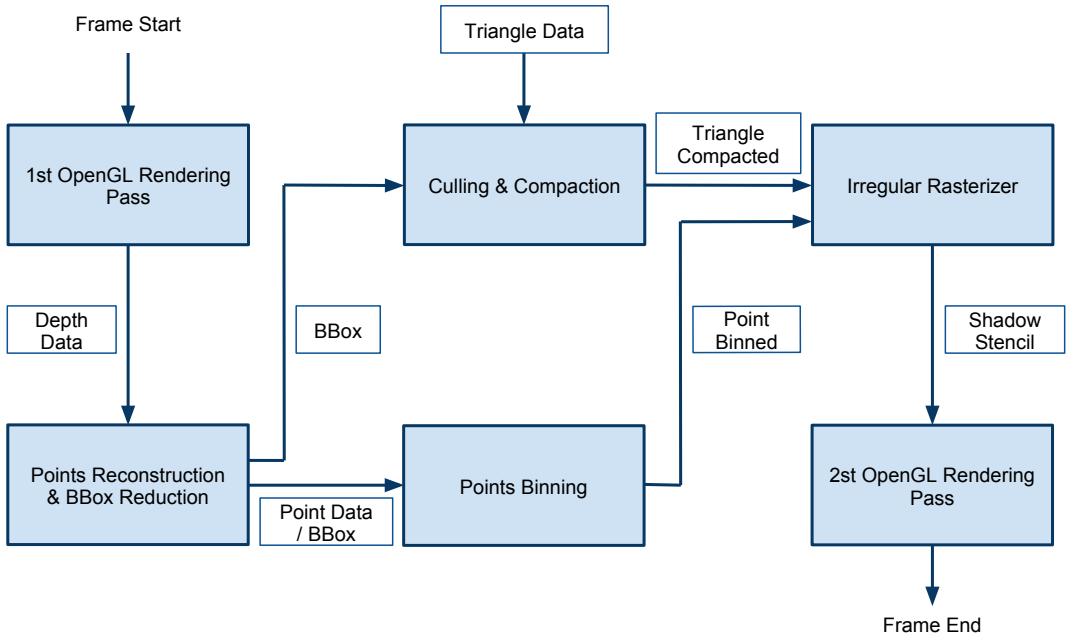


Figure 2.1. Algorithm overview

The general processing steps and data flows are illustrated in Figure 2.1. All the rendering steps shown in the figure are performed each frame, meaning the rasterizer is able to take on dynamically updated pixel and triangle data and produce correct rendering outcome accordingly each frame.

The whole system is conceptually partitioned into six parts (in the figure they are denoted by the blocks filled with light blue). Two OpenGL rendering passes are required as the first and the last steps separately. All the other four parts are implemented as CUDA kernels. They are coded in CUDA 3.0 so as to benefit from the improved interoperability between OpenGL and CUDA. Arrows illustrates the data flows between different parts, and the blocked text label beside each arrow shows what kind of data are passed from one part to the other. Triangle data are extracted from model files and uploaded into the GPU memory during initialization. All the other intermediate data shown in the figure are pre-allocated on the GPU at the initial phase of the application, and being updated during each frame. The data exchange between different parts of the system takes place entirely in the GPU memory.

2.1 1st OpenGL Pass and Point-Reconstruction

The rendering of a single frame starts with the first OpenGL rendering pass, which generates the depth values of desired pixels. Based on the provided depth data, pixels are then reconstructed in the point-reconstruction stage, where the world space 3D coordinates of the pixels seen on the screen are restored. After the world space coordinates are obtained, the reconstructed pixel points are treated as point or vertex primitives. They are then transformed into the light-view space and projected onto the light-view image plane, in the same way as in OpenGL when vertices are transformed from the world space to the camera space and then projected onto the camera image plane.

Up until now, the projected pixel points can be seen as sample points on the light-view image plane. The reason why we use the term 'sample points' is because the distribution of those points is similar to an irregular sampling scheme across the light-view image plane. The light-view image plane is equivalent to a virtual screen. The location of each sample point on the virtual screen defines where a visibility test will be performed, and geometrical information will be converted into pixel information.

Since the light-view image plane is a virtual screen that does not have a fixed size, view-port scaling can be omitted. As long as sample points and triangle data are transformed and projected onto the same image plane in the same way, rasterization should be working just fine, and visibility tests should be able to generate its correct results.

Since the light-view image plane, the virtual screen we are working on, does not have a fixed size and position, there is a need to calculate the exact bounding box of all the sample points on the plane. We will discuss the purpose of calculating the bounding box in more detail in the following chapters. Basically, it is used to build acceleration data structures and eliminate unnecessary computations. As a part of the point-reconstruction, a parallel reduction is performed to calculate the exact bounding box of all sample points.

2.2 Point Binning

It is evident that the irregular rasterization process is camera-view dependent, meaning it depends on the number of pixels that we are going to render on the screen and the angle with which we are looking into the scene. In addition, the rasterization process is also scene dependent. The larger the number of triangles in the scene the more potential visibility tests that we have to perform against them. A rough estimation of the complexity of the irregular rasterization can be formulated as the number pixels times the number of triangles.

In order to accelerate the process and avoid unnecessary computations, we build

an acceleration data structure for the sample points: a 2D uniform grid. The construction of the grid is based on the position of each sample point on the light-view image plane. We evaluate the X- and Y-coordinates of each sample point and put it into the corresponding cell or bucket it belongs to. A more figurative name of this procedure is binning, commonly referred as data binning, where data values are categorized into different bins.

We provide two different methods for point binning: the first one is based on the atomic operations supported by CUDA (computation capability 1.1); for older hardware where atomic operations are not available, we have another implementation built upon radix sort. Similar to the particles example in CUDA SDK [Gre08], as stated in the SDK document, Simon Green also used two approaches to build a uniform grid for their particle system: one is based on atomic operation and the other one is based on radix sort. Our radix sort binning algorithm is more or less the same as what Simon did for their particles system. Both of us employed the radix sort algorithm described in [SHG09], but the difference is we are building a 2D uniform grid on the light-view image plane, and their particle system resides in a 3D uniform grid. For the atomic operation approach, they did not fully explore the potential of atomic operations. We manage to utilize the support of atomic operation on the shared memory (computation capability 1.2 and higher) and drastically improved the performance so as to make it more efficient than the sorting based one.

2.3 Triangle Culling and Compaction

Geometry primitive culling is a common speedup technique in real-time rendering. Alike with standard OpenGL rendering pipeline, our irregular shadow map design is also driven by rasterization. So we can employ the same culling techniques as in OpenGL, such as back face culling and view frustum culling in our system to enhance the performance.

Initially, triangles are transformed and projected from the model space onto the light-view image plane, the same way as we did for the sample points. After that, the signed area of each triangle is evaluated, during which the sequence of the three vertices consisting the triangle is determined. If back face culling is enabled, back-facing triangles, the vertices order of which is clockwise, are culled away. The pre-calculated bounding box of the sample points determines the region of the virtual screen on the light-view image plane. Triangles located outside of the bounding box are of no interest in our rasterization, so they will be marked invalid as well.

After the culling evaluations, triangle data are compacted by a parallel compaction process, which is primarily a parallel prefix sum [SHG08] plus a data reordering stage. More specific information regarding triangle culling and compaction is included in chapter 4.

2.4 Irregular Rasterizer

Up until this point, the sample point and triangle data have been trimmed and packed in a good form, and are ready to be processed by the irregular rasterizer. In this stage, triangle data are fetched from the global memory, edge equations are set up, and irregular sample points are traversed and evaluated in a triangle bounding box traversal fashion.

Depth interpolation is performed whenever a triangle is rasterized on a sample point position. If the triangle happens to be the shadow caster of the sample point, meaning it is closer to the light source than the point, the sample point will be marked as in-shadow in the shadow stencil buffer. The size of the shadow stencil buffer is equal to the number of pixels we have on the screen. Each pixel on the screen has its unique in-shadow mark that indicates whether this pixel is in shadow or not.

In the last step, the second OpenGL rendering pass, a lookup from the shadow stencil buffer is invoked for each poixel. It checks whether the fragment is in shadow or not, and computes the correct shading for that fragment.

2.5 Wrap-up

After all these six steps, shadows shall be rendered correctly into the frame buffer, and eventually be displayed on the screen.

The following chapters, including chapter 3, 4 and 5, work as a more comprehensive description of our irregular shadow mapping design. Depth generation, point-reconstruction, and parallel reduction will be explained more in-depth in chapter 3. Chapter 4 will explain a variety of acceleration techniques we have deployed, which includes point binning, triangle culling and compaction. Chapter 5 will focus on the implementations of our irregular rasterizer.

Design Specifications

Here is the development environment that we have employed in our design:

- CUDA Version: 3.0
- API: CUDA C
- Hardware: Geforce GTX 260.

For demonstration purpose, the resolution we tested is:

- Resolution: 512 x 512.

This is parameterized in the design, so it can be adjusted in compile time.

3 POINT-RECONSTRUCTION

As an input to our rasterizer, the irregular sample points need to be reconstructed from the first standard OpenGL rendering pass. This step is done by: for each pixel on the screen, we take the pixel position and depth information in the screen space, apply it with the corresponding transformation matrices to transform the sample points from the camera image plane back to world space, and then project them onto the light-view image plane (Figure 3.1). A more in-depth description of our approach is presented in this chapter.

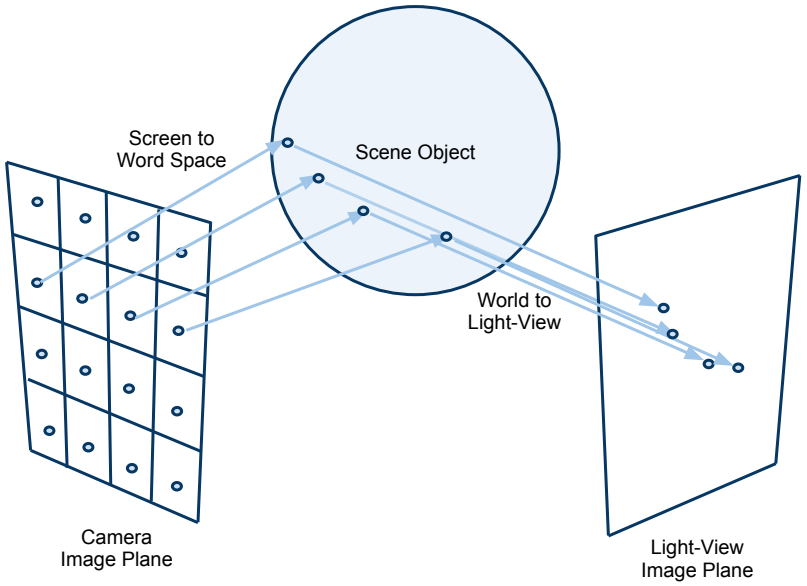


Figure 3.1. Sample points reconstruction

3.1 Depth Values

In the camera image plane, pixel positions are fairly easy to obtain. It is simply a matter of shifting the value of the pixel indices on both X- and Y-directions by 0.5. The extraction of the depth values involves more steps.

3.1.1 Using glReadPixels

One option for extracting the depth values is to use the OpenGL function call, `glReadPixels` [Gro08], which returns the pixel data from the frame buffer updated by the first OpenGL rendering pass. When the format parameter is set to be `GL_DEPTH_COMPONENT`, `glReadPixels` returns the depth values from the depth

buffer, which is converted into floating point, and mapped to the range from 0 to 1.

This approach is trivial to implement, but has the following drawback. According to the OpenGL specification, `glReadPixels` extracts the data from frame buffer into the client memory. This means the data is implicitly transferred from the GPU back to the CPU. If we pack the depth values into a buffer object or a `cudaGraphicsResource` (CUDA 3.0), and pass it into a subsequent CUDA kernel, an implicit `cudaMemcpy` routine is invoked so as to bring them back to the GPU memory. The unnecessary memory transfer introduces considerable overhead to the overall performance, so we decided to explore more efficient solutions.

3.1.2 Using Frame Buffer Objects

Another way to extract information from a frame buffer is to create a frame buffer object (FBO) [Ahn08] and bind it into the OpenGL rendering pipeline. In this way, instead of being displayed on the screen, the rendering outcome from OpenGL will be redirected onto the bound frame buffer object, which can be used for other purposes. We employed an FBO in our design to extract the depth values from OpenGL.

A frame buffer object is essentially a container of a number of rendering destinations. There are two types of images that can be attached to FBOs: texture images and render buffer images. Textures or render buffers can be attached onto the depth attachment point of FBOs to accommodate the depth values. However, as far as we have tested, they cannot be accessed by CUDA kernels. We suspect that the functionality of the FBO extension in OpenGL is implementation dependent, and the interoperability between FBOs and CUDA is not fully supported at this point. Therefore, an object attached onto the depth attachment point of an FBO cannot be used as a `cudaGraphicsResource` in CUDA.

However, we managed to use the color attachment points on the FBO. Apparently in this case we have to write a simple shader program to dump the depth values onto the color channels of the frame buffer. According to the GLSL specification [KBR06], the special variable `gl_FragCoord` can be directly used in fragment shader programs to extract the fragment positions including the depth.

Nevertheless, our solution was slightly different. As shown in Figure 3.2, we have a vertex shader program that simply applies the model-view-projection transformation for each vertex, and then the Z- and W-components of the transformed vertices are passed to the fragment shader. In the fragment shader, the perspective divisions are performed, and the depth values are mapped to the desired range (from 0 to 1). At the end of the fragment shader program, the depth values are dumped to the color channel of the FBO and are ready to be used by the subsequent CUDA kernels.

Although the depth information will be redirected to the color attachment of the FBO, a depth attachment is still needed for the FBO. The reason is that it facilitates

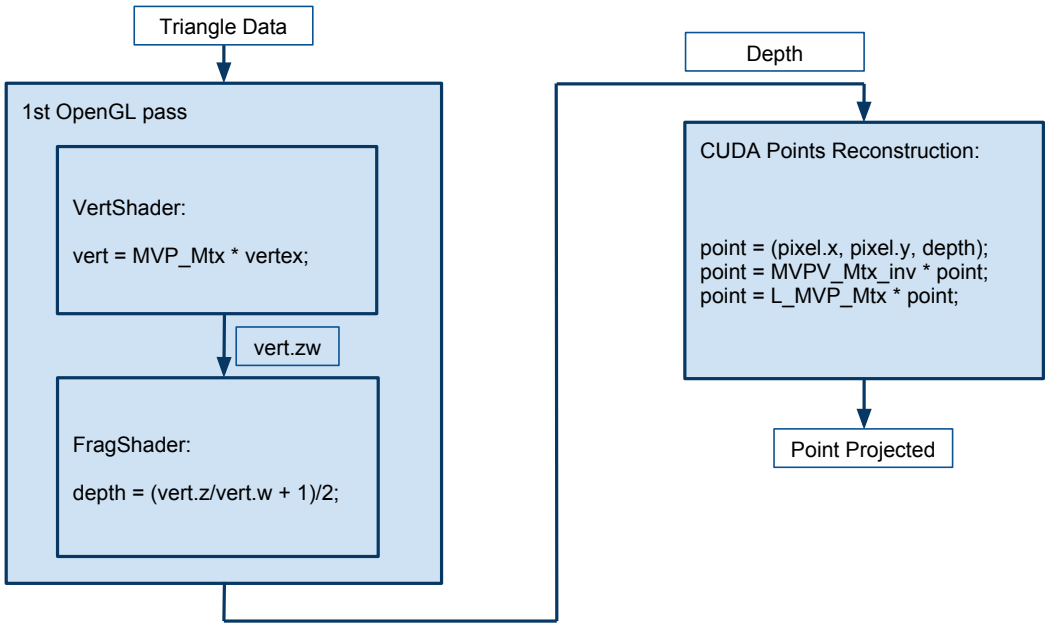


Figure 3.2. 1st OpenGL pass and Point-Reconstruction Kernel

the correct functionality of the OpenGL depth test.

3.2 Sample Points' Reconstruction

The process of the sample points' reconstruction is no more than a series of vertex transformations between three different spaces: the screen space, the world space, and the light space. The CUDA point-reconstruction block in Figure 3.2 illustrates the process. At the end of the kernel, the projected points are stored into the global memory.

3.2.1 Memory Access Consideration

We only generate the depth values from OpenGL, and the sample points are reconstructed in a separated CUDA kernel. It is evident that sample points can be directly reconstructed by the OpenGL shader program and stored in the FBO. The reason we let the CUDA kernel handle this is that not only do we want to minimize the number of times that the point data are accessed, but also we want to reduce the data size of each access.

As what will be discussed in the following sections, the building of the acceleration data structure of the point data is dependent on the exact bounding box of all the sample points. In order to figure out the bounding box, a parallel reduction needs to

be performed across all sample points. If the points are reconstructed in OpenGL, two coordinates (X and Y) per point need to be accessed by the subsequent reduction kernel. In contrast, if OpenGL only generates the depth values, the points can be reconstructed on-the-fly and be fed to the reduction directly. By combining the reconstruction with the reduction in the same kernel, the global memory access in between can be eliminated, and only a depth value per pixel is needed by the kernel.

3.2.2 Practical Details

Here are a few practical tips that we have applied to optimize the point-reconstruction process.

Two matrices are needed for the transformations: the inversion of the view-projection-viewport matrix of the camera space, and the view-projection matrix of the light space. They can be packed in an array and copied into the GPU's constant memory. Constant memory is cached, and it is accessible for all CUDA kernels launched afterward.

Although the homogenization steps are missing in Figure 3.2, they need to be enforced right after each matrix multiplication. The homogenization is, however, only performed on the X- and Y-coordinates, keeping the Z-coordinates in linear space in order to simplify the depth interpolation in the rasterization stage. The instructions are optimized by pre-calculating the reciprocal of the W-component after each matrix multiplication and multiplying each coordinate component with the reciprocal to save costly floating-point division instructions.

3.3 Bounding Box Reduction

After point-reconstruction and transformation, sample points are irregularly distributed on the light-view image plane, as shown in Figure 3.3.

3.3.1 Why Bounding Box?

The idea of the bounding box is simple but of great importance. There are two reasons why it is needed. First, the 2D uniform grid of the sample points needs to be built right inside of the bounding box. The bounding box abstracts the distribution of the irregular sample points, and it encapsulates all the sample points that need to be rasterized in a compact rectangular region. Therefore, the 2D grid of the sample points should be built by dividing up the area of the bounding box uniformly along both X- and Y-axes. When the size and the position of the bounding box change, the 2D grid is self-adaptive to this change. Thus, a more uniformed point distribution is maintained.

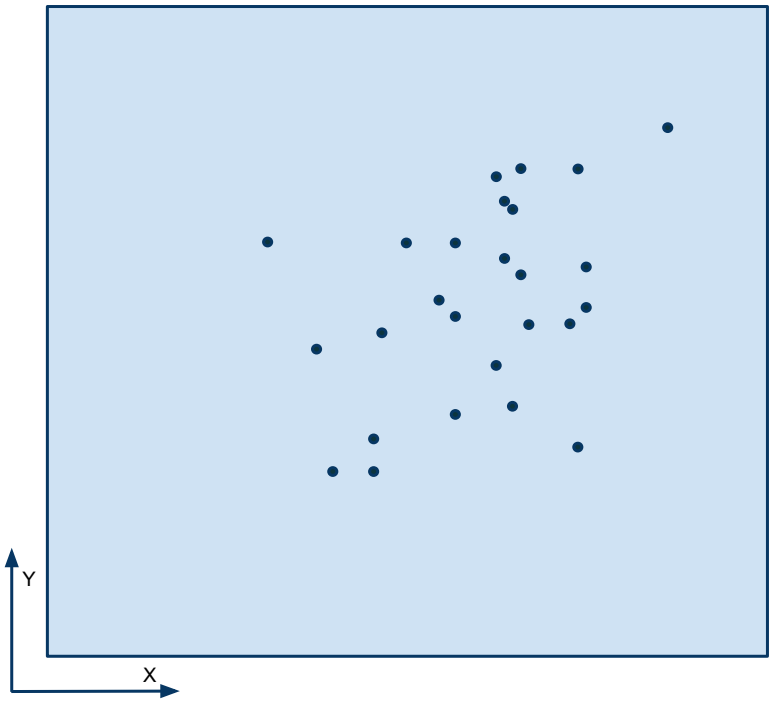


Figure 3.3. Sample Points on Light-view Image Plane

The second reason falls onto the triangle data. Because the triangles located completely outside the bounding box are of no interest for our rasterization, they should be culled away.

As implied in Figure 3.4, the computation of the bounding box is the process of finding the minimum and maximum values of the X- and Y-coordinates among all sample points.

3.3.2 Parallel Reduction

Parallel reduction is a common and powerful GPGPU primitive [OLG+07]. A large stream input is reduced into a smaller stream or possibly a single element stream after reduction. It can be used to compute the sum or maximum value of all the elements in a stream.

As shown in Figure 3.5, parallel reduction is implemented in a tree-based approach on GPUs [Har07]. In the figure, a max reduction is performed to compute the maximum value of the input stream. However, on each level, any associative binary operation can be applied. Since the bounding box of all sample points is our interest, only min and max operations are concerned.

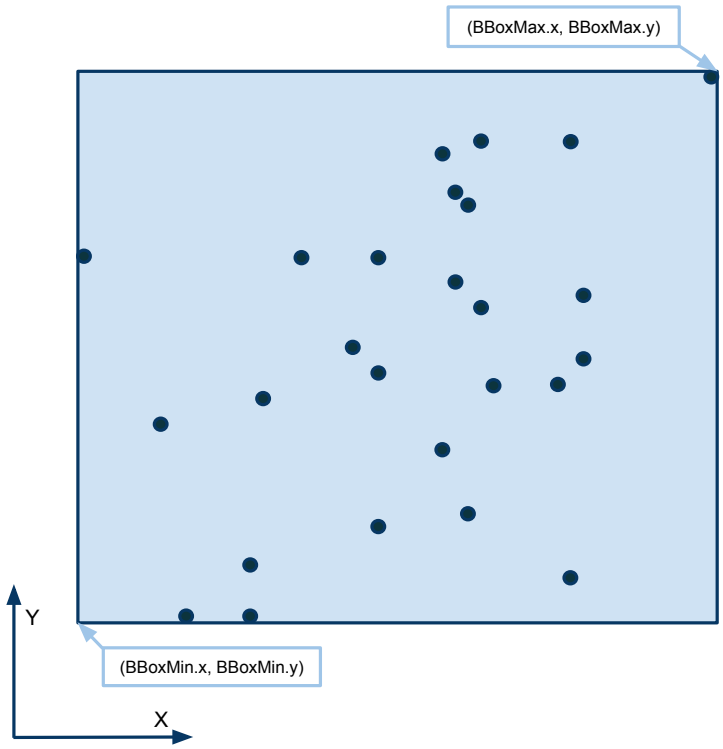


Figure 3.4. Bounding Box of Sample Points

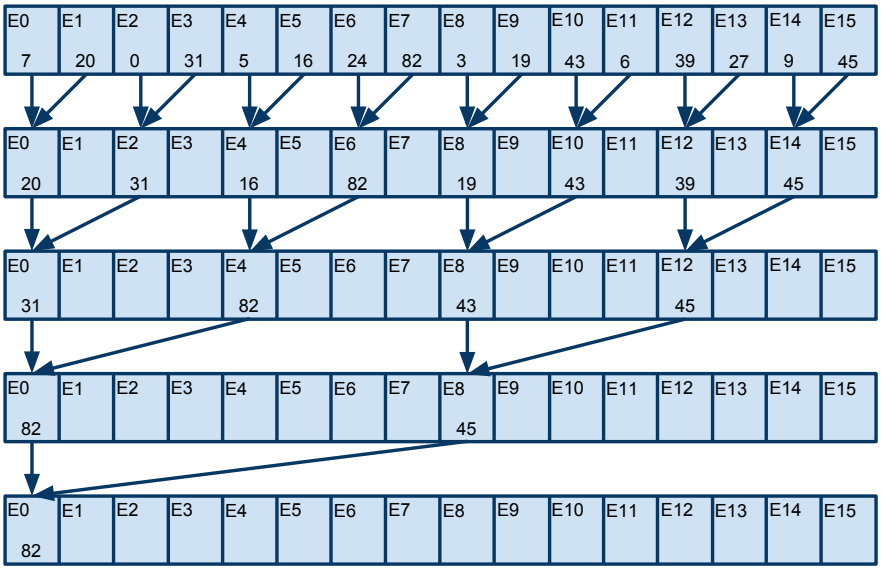


Figure 3.5. Parallel Reduction: Max

3.3.3 Using Existing Libraries

There are a number of existing data-parallel algorithm libraries implemented in CUDA, which can be used in your design to release you from making everything

from scratch. For example, CUDPP [HOS+] and Thrust [HB] have most common GPGPU algorithms and primitives well implemented in CUDA including reduction, prefix scan, and radix sort.

Even though four different min and max values can be computed by separate standard reduction routines that are supported by the libraries, it is not an efficient solution in terms of performance. Point data need to be arranged in the way that it is compatible with the library functions, and compromises have to be made on redundant global memory accesses.

3.3.4 Customized Reduction Kernel

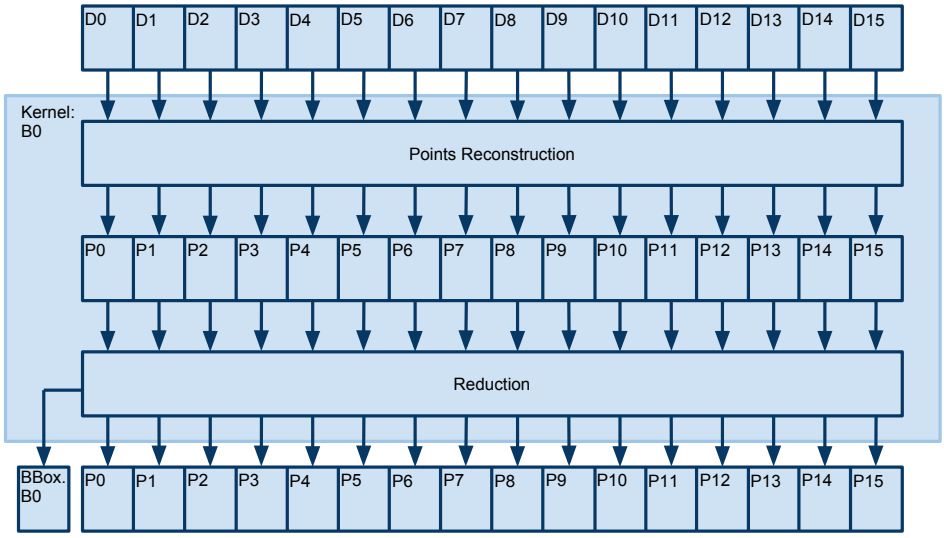


Figure 3.6. Points Reconstruction and Reduction: Block 0

A customized parallel reduction kernel integrated with the point-reconstruction is the optimal solution in this stage. First, the reconstructed points are buffered on the shared memory, and reductions are performed across the buffered points within each block. Four binary operations (min for X, min for Y, max for X, and max for Y) are stacked together at each level. Finally, reconstructed points are stored in the global memory, and each block produces a `float4` bounding box of itself (Figure 3.6).

A second reduction stage is needed to gather all the intermediate bounding box data. As shown in Figure 3.7, the second reduction stage is invoked to compute the exact bounding box across all blocks. At the same time, according to the desired grid dimensions, the cell sizes of the uniform grid on X- and Y-directions are calculated as well.

We followed the guideline discussed by Mark Harris in [Har07] to optimize our reduction kernels. There are a few crucial techniques that we like to point out.

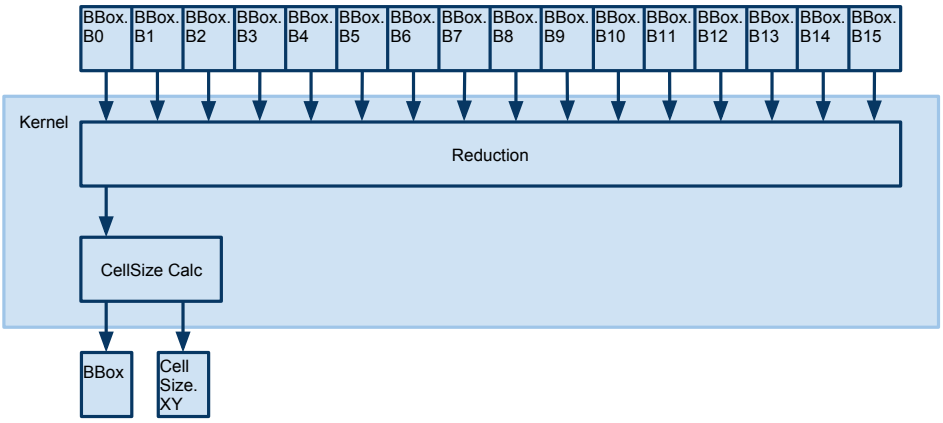


Figure 3.7. Reduction Final Stage

Sequential Addressing

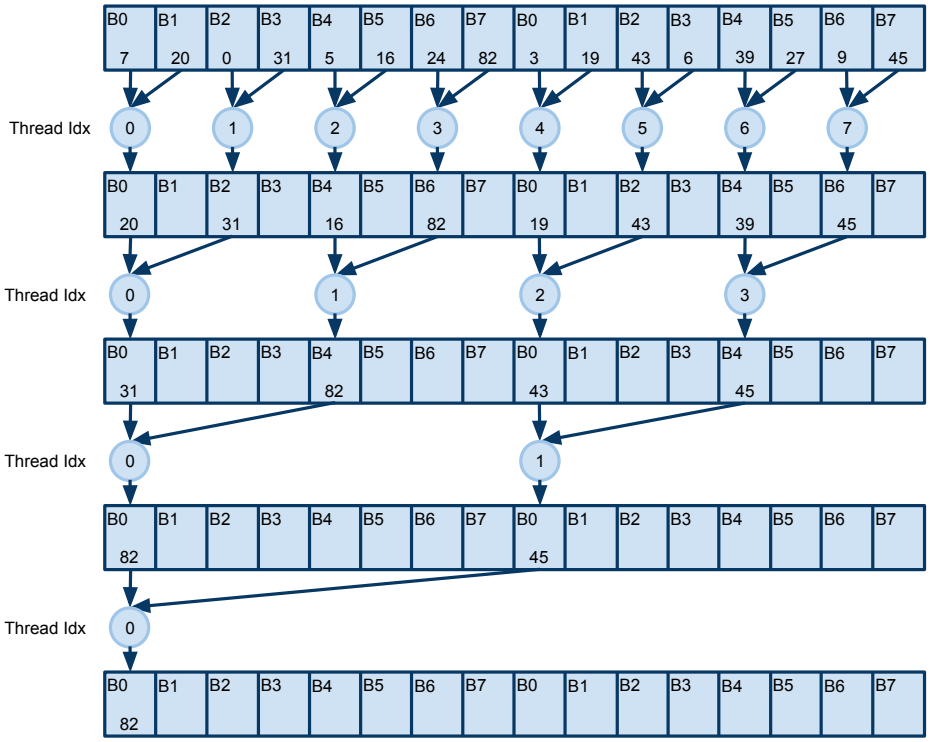


Figure 3.8. Parallel Reduction: Interleaved Addressing

The shared memory residing in a streaming multiprocessor (SM) of a CUDA GPU is divided into 16 banks. Within a half-warp, accesses from different threads to the same shared memory bank will be serialized, which is called shared memory bank conflicts. Let us go back and review the shared memory access pattern in Figure 3.5. For illustration purpose, let us assume that there are only 8 shared memory banks.

Therefore, a hypothetical but reasonable half-warp size would also be eight. As illustrated in Figure 3.8, this interleaved memory addressing pattern causes shared memory bank conflicts on every level of the reduction.

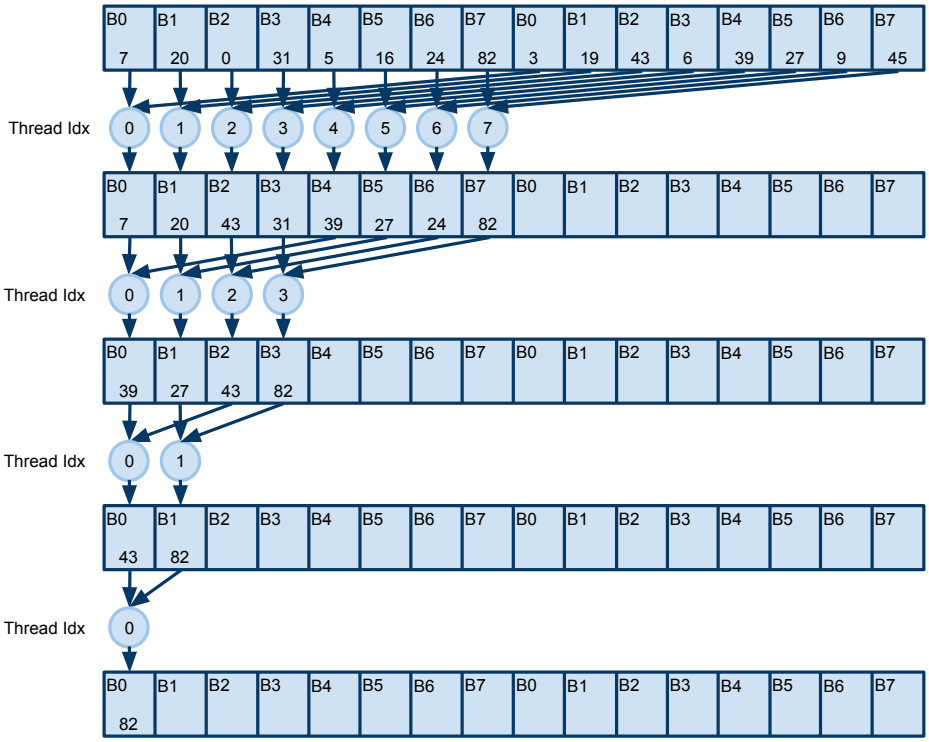


Figure 3.9. Parallel Reduction: Sequential Addressing

One way to address this issue is to enforce sequential addressing (shown in Figure 3.9). By using sequential addressing, each thread within a half-warp only accesses its unique shared memory bank, so bank conflicts are eliminated.

Multiple Loads

One may notice that after the first level reduction, half of the threads are idle. In fact, multiple loads can be issued by each thread at the top level, and binary operations can be performed on the registers before the results are stored in the shared memory. In this way, we slightly increased the granularity of each thread, but the total number of threads is at least halved. The same applies for the second stage reduction kernel.

Loop Unrolling

Loops introduce considerable instruction overhead to CUDA kernels. Loops are initially needed to control the iterations through all reduction levels. But as long

Approach	cudppScan		Customized Reduction	
Item	Routine/Kernel	Time (μs)	Routine/Kernel	Time (μs)
1	1st OpenGL	394.48	1st OpenGL	394.48
2	pointRecon.	111.39	pointRecon. & BBoxReduction	147.04
3	cudppScan×4	348.13	BBoxReduction FinalStage	7.94
Sum		854.00		549.46

Table 3.1. Point-Reconstruction & Reduction Performance Summary

as the data size handled by each threads block is fixed, the number of iterations is known. Therefore, loops can be completely unrolled.

Final Performance

Again, we do not intend to reiterate what Mark Harris has covered in his parallel reduction discussion. As far as we have tested, the points stated above are critical to achieve a better performance in this stage.

After some tweakings, four depth loads together with point reconstructions are performed at the top level; block sizes are set to 256 to have a better occupancy; loops are completely unrolled. Finally, the GPU execution time of the two reduction kernels all-together is 153.89 μs.

3.4 Wrap-up

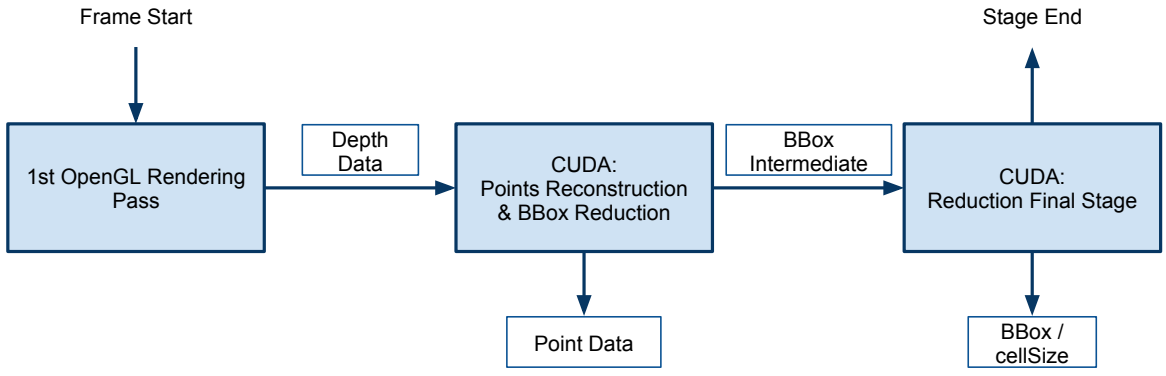


Figure 3.10. Point-Reconstruction & Reduction Stage

The performance of the point-reconstruction and reduction stage is summarized in Table 3.1. We used `cudppScan` to compare with our customized kernel (reduction

is not yet implemented in CUDPP). The OpenGL rendering time is measured by a software timer, and the CUDA kernel execution times are taken from the GPU times measured by the CUDA visual profiler. For simplicity, the resolution is set to 512×512 in our design, meaning there are $250,000$ sample points which need to be processed. Although point-reconstruction is independent from the scene, the speed of the first OpenGL pass does depend on the number of triangles in the scene. For demonstration purpose, the model we tested here contains around $14,000$ triangles.

4 ACCELERATION TECHNIQUES

In this section, we discuss various techniques that we have applied in our design to accelerate the access of the sample point data and the triangle data.

4.1 Sample Points Data Structure

In our design, the rasterizer needs to process a quarter million sample points (the resolution is 512×512). Each of the sample point has three coordinates, X, Y, and Z, in floating point. Not considering any other extra information that we might have to pack into the point data structure, the size of the point data is around 3 MB.

Because sample points are irregularly distributed across the light-view image space, there is no concrete correlation between the index of the point and its spatial location. If no accelerated data structure is applied, for an adequately sized scene of 100 thousand triangles, due to the lack of spatial information each triangle will have to iterate through all the sample points. If we times the data size of the sample points by the number of triangles, there are 3 terabyte point data that we have to access from the global memory. Moreover, between the memory accesses, the computations are heavy. It means all the memory accesses will be dispersed across the entire rasterization process, and memory access latencies will not be hidden very well between iterations. This leads to an even worse performance degradation.

Considering the facts stated above, building an accelerated data structure for the point data is inevitable.

4.1.1 2D Uniform Grid

As we known, rasterization is basically working in two-dimensions, so it is not necessary to adopt any more complicated 3D structure. Because uniform grids are fairly easy to build and to traverse, we chose to build a 2D uniform grid structure for the point data.

Figure 4.1 shows how the grid looks like on the light-view image plane. The bounding box of the sample points is taken and divided uniformly along X- and Y-axes into a grid. Each individual rectangular region, of which the grid consists, is called a cell or a fragment. The cell index is arranged in such a order that it is ascending from the lower left corner to the upper right corner of the grid. The figure shows that the indices of different grid cells are vectors of size two, but in the real implementation they are converted to be one-dimensional and stored as `unsigned int`s.

The points located at different regions of the bounding box are binned to their own

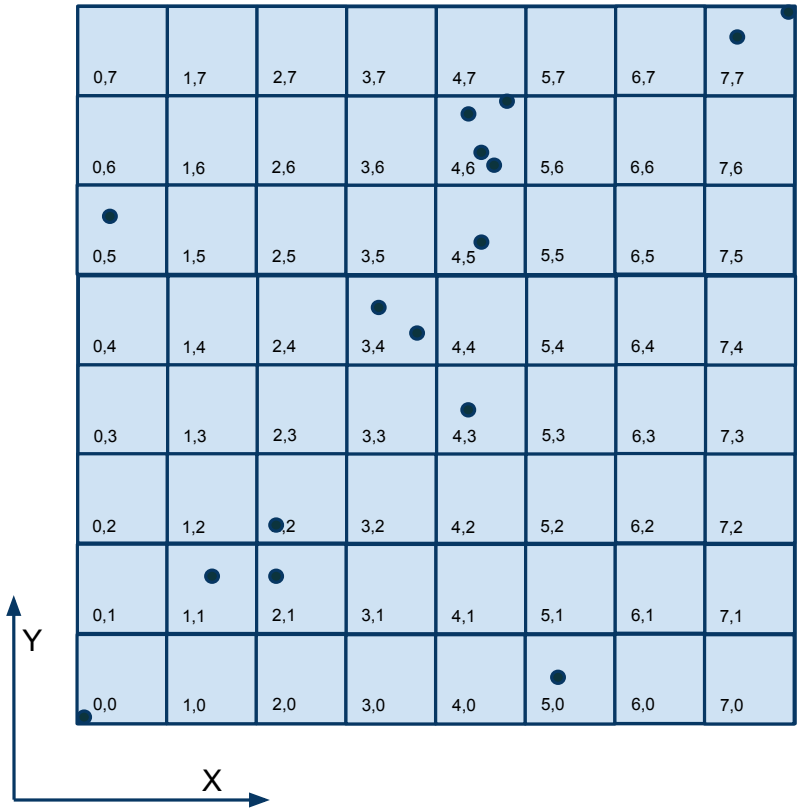


Figure 4.1. 2D Uniform Grid

bucket cells. When a specific part of the grid is desired, only the points belonging to the desired cells are required. Figure 4.2 shows the basic idea of how the point data access can be accelerated. When a triangle (shown in green) needs to be rasterized, we compute the bounding box of the triangle (shown in yellow), and we pick a number of grid cells that are covered by the triangle bounding box (marked in red); the points binned to the red cells are required, and the rest are skipped.

After triangles are transformed and projected onto the light-view image plane, for triangle, $\triangle \mathbf{P}^0\mathbf{P}^1\mathbf{P}^2$, its bounding box can be calculated as:

$$\begin{aligned} BBox_{min} &= (min(p_x^0,\ p_x^1,\ p_x^2),\ min(p_y^0,\ p_y^1,\ p_y^2)) \\ BBox_{max} &= (max(p_x^0,\ p_x^1,\ p_x^2),\ max(p_y^0,\ p_y^1,\ p_y^2)). \end{aligned} \tag{4.1}$$

$BBox_{min}$ and $BBox_{max}$ denote the lower left corner and the upper right corner of the bounding box. The grid cells covered by the triangle bounding box are:

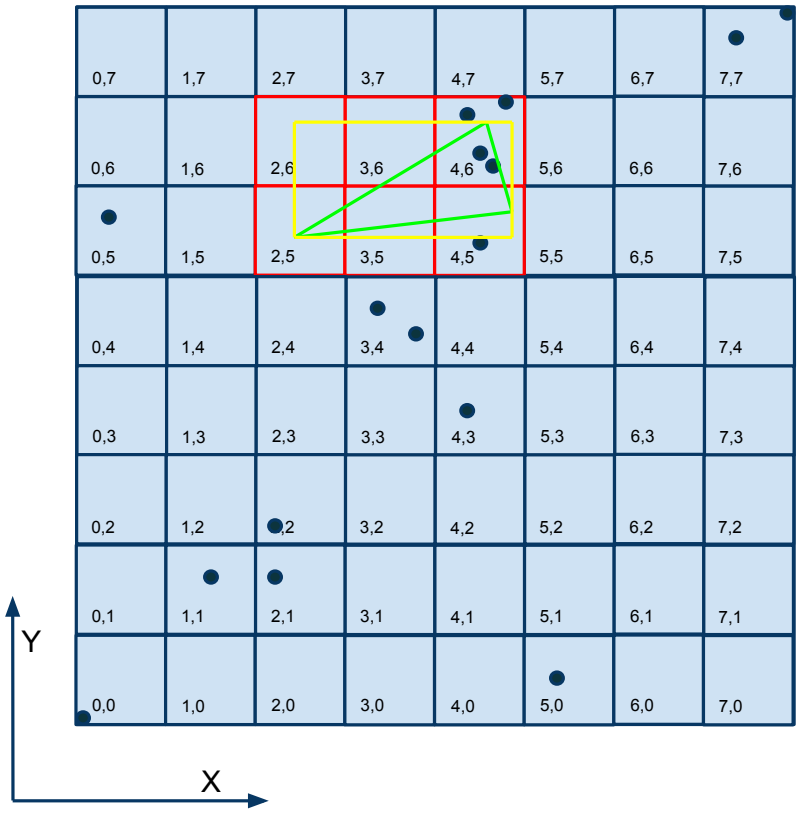


Figure 4.2. Triangle Bounding Box Traversal

$$\begin{aligned} cell_x &= \left[floor\left(\frac{BBox_{min}}{cellSize_x}\right), floor\left(\frac{BBox_{max}}{cellSize_x}\right)\right] \\ cell_y &= \left[floor\left(\frac{BBox_{min}}{cellSize_y}\right), floor\left(\frac{BBox_{max}}{cellSize_y}\right)\right]. \end{aligned} \tag{4.2}$$

$cell_x$ and $cell_y$ are the range of the covered cells on X- and Y-directions respectively. If we put them together, the complete list of the covered cells denoted as $cell_{covered}$ can be expressed as:

$$cell_{covered} = [(cell_x[0], cell_y[0]), (cell_x[1], cell_y[0])...(cell_x[last], cell_y[last])]. \tag{4.3}$$

Similar triangle bounding box traversal approaches have been discussed in [LHLW10] and [FLB+09], but their works only focused on rasterizations of regular sample points.

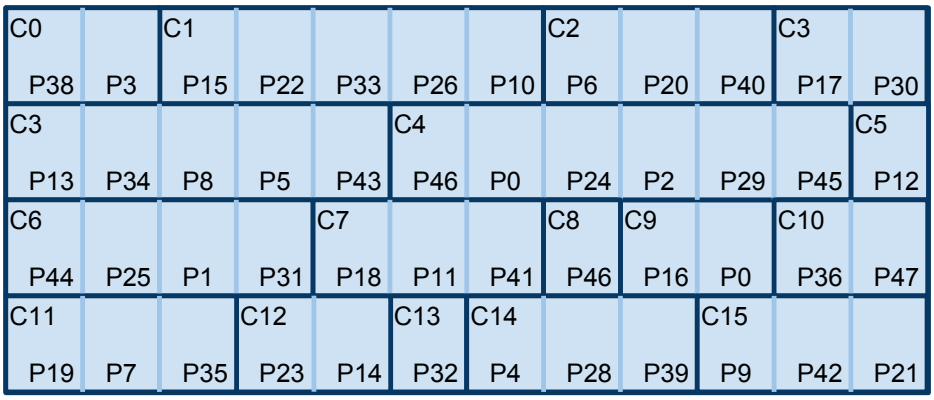


Figure 4.3. Point Data in 2D Uniform Grid

4.1.2 Point Data Structure in Detail

The 2D uniform grid, i.e., the accelerated data structure that we build for the point data, was introduced in the previous section. Here we discuss some lower level details regarding how point data are stored in the GPU memory.

Figure 4.3 visualizes the grid data structure at the point level. $P38$ stands for point of index 38, and $C0$ stands for cell of index 0. The data structure starts with the points binned to the first cell of the grid, and they are followed by the points belonging to the second cell. It goes on like this, and ends with the points of the last cell of the grid. Due to the irregularity of the point distribution, the cell sizes tend to be different. Despite the fact that the allocated memory region is conceptually divided into different cell sections, the data can still be arranged in such a compact way that it does not require extra physical memory space. Please note that the different rows in Figure 4.3 only represent linear memory continuations of the previous rows. The whole data structure is actually stored in a continuous linear memory chunk. For instance, as shown in the figure, $P39$ and $P13$ both belong to $C3$, and thus, $P13$ is sitting right next to $P30$ in the memory address space.

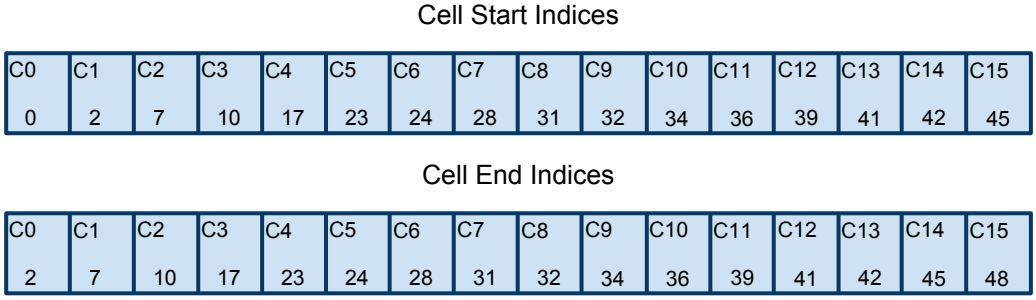


Figure 4.4. Cell Start and End Indices

In order to access the points of a specific cell or fragment, two separate arrays are

needed to store the start and end indices of different grid cells. Figure 4.4 illustrates the two indices arrays, which is associated with the previous point data structure figure. We call them cell start indices array (CSI array) and cell end indices array (CEI array). Each element of the CSI array is the index of the first point binned to its corresponding cell. The CEI array contains the exclusive ending positions of different cells, meaning the end index of cell number 1 is essentially the start index of cell number 2. The reason why we made the CEI array exclusive is because it is easier to generate in this way. Once the CSI array is obtained, the CEI array is given at the same time except the last element that can be easily fixed.

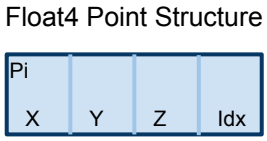


Figure 4.5. Float4 Point Structure

Let us dig into each individual point. Each point has three coordinates in floating point: X, Y, and Z. One potential arrangement of the point data is to store them in a `float3`, but the drawback is that the built in vector type `float3` in CUDA is treated as a structure of array. This means when a thread reads one `float3`, three memory transactions will be invoked one after the other to access one `float` at a time from three different arrays. In this way, although coalescing is maintained, only 64 Byte memory transactions are issued (4 Bytes per thread), and memory bandwidth is not utilized in a good way.

Alternatively, we chose to pad in a forth component and store the point data as `float4`s. First of all, `float4` is treated as array of structure in CUDA. When one `float4` is being accessed by a thread, two 128 Byte coalescing memory transactions are issued. Comparing to `float3`, `float4` leads to a better bandwidth utilization and less memory transactions.

Beside its performance consideration, it turns out that the forth component is actually needed in our rasterization stage. After the binning process, the point data are reordered. Due to the fact that the shadow stencil buffer, the output of our rasterizer, follows the same indexing scheme as the pixels in the screen space, we need to find a way to get track of the original indices of the reordered points. If we pad the point data with a forth component containing its original screen space index, the problem is resolved. The index component is stored as `float` and will be cast back to `unsigned int` whenever a shadow stencil buffer update is required. Figure 4.5 exemplifies the `float4` point data structure.

4.2 Point Binning

We preset two methods to perform the binning of the irregular sample points.

4.2.1 Radix Sort Based Binning

Hash Calculation and Data Reordering

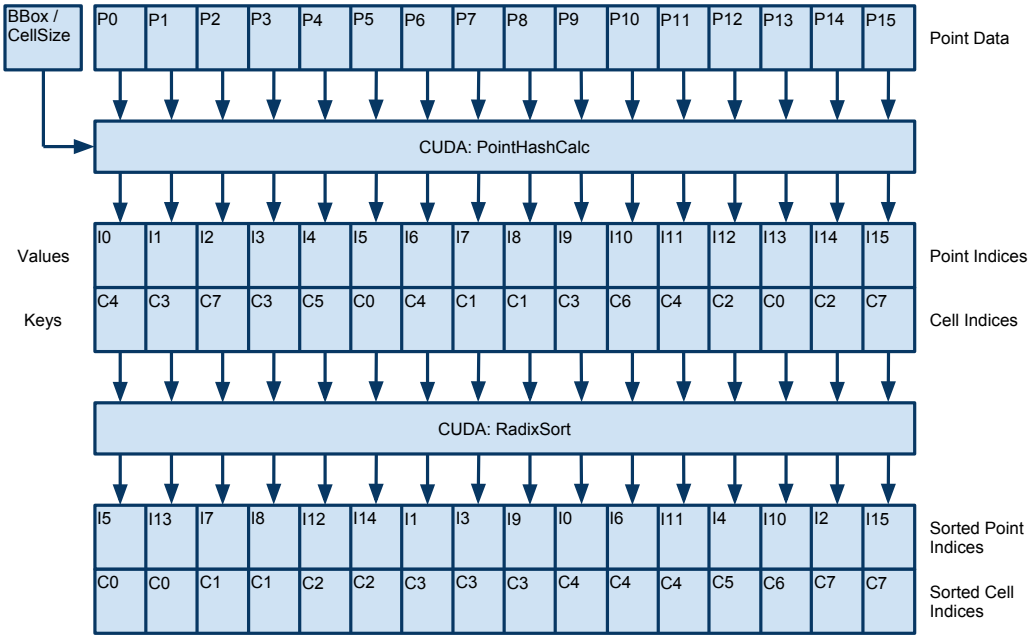


Figure 4.6. Radix Sort Binning I

The first one is called radix sort binning which, as one can tell from its name, is built based on the GPU radix sort algorithm described in [SHG09]. Figure 4.6 explains the first part of the process referred to as hash calculation and data reordering. During hash calculation and data reordering, the index of the cell, to which each point belong, is calculated, and point data are reordered accordingly.

In kernel `PointHashCalc` (stands for point hash value calculation), each thread fetches one point and computes the cell index it belongs to according to the sample points bounding box and the sizes of the grid cells generated from the precedent stage. The hash value is essentially the linear cell index stored as `unsigned int`. Overflow is avoided by clamping it to the boundary values. The calculated cell indices array is treated as the keys in radix sort. An original point indices array is also generated and then is used as the values in radix sort.

After sorting, the point indices are shuffled by the radix sort kernels in the way that it is aligned with the sorted cell index array. As shown in Figure 4.7, a point data reordering kernel is invoked to rearrange the actually point data into the desired form described in the previous section. The point data are gathered from various places specified by the sorted point indices, and then stored back to the global memory in a coalesced way.

The reason that we employed the point index array as an intermediate data structure

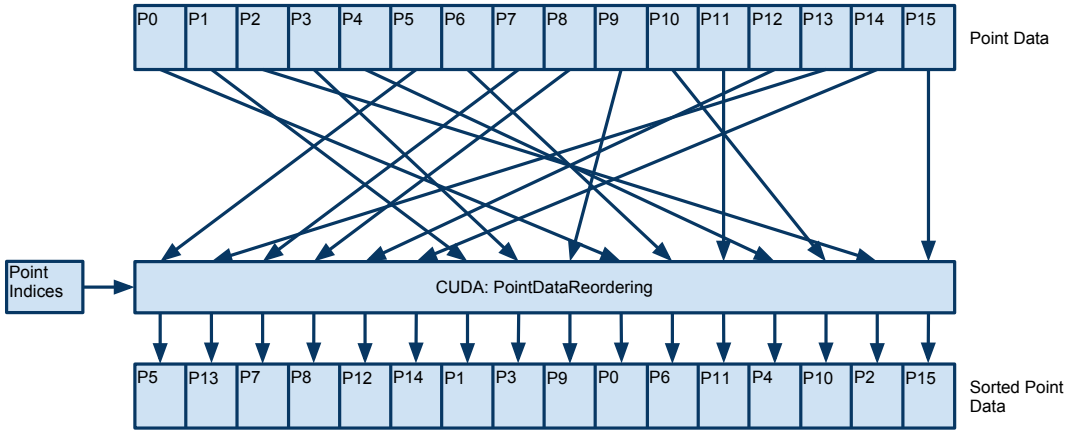


Figure 4.7. Radix Sort Binning II

and then applied an extra data reordering step is because the radix sort implementation we used only supports 32-bit values. However, comparing with the execution time of the radix sort, the reordering kernel is still a small fraction.

Cell Indices Generation

As stated in the previous section, cell start and end indices are needed for the grid traversal. That motivates the second part of the radix sort binning process, where CSI and CEI arrays are generated.

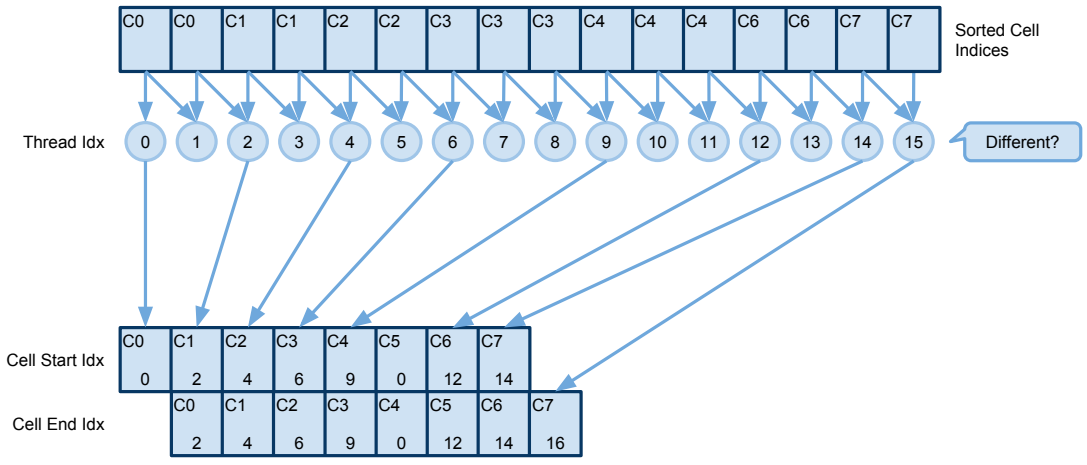


Figure 4.8. Cell Start and End Indices Generation

As shown in Figure 4.8, each thread loads two elements from the sorted cell indices array except thread number 0: one from its in-place index and the other one from its lower next index. Two cell indices are then compared. If the two are different, the

corresponding cell start and end indices are updated with the index of the thread. Special cares are taken of for the first and last cells.

Prefix Fill

One may notice that if one cell happens to be empty, meaning no point has been binned to this cell (cell number 5 in Figure 4.8), the way we described above fail to generate the cell start and end indices for it. In order to fix this, a customized prefix scan routine is added to fill up the indices of empty cells. We call it prefix fill.

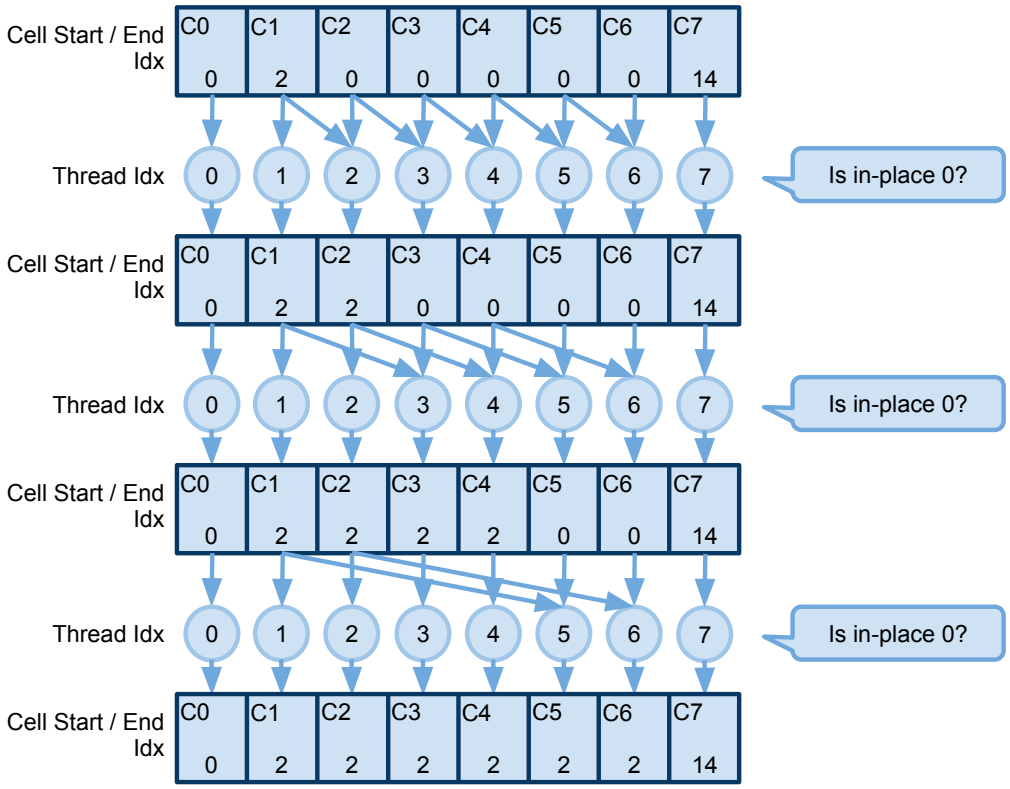


Figure 4.9. Cell Start and End Indices Filling I

Prefix fill replaces the binary add operation of a prefix sum with the following code:

1: **if** $cellStartIdx[threadIdx] = 0$ **then**
2: $\quad cellStartIdx[threadIdx] \leftarrow cellStartIdx[threadIdx - 1]$
3: **end if**

When the in-place element is 0 or another initial value to mark it as empty, the in-place element is updated with the value of the left operant of the binary operation.

Apart from this, prefix fill follows exactly the same pattern as in prefix scan. Therefore, it also consists of three steps or kernels: segmental scan within each block,

recursive scan across all blocks, and vector addition to compensate the preliminary segmental scan results. A more figurative interpretation is given in Figure 4.9.

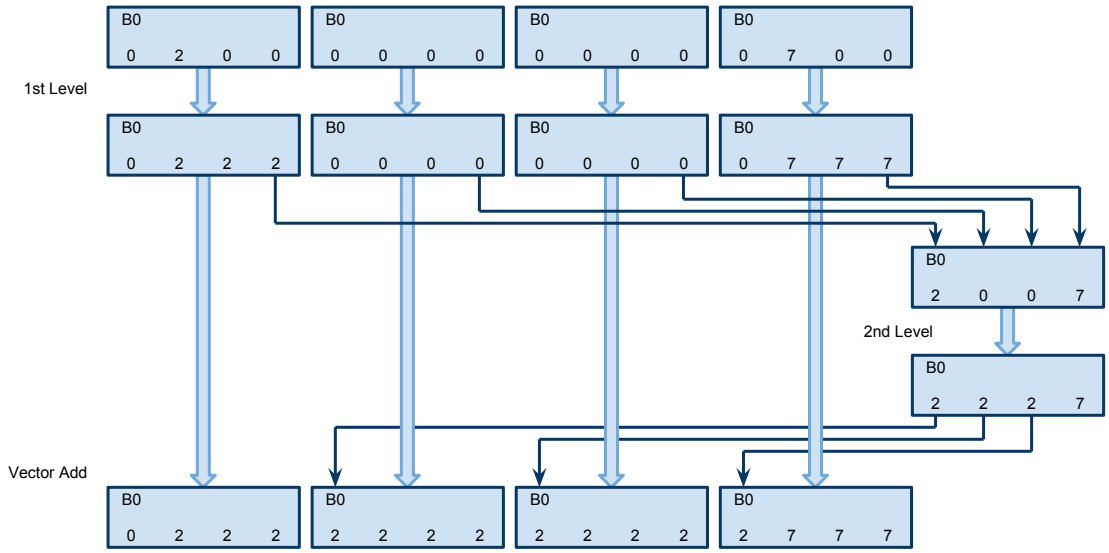


Figure 4.10. Cell Start and End Indices Filling II

As it shows in the figure, a second level scan is needed to propagate the partial scan outcomes to the global scale. At last, a vector addition step is applied to carry out the actual compensations. At all levels, a fill-if-empty-otherwise-not rule is enforced.

We did not urge on applying more advanced optimization techniques as described in [SHG08] but keeping prefix fill in a more or less naive form. This is because prefix fill works on a relatively small data set (grid width $\times$ grid height $= 128 \times 64$ in our design), and it is not the bottleneck of our radix sort binning. One thing worth to mention is that CSI and CEI arrays should be initially reset each frame to avoid violations from the previous frame.

Wrap-up

Here, we give a rundown on our radix sort based point binning method. Figure 4.11 illustrates the radix sort binning at kernel level. CUDPP radix sort is abstracted as a single routine. After sorting, we merged the point data reordering and the preliminary CSI and CEI generation into one kernel, following which prefix fill routine is invoked to fix the untouched indices.

The performances are summarized in Table 4.1. Kernel execution times are taken from the GPU times measured by the CUDA visual profiler. The resolution is set to 512×512, so the data size of the sample points is $250,000$. Although point distribution does affect the speed of the radix sort binning, in general the performance of this method is dictated by the data size of the sample points.

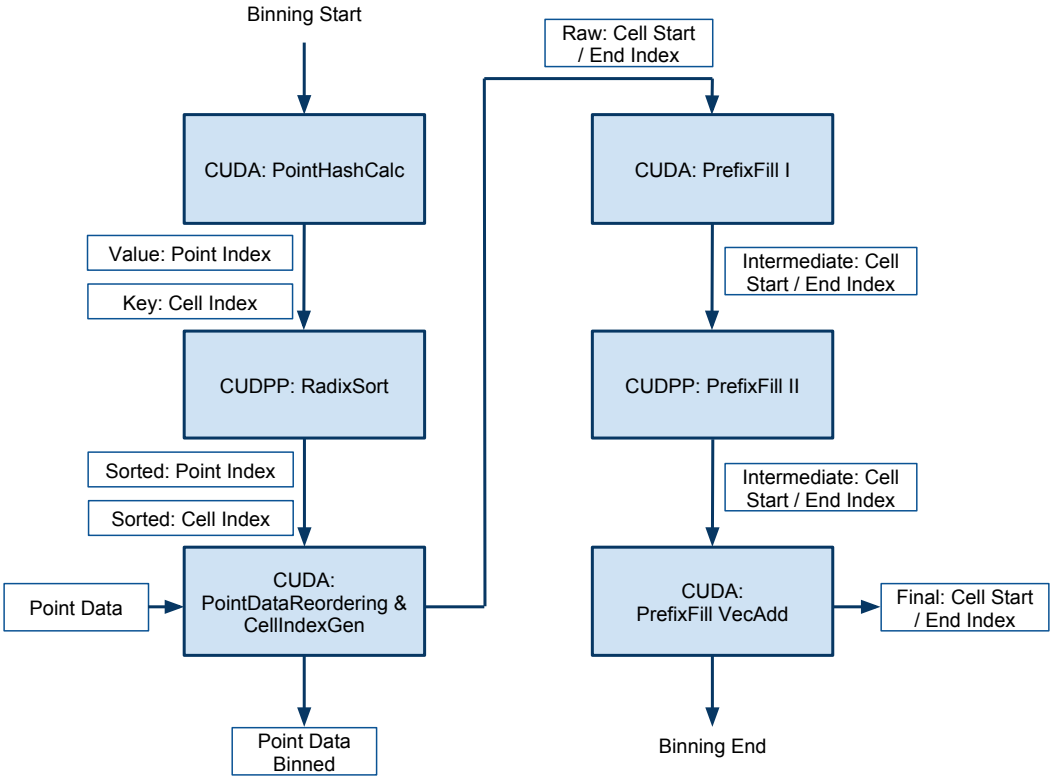


Figure 4.11. Radix Sort Point Binning

Approach	RadixSort Binning	
Item	Routine/Kernel	Time (μs)
1	PointHashCalc	244.35
2	CUDPP:RadixSort	1266.43
3	PointDataReordering & CellIndexGen	121.63
4	PrefixFill_I	12.74
5	PrefixFill_II	9.12
6	PrefixFillVecAdd	8.80
Sum		1663.07

Table 4.1. Radix Sort Point Binning Performance Summary

4.2.2 Atomic Operation Based Binning

The second point binning approach that we are going to discuss is based on the atomic operations supported by CUDA. We start with a naive approach and describe how it can be improved to be even faster than the radix sort binning.

A Naive Approach

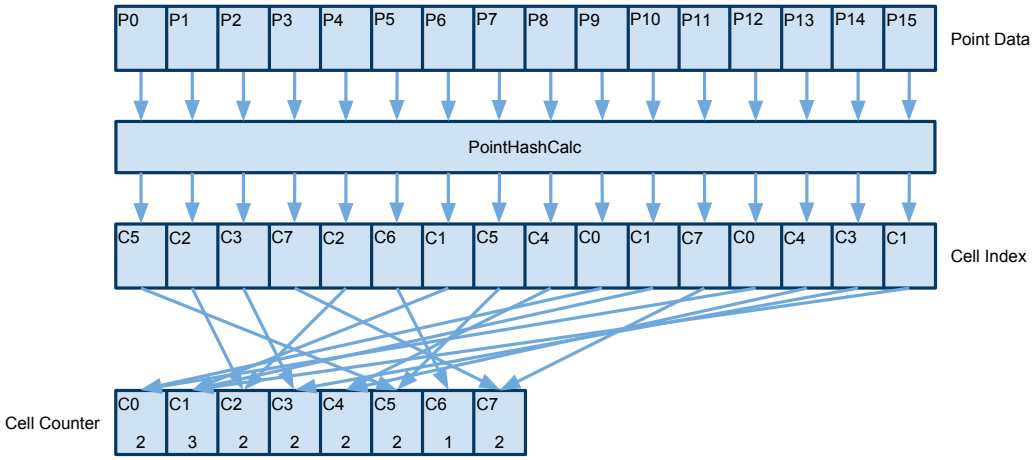


Figure 4.12. Atomic Binning: Naive I

A naive atomic operation binning is fairly easy to implement which employs atomic operations only on the global memory.

First, as we did in the radix sort approach, each thread fetches its own point data and calculates the hash value, which is essentially the cell index of this point. Then we enter a stage called counting stage (Figure 4.12), where we count for all grid cells how many points belong to each one of them. A cell counter array is maintained in the global memory to get track of the number of the points that have been binned to each cell.

During the counting stage, an `atomicAdd` is performed by each thread on its corresponding cell counter to increase its value by one, implying that there is a new point has been binned to this specific cell. When multiple threads are accessing the same cell counter, synchronizations need to be enforced, and this is why atomic operations are involved. After all the atomic operations are complete, the counting stage finishes with a cell counter array containing the number of points binned to each cell.

At this point, point data are left untouched. Therefore, we need to make a transition from the counter array to the information regarding how the point data can be reallocated piece by piece. It turns out that the generations of the cell start and end indices array will guide our way.

So the transition can be made in two steps. First, the cell start indices array can be generated by applying an exclusive prefix sum on the cell counter array that we just obtained. And the cell start indices array gives the information of the starting positions for each cell in the targeted grid structure shown in Figure 4.3. The rest of the question is how the points should be arranged within each cell section, where the order of the points does not matter, but each one has to have its unique position.

Instead of starting another round of chaotic atomic operations, we saved the returned value from each atomic operation during the counter stage in a separate array, and that value is indeed the unique internal offset for each point within the cell it belongs to (Figure 4.13). The final offset per point array is produced by offsetting the cell internal offset of each point with its corresponding cell start index. By now, we are able to reorder the point data to its desired place.

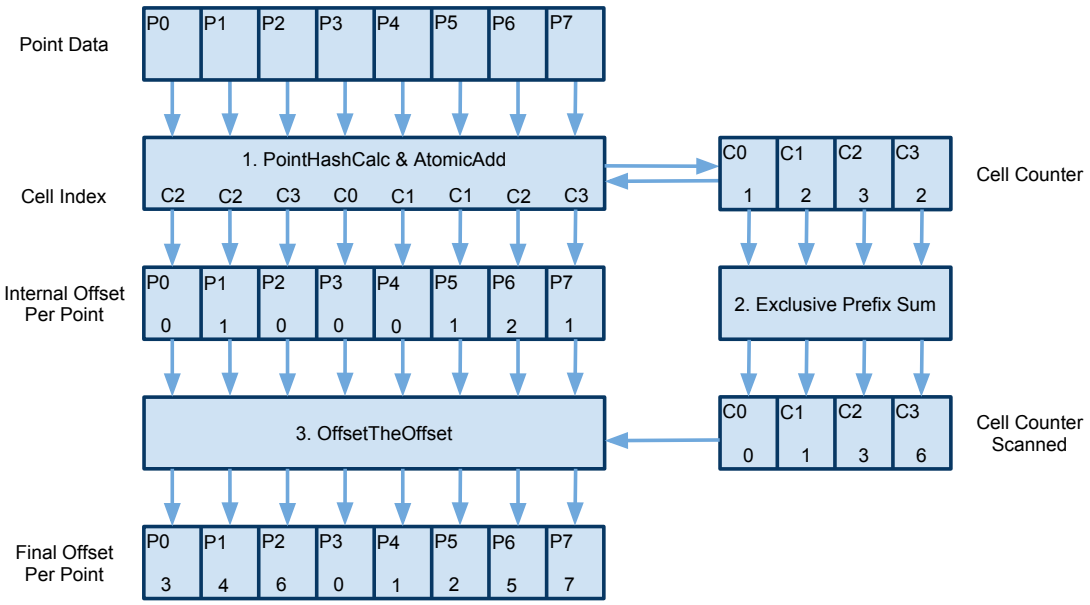


Figure 4.13. Atomic Binning: Naive II

One optimization can be made for the internal offset per point array (shown in Figure 4.13). The cell internal offset for each point and the cell index of the point can both be encoded into 16 bits, and two values can be packed into one 32-bit `unsigned int`. This is safe because our grid dimension is 128×64 giving 8191 as the maximum value of cell indices. In the worst point distribution case we experienced, the maximum number of points per cell never goes above $10,000$. Therefore, 16-bit (from 0 to 65535, if `unsigned int`) is enough to accommodate both values. By doing this, when the `OffsetTheOffset` kernel (step 3 in Figure 4.13) is invoked after the prefix sum, both data are accessed in a more efficient way, and unnecessary computations are saved.

Point data reordering is the final step of the atomic operation binning, which is simply done by taking the final offset for each point and reallocate the point data in a scattered way. After that, point data are rearranged as desired, and CSI and CEI arrays are ready to be processed by the rasterizer.

The performance of the naive approach is congested by the large number of atomic operations performed on the global memory. For a resolution of 512×512, a quarter million atomic operations have to be performed by the CUDA kernel, not mentioning how many serializations have occurred on the same global memory addresses. The

way that the atomic operations are performed results in a huge performance drop.

Atomic Operation on Shared Memory

Because the latency of accessing the shared memory is much smaller than that of accessing the global memory in CUDA, one improvement for the atomic operation binning can be performing atomic operations first on the shared memory, and then merge the partial results back to the global memory through a second round of atomic operations. CUDA devices with computation capability 1.2 or higher support atomic operations on both shared and global memory.

One challenge is that the size of the shared memory residing in a streaming multiprocessor (SM) is limited (16KB). Our grid dimension is 128×64 that produces $8,192$ 4-Byte `unsigned int`s consuming $8\text{K} \times 4\text{B} = 32\text{KB}$ memory. Therefore, it is not possible to buffer the whole copy of the cell counter array into one SM, and they need to be divided into groups — 4 groups in our case (8KB per SM).

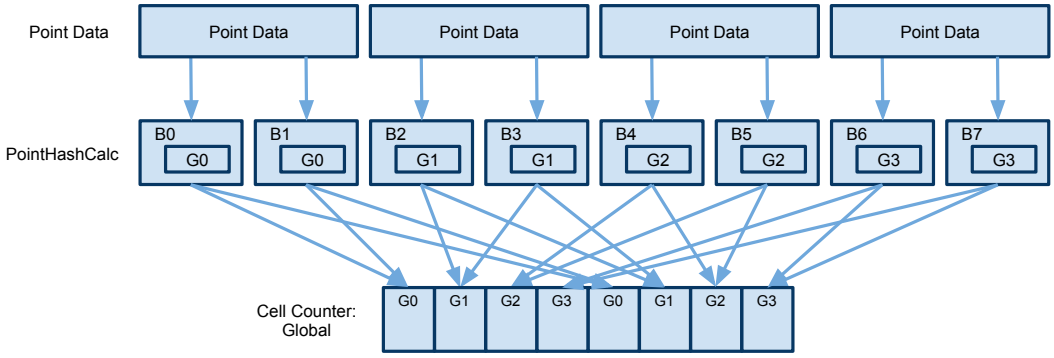


Figure 4.14. Atomic Counting on Shared Memory

Figure 4.14 illustrates the modified counting stage of the atomic operation binning. All the launched thread blocks are divided into 4 groups. Each group is responsible for a specific subset of the cell counter array, and each block within the same group buffers the same copy of the subset in its shared memory. Cell counters are assigned to different groups in a interleaved way so as to achieve a better load balancing.

As shown in the figure, each group will have to go through the entire point data, and `atomicAdd` is only performed on the shared memory when a point falls into the cells assigned to this group. In this way, each group is able to collect a part of the point distribution information that it is responsible for, and eventually all different parts are merged from different groups back to the global memory through atomic operations to form the complete cell counter array.

In the improved counting stage, memory access serializations on the global memory are replaced with warp serializations on the shared memory. Therefore, latencies are reduced. What can also be observed from Figure 4.14 is that the total number

of atomic operations that need to be performed on the global memory becomes the product of the number of blocks per group and the size of the cell counter array. This means the number of the global memory atomic operations required is not dependent the screen resolution anymore.

As stated above, the smaller the group size (the number of thread blocks per group), the less atomic operations need to be performed on the global memory. On the other hand, all sample points need to be processed by each group, meaning a quarter million points are parallelized among the thread blocks belonging to the same group. Therefore, the bigger the group size, the less points need to be processed by each block. There is a trade-off between a bigger and a smaller group size. After a series of tests with different parameter combinations, we found that 4 groups with 32 blocks per group gives the best performance at this stage.

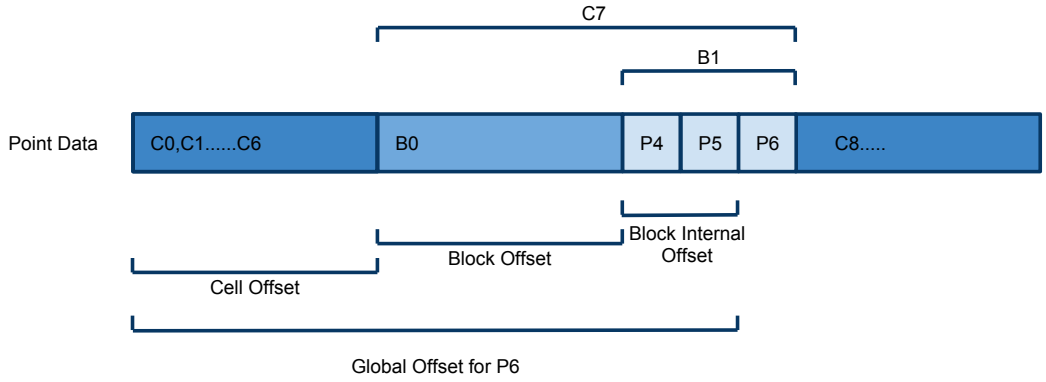


Figure 4.15. Sample Point Offset

Although the counting stage is resolved in a better way, it takes more effort for us to calculate the right offset for each point, which is needed for the reordering. We are going to explain this through a concrete example. As shown in Figure 4.15, point number 6 ($P6$) was handled by block number 1 ($B1$), regardless which group it belongs to. $P6$ has been binned to cell number 7 ($C7$). The desired reallocation position for $P6$ is shown in the figure. The corresponding global offset of $P6$ consists of three parts: cell offset of $C7$, block offset of $B1$ within $C7$, and a block internal point offset specifically for $P6$. Thus, for each sample point p^i, its global offset o^i_{global} can be expressed as:

$$o^i_{global} = o^i_{cell} + o^i_{block} + o^i_{blockInternal}. \tag{4.4}$$

In order to obtain all three offset components, Figure 4.16 illustrates the five steps involved to calculate the right offset for each point, and they are explained as follow:

1. `HashCalc`: Hash calculation is performed per point to identify the cell each point belongs to.

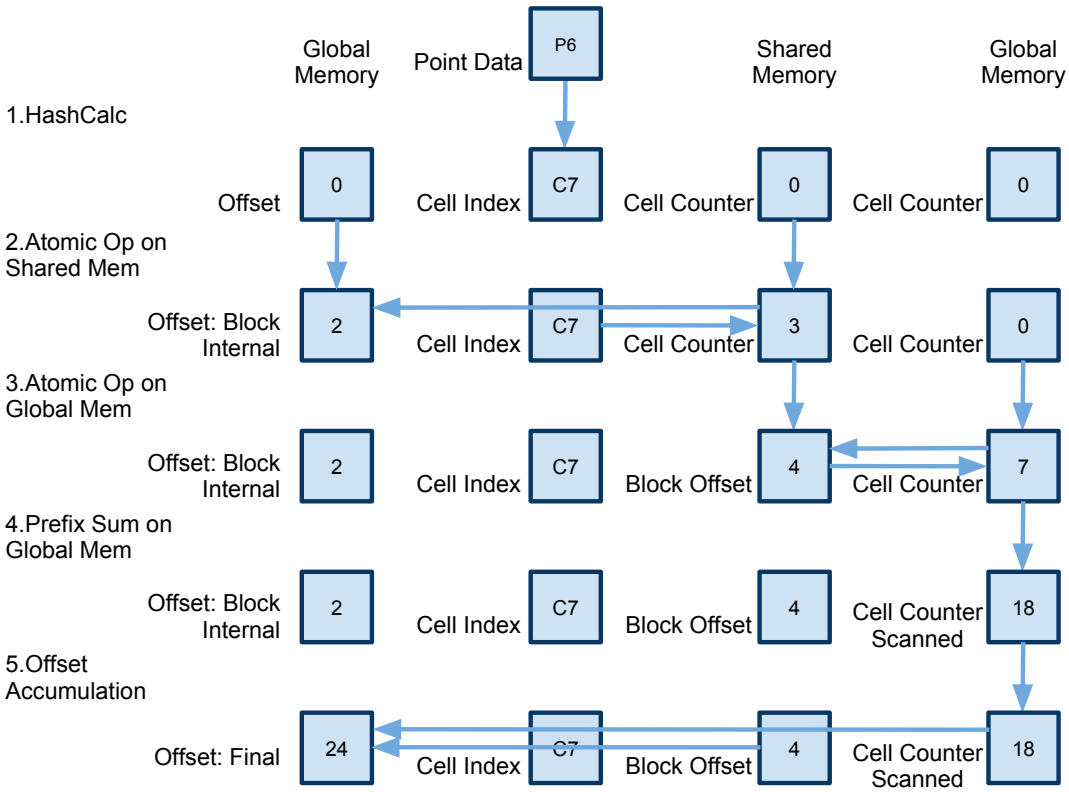


Figure 4.16. Sample Point Offset Calculation

2. `Atomic op on shared mem`: The cell counter increment is first performed on the shared memory. We store the returned value of the atomic operation in a separate array and label it as block internal offset, which is the block internal offset for the point within the cell it belongs to. Then the hash value (cell index of the point) and the block internal offset are encoded into a 32-bit `unsigned int` and stored in the global memory.

3. `Atomic op on global mem`: The value of the shared memory counter is merged onto the global memory counter, and the old value of the global memory counter is swapped back to the same shared memory location. The value that we got back from the global memory is essentially the block offset within the cell represented by the counter. Finally, the block offset array is dumped to the global memory.

4. `Prefix sum on global mem`: A prefix sum is performed on the cell counter array formed on the global memory to generate the cell start indices array.

5. `Offset accumulation`: The block internal offset array generated in step 2 is fetched and decoded. Then the block internal offset is accumulated with the block offset (from step 3) and the corresponding cell start index (from step 4). The sum gives the correct global offset for each point.

The rest of the binning can be directly inherited from the naive approach, so there is no need to reiterate.

Wrap-up

The performances of the atomic operation based binning is summarized in Table 4.2. Again, kernel execution times are taken from the GPU times measured by the CUDA visual profiler. Although load balancing techniques are applied, the atomic operation based binning is indeed view dependent. The performance varies from frame to frame, but the fluctuation is still tolerable. The execution times listed in the table is tested with a relatively small scene which contains around 14K triangles. Due to the fact that smaller scenes tend to have worse point distributions, the performance figure listed in the table below is more of a conservative number. In most cases, a better performance can be expected.

Approach	Atomic Op. Binning	
Item	Routine/Kernel	Time (μs)
1	HashCalc & AtomicCounting	503.01
2	CUDPP:Scan	29.66
3	PointDataReordering	339.97
Sum		968.67

Table 4.2. Atomic Operation Point Binning Performance Summary

4.3 Triangle Data Structure

4.3.1 Acceleration Data Structure Consideration

Thanks to the revolution of GPGPU, the programmability of modern GPUs has been fundamentally ameliorated. In the last few years, we have seen novel algorithms proposed for constructing geometry acceleration data structures on graphics hardware. K. Zhou introduced the first real-time kd-tree construction algorithm on GPUs [ZHWG08]; C. Lauterbach et al. presented a fast BVH construction method on GPUs [LGS+09]; A real-time uniform grids construction algorithm was described by J. Kalojanov [KS09].

Those methods work quite well with more advanced rendering techniques like ray tracing, but are they really suitable for our rasterizer? It turns out that they are still too expensive to use. One performance figure can be observed from [LGS+09]. For the Sibenik model (82K triangles), it takes around 30 ms to construct the BVH on GPUs.

Even though irregular shadow mapping can be seen as a trimmed-down version of ray tracing, it only generates shadows, a small part of the overall rendering outcome. We are not inclined to follow the same path as many did on GPU ray tracing, because we are only concerned about shadow generations. Our algorithm should be able to be integrated with the standard rasterization based rendering pipeline, so it can be used as a replacement for traditional shadow mappings under certain circumstances. If our design has the same or similar complexity as GPU ray tracing, people would not even consider using it. They can either go for ray tracing which gives more realistic rendering effects including shadows, or stick with the existing real-time shadow generation algorithms where they can still make a satisfactory balance between the quality and the speed. Therefore, our irregular rasterizer should be kept as simple as possible and be affordable for even slightly older hardware generations. Towards the end, we decided not to use acceleration data structures at triangle level.

4.3.2 Triangle Data Structure in Detail

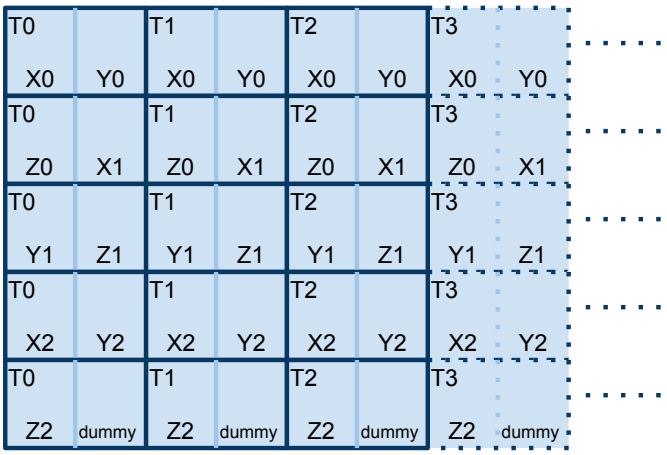


Figure 4.17. Triangle Data Structure of Array

In order to achieve a better throughput when accessing the triangle data from the global memory, the triangle array is arranged as a `float2` structure of array (shown in Figure 4.17). `T0` stands for triangle number 0 and `X0` is the X-coordinate of the first vertex of `T0`. Loading a `float2` per thread within a warp is associated with a 128-Byte coalescing memory transaction in CUDA, which is the biggest data chunk that can be handled by one memory transaction. Triangle data are stored in five `float2` arrays in an interleaved way. Since there are only nine `floats` of a triangle, we pad a dummy floating point value together with the Z-coordinate of the third vertex in the last `float2` array.

4.4 Triangle Culling and Compaction

4.4.1 Why Triangle Culling and Compaction?

Triangle culling is important not only because unnecessary computations can be excluded from the rasterization stage, but also it contributes to load balancing. After we cull away the back-facing triangles and the triangles that are located outside of the sample points bounding box, the valid triangles left are most likely filling up the whole bounding box area, meaning the less the triangles left, the bigger they are on the light-view image plane. As what we will discuss in chapter 5, bigger triangles lead to performance drops in the rasterization stage. In order to maintain a satisfactory performance, more parallelism needs to be explored for each triangle. So if triangle data can be filtered, compacted, and then passed onto the rasterization stage, the rasterizer can be dynamically configured to deploy a larger number of threads working on the same triangle, and thousands of unwanted threads are prevented from being launched and scheduled. Triangle culling and compaction result in a significant performance enhancement in the rasterization stage.

4.4.2 Triangle Culling

Our irregular rasterizer is highly customized to generate shadow information. Therefore, as long as it produces the correct shadow, irrelevant triangle data can be filtered before being streamed into the rasterizer. If objects in the scene are modeled as enclosed objects, from the light point of view only front-facing triangles are needed to determine whether the object is a shadow occluder or not. So back-facing triangles can be culled away. By doing this, we could roughly halve the number of triangles that need to be processed during rasterization.

Back-face culling is done by evaluating the signed area of the triangle after the edge equation setup. If the signed area of the candidate triangle is smaller than zero, it will be culled away.

The irregular rasterization is only happening within the region of the sample points' bounding box. Thus, triangles that are located outside of the bounding box can be ignored and culled as well.

The triangle culling is done by evaluating each triangle against the culling rules described above, and the evaluation results are written on the validation flag array (Figure 4.18), where valid triangles are marked as 1s, and invalid ones are marked as 0s.

A further investigation on this direction is that only the occluders are actually required for shadow generations. By only sending the triangle data of the potential occluders to the rasterizer, the shadows on the receiver and even the self-shadows

on the occluders themselves can be maintained correctly. That gives the chance for us to further reduce the size of triangle data that need to be processed each frame, and it does not introduce potential artifacts into the rendering outcome. What we will also realize in chapter 7 is that it actually causes artifacts if we rasterize the triangle data of the receivers.

4.4.3 Triangle Data Compaction

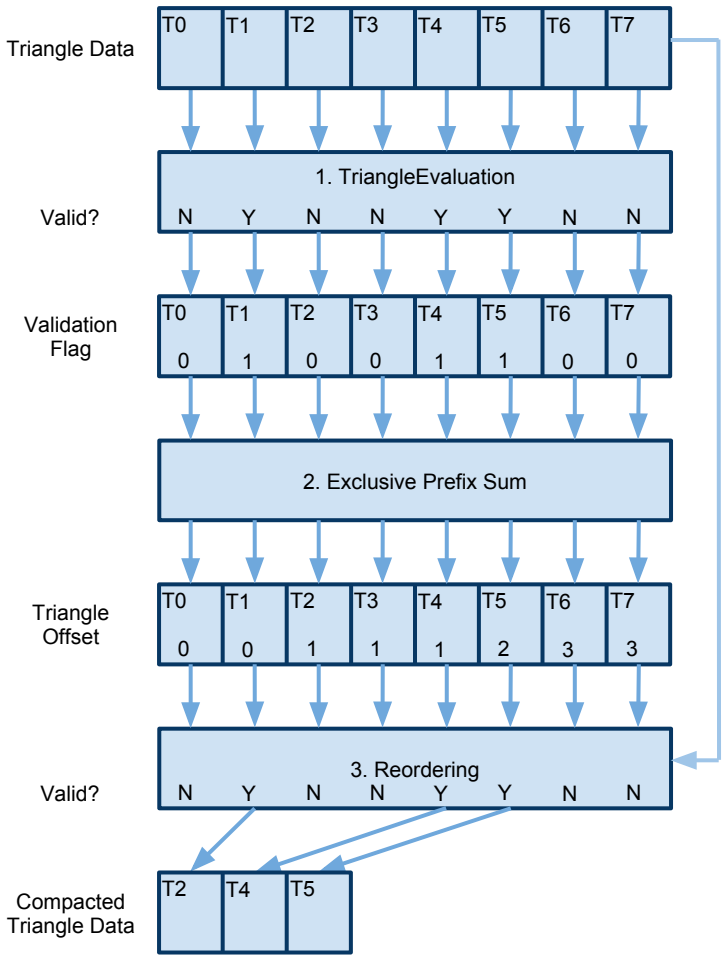


Figure 4.18. Triangle Culling and Compaction

The compaction process is driven by a prefix sum routine. We employed the CUDPP scan library function in this stage, which was implemented based on the parallel prefix sum algorithm described by S. Sengupta [SHG08].

The triangle compaction starts with applying an exclusive prefix sum on the validation flag array generated by the culling step (Figure 4.18). After the prefix sum, the validation flag array becomes the reallocation offset list for the triangles. Then the reordering kernel is invoked to fetch all the triangle data and evaluate their

illegibilities again. Data reallocations are only performed on the valid triangles to the positions indicated by the offset list. After the three steps shown in Figure 4.18, triangle data are filtered and compacted as desired.

4.4.4 Wrap-up

Figure 4.18 also summarizes the triangle culling and compaction stage at kernel level. Apparently, the performance of this stage is dependent on the size of the triangle data and the camera-view angle. The performance figure shown in Table 4.3 was a testing result with a model containing 14K triangles. Although the size of the tested model is small, the overhead introduced by the triangle culling and compaction stage is minor.

Approach	Atomic Op. Binning	
Item	Routine/Kernel	Time (μs)
1	TrisCulling	19.14
2	CUDPP:Scan	31.30
3	TrisReordering	11.58
Sum		62.02

Table 4.3. Triangle Culling and Compaction Performance Summary

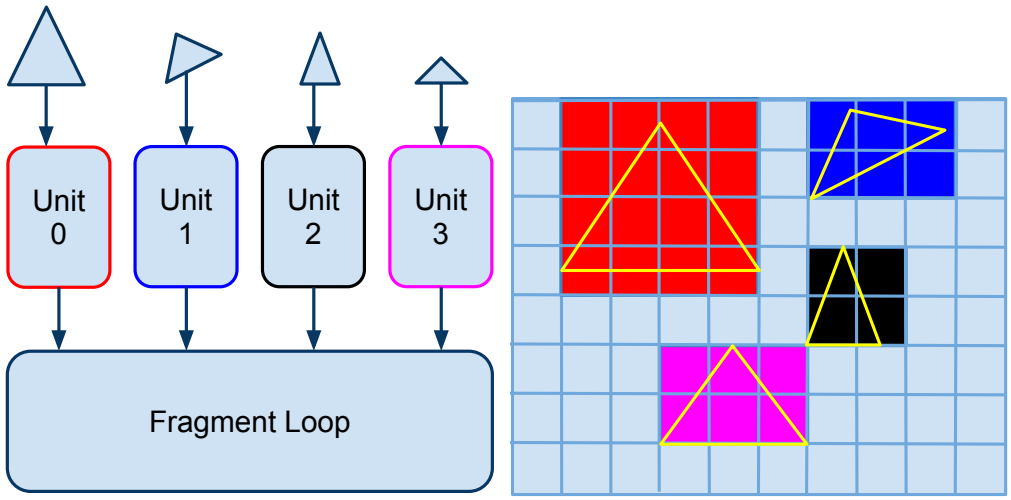
5 TRIANGLE RASTERIZATION

In this chapter, two different triangle rasterization algorithms are going to be described. In CUDA, we always talk about threads, half-warps, warps, or blocks, but this report will stick to something more abstract, which is called a 'unit'. A unit is a group of threads that works together. It could be of any size as long as the block dimension limitation is not violated.

Basically we divide the thread block into a number of units that consists of a few warps or just a few threads. The size of the unit (the number of threads per unit) and the number of units per block are called unit configurations. The rasterization computation of a single triangle is going to be parallelized among a specific unit. CUDA kernels could then be launched with different unit configurations, so that it is possible to investigate the best unit configurations in terms of the speed to rasterize triangles.

Both algorithms employ this methodology, but the difference is in how they traverse over the triangles and the fragments of the sample point grid. Later in this report, these two algorithms are referred as `Kernel1` for the first algorithm and `Kernel2` for the second one.

5.1 Kernel 1



(a) One Unit Per Triangle (b) BBox Traversal Per Unit

Figure 5.1. `Kernel1`

`Kernel1` is built on the basis of one unit per triangle (Figure 5.1a). Each unit fetches one unique triangle from the triangle stream, set up the edge equations, and computes the bounding box of the triangle. The bounding box serves as an acceleration

technique to reduce the number of points that need to be processed. Figure 5.1b illustrates the fragment regions covered by the four triangles being handled by four different units.

5.1.1 Fragment Loop

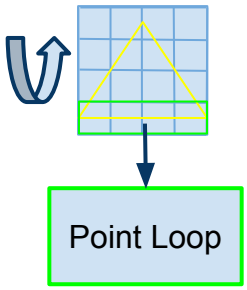


Figure 5.2. `Kernel1` Fragment Loop

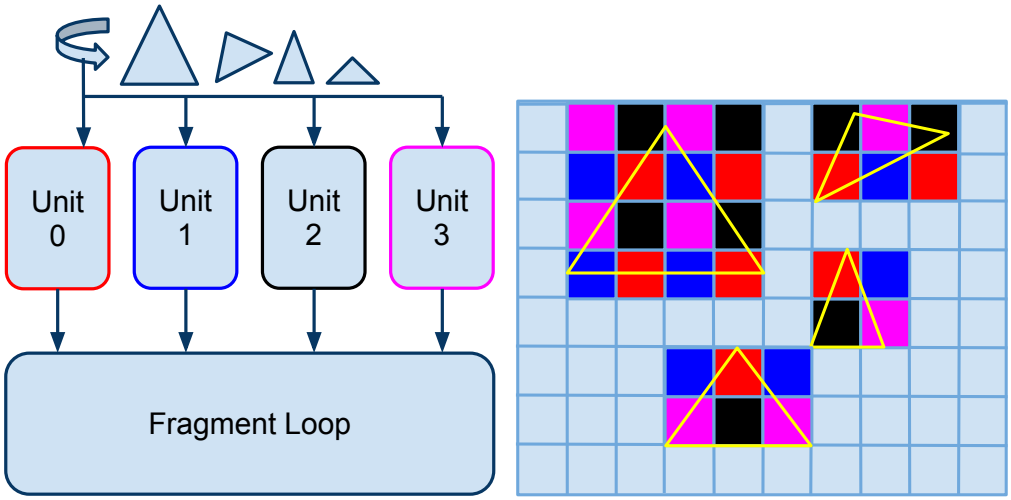
After the fragment region covered by the triangle is computed according to the bounding box of the triangle, each unit in `kernel1` will then go through the region one fragment after another in a nested loop first along the X-axis and then the Y-axis. This loop is called a fragment loop.

Within the fragment loop, the sample points belonging to a specific fragment/cell are accessed according to the associated cell start and end indices generated in the binning stage. The points are parallelized across the entire unit, meaning each thread will fetch its own point and rasterize the point against the same triangle. Due to the irregularity of the point distribution, sometimes not all the threads within the same unit are utilized. If the number of the points binned to the same fragment is larger than the size of the unit, they will be serialized within the iteration of the fragment loop.

Along the X-axis, the sample points belonging to two adjacent fragments are located contiguously in the global memory, so they do not need to be processed by two separate iterations. From another perspective, along the X-axis, the end index of any cell is the start index of the next one. Therefore, the whole fragment line along the X-axis can be accessed by only referencing the start index of the first cell and the end index of the last cell. In other words, each fragment line along the X-axis can be treated as a big fragment (Figure 5.2).

In `kernel1`, the fragment loop is only performed along the Y-axis, and each fragment line long the X-axis is parallelized across the unit. In this way, the references to the CSI and CEI arrays are minimized, and each unit is better utilized during each iteration. The detail of the rasterization follows the same mathematics described in chapter 1.

5.2 Kernel 2



(a) One Unit Per Fragment Set (b) BBox Traversal Per Unit

Figure 5.3. `Kernel2`

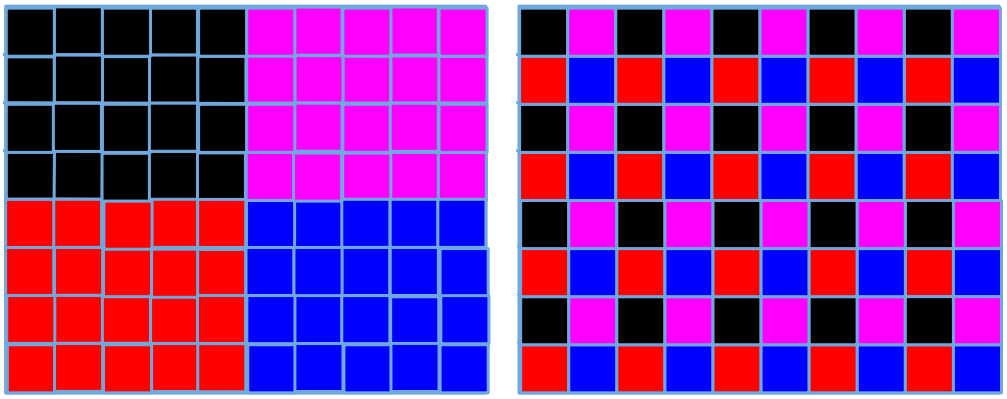
`Kernel2` follows a slightly different idea. `Kernel1` tends to explore the parallelization of the triangle stream, while `Kernel2` attempts to parallelize the fragment executions for each triangle. In `kernel2`, all sample point fragments are divided into groups, and each fragment group is assigned to a specific unit. Each unit will loop over all triangles and only process the fragments that it is responsible for (Figure 5.3a).

As shown in Figure 5.3b, fragments are divided into different groups, and each fragment is marked with the color representing its group identity. Each unit iterates through the entire triangle stream in a loop. When rasterizing one triangle, the triangle bounding box and the fragment region covered by the bounding box is computed the same way as in `kernel1`, but instead of looping over the entire region, each unit only accesses the fragments of its own. In `kernel2`, the rasterization process of each triangle is spread among different units.

In order to avoid the overhead of calculating the edge equations and bounding box multiple times for the same triangle, the triangle data are preprocessed in the culling and compaction stage where related triangle information are calculated beforehand.

5.2.1 Load Balancing Scheme

Figure 5.4 illustrates how two different schemes can be used to spread out the fragments among different units. The hierarchical scheme serves as a counterexample where fragments located close to each other are grouped together. In this way, unbalanced triangle distribution would cause unbalanced workloads between different units, which violates the design intent of `kernel2`. The interleaved scheme realizes what we are trying to achieve in the design of `kernel2`. The fragments covered by



(a) Hierarchical (b) Interleaved

Figure 5.4. Fragment Distribution Schemes

each triangle are averagely distributed among the units, regardless of the size and the position of the triangle. One should be aware that although the fragments are distributed in a balanced way, the point distribution is still inherently unbalanced.

5.2.2 Fragment Loop

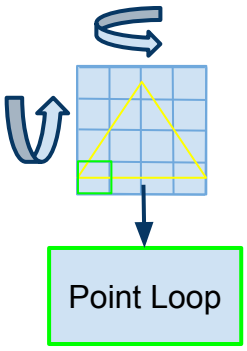
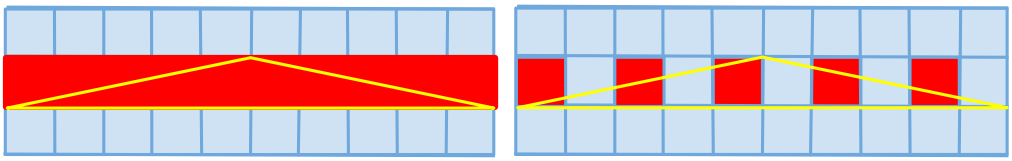


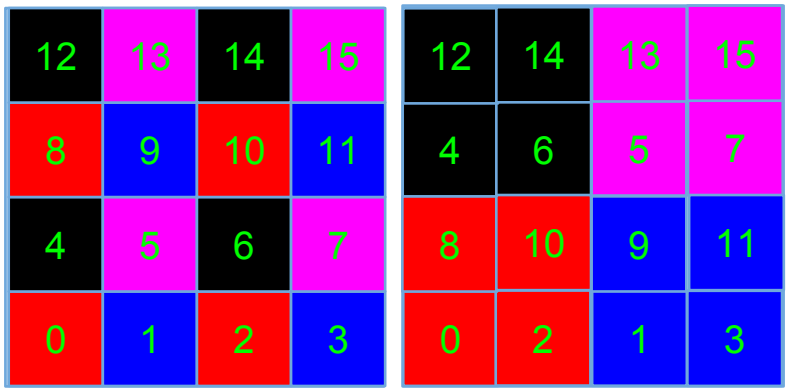
Figure 5.5. Kernel 2 Fragment Loop

In `kernel2`, the fragment loop is more of a brute-force approach. The desired fragments are not located next to each other in the global memory anymore. Therefore, each unit will have to access one fragment at a time along both the X- and Y-axes (Figure 5.5).



(a) Kernel 1 (b) Kernel 2

Figure 5.6. Fragment Fetch Per Line



(a) Original (b) Shuffled

Figure 5.7. Better Fragments Arrangment

5.2.3 Future Implementation

As we known, `kernel1` is congested by the biggest triangle. The bigger the triangle, the more fragments it covers, and the longer the fragment loop. `Kernel2` provides a static solution to average out the workload of each triangle across the entire chip, but the overheads are impossible to neglect. Here we discuss some potential optimizations that can be applied to `kernel2` so as to improve its performance.

Both kernels are constructed with fragment loops. The fragments/cells of the sample point grid are heavily referenced during rasterization. The way that the sample point fragments are traversed differentiates the performances between different rasterization kernels. Comparing to `kernel2`, the fragment traversal is more efficient in `kernel1`: less references to the CSI and CEI arrays, bigger batch size when fetching the points, and smaller loop overhead (Figure 5.6).

However, as shown in Figure 5.7, the point binning stage can be modified in such way that it facilitates the same fragment access pattern for `kernel2` as what is available for `kernel1`. Instead of reallocating the point data according to its spatial location, the sample points belonging to the fragment group are put right next to each other preferably along the X-axis. This modification would accelerate the fragment loop inside `kernel2`.

In addition, the triangle data can be binned to the same 2D uniform grid that we built for the sample points. By doing this, the triangle loop overhead in `kernel2` is to some extent alleviated.

5.3 Point Loop

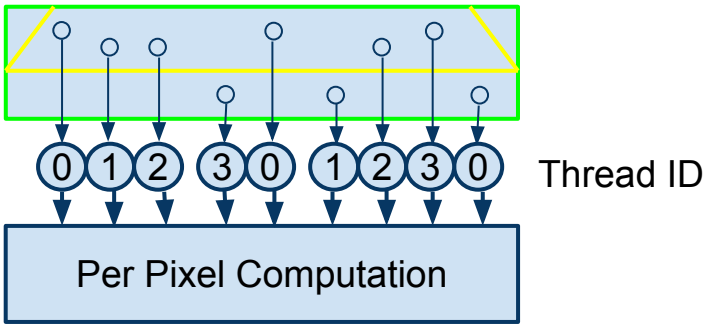


Figure 5.8. Sample Point Loop

Within each iteration of the fragment loop, a point loop is performed. Because each unit consists of a number of threads, each thread participates in fetching the points in the fragment line and performing per pixel computations on them. Figure 5.8 illustrates the computation of one fragment line where the unit length is four. One observation to be made is the repetition of the `Thread ID`. If the number of points located inside the fragment line is bigger than the unit size, all threads need to iterate over this point set.

5.4 Per Pixel Computation

At the lowest level of the loop hierarchy of the rasterization kernel, per pixel computations are performed by each thread. Each per pixel computation carries out the actually rasterization computation of one sample point against one triangle. As described in chapter 1, first the visibility test is performed to check if the point is inside the triangle. If so, the depth value for the triangle is then interpolated and compared to the depth of the point. And if this point is in shadow, the thread writes 0 to the shadow stencil buffer at the location pointed by the original index component of the point data.

The shadow stencil buffer, which is actually a texture, will then be used in the last rendering pass. In the last rendering pass, directional light components are applied to the pixels which are not in shadow, and only ambient light is applied to the pixels in shadow. The shadow stencil buffer needs to be reset to its initial values at the beginning of each frame, which is 1 in our case indicating no shadow is detected.

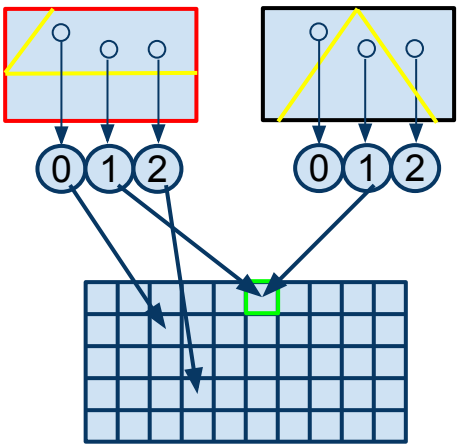


Figure 5.9. Shadow Stencil Update

Figure 5.9 illustrates the case where the same fragment is being processed by two different units for two different triangles. This occurrence only exists in `kernel1`, because, in `kernel2`, each fragment is always taken care of by a specific unit. The unit marked in red needs to update three positions in the shadow stencil buffer, while the unit marked in black only needs to perform one write. The update of the shadow stencil buffer is performed in a scattered way, which may lead to a performance degradation when the larger area of the screen is in shadow. The green mark in Figure 5.9 reveals the fact that the same stencil value can be updated by different units. In this case, synchronization is not required, because the shadow stencil buffer is always updated with the same value.

5.4.1 Future Implementation

As mentioned earlier, per pixel computations involve a large number of scattered writes if shadows are detected. One possible fix for this is to use another intermediate shadow stencil buffer. The intermediate buffer has the same size as the shadow stencil buffer but with a different ordering scheme. Each element in the intermediate buffer is indeed the shadow stencil value for each point, but they are arranged in the same order as the binned point data. Therefore, the update of the intermediate buffer follows the same pattern as the sample points fetching, which is coalesced within each unit.

As shown in Figure 5.10, after all triangles are rasterized, the shadow stencil values will then be reshuffled from the intermediate buffer to the shadow stencil buffer in its desired order. In this way, scattered writes are minimized.

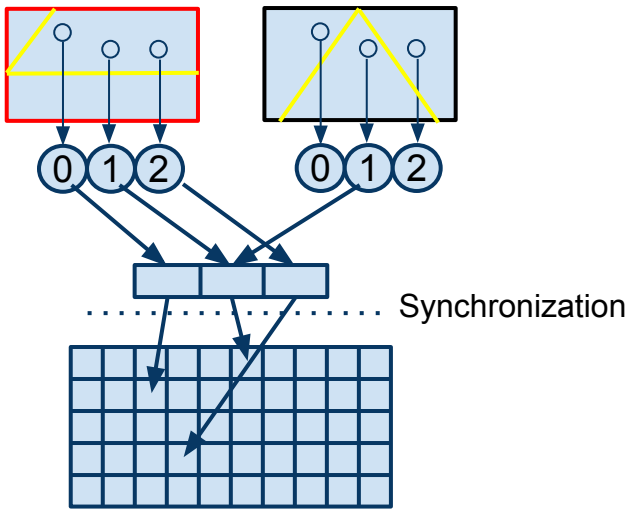


Figure 5.10. Synchronization Before Shadow Stencil Update

5.5 Execution Time Analysis

In order to understand the differences between the two rasterization kernels, please consider the follow analysis. Assume that we always have enough processors to execute the kernel, so all units are executed in parallel. To put aside the impact of the point distribution, assume that it always takes the same amount of time to execute one fragment.

The execution time of the kernel is dictated by the unit that takes the longest time to execute. For `kernel1` and `kernel2`, the execution times can be expressed as:

f^i : the number of fragments for triangle i.

t_{frag} : the time to execute one fragment.

n : the number of triangles

m : the number of units

$T_{kernel1\_exec}$: the execution time of `kernel1`

$T_{kernel2\_exec}$: the execution time of `kernel2`

$$\begin{aligned} T_{kernel1\_exec} &\approx t_{frag} \times max(f^1, f^2, f^3, .., f^n) \\ T_{kernel2\_exec} &\approx \frac{t_{frag}}{m} \times \sum_{i=1}^{n} f^i. \end{aligned} \tag{5.1}$$

A few observations can be made from Equation 5.1. If all the triangles are more or less of the same size, and if we can launch enough units, then the execution times of `kernel1` and `kernel2` should be the same. But the number of units that can be launched is associated with the number of groups we divide the fragments into, which cannot exceed the grid dimension. For most cases, it is a bad idea to assign just one fragment to each unit, and the grid dimension should be kept at a certain level. Therefore, m is normally smaller than n. In other words, if triangles are of equal sizes, `kernel1` is faster.

On the other hand, if the sizes of the triangles are terribly unbalanced, load balancing comes into the picture. There are cases where the execution time of `kernel2` could be even smaller than that of `kernel1`.

One may notice that the triangle access and process time is not included in Equation 5.1. So we can add these into the equations:

t_{tri} : the time to fetch and process one triangle

$$\begin{aligned} T_{kernel1\_exec} &\approx t_{frag} \times max(f^1, f^2, f^3, .., f^n) + t_{tri} \\ T_{kernel2\_exec} &\approx \frac{t_{frag}}{m} \times \sum_{i=1}^{n} f^i + n \times t_{tri}. \end{aligned} \tag{5.2}$$

From equation 5.2, we may realize another overhead of `kernel2` introduced by the triangle loop.

5.6 Performance Analysis

In this section, two main aspects that impact the performance of our rasterization kernel will be investigated. The first aspect is how the performance is affected by the number of triangles that need to be processed, and the second one is the influence of the point distribution.

5.6.1 Number of Triangles

Figure 5.11 compares the kernel performances when different number of triangles are processed. The comparison is made from the same view angle and with the same triangle data. When processing a large number of triangles, `kernel1` has better performances than `kernel2`. This result is accordant with the analysis we made in the previous section.

5.6.2 Point Distribution

In Figure 5.12 and Figure 5.13, the same scene are rendered from two different camera positions. The yellow dots in Figure 5.12a and Figure 5.13a represent the reconstructed irregular sample points from the light point of view. The camera position of Figure 5.12 results in a more uniform sample point distribution across the fragments, while Figure 5.13b illustrates a very unbalanced point distribution. Beside the fragment distribution, the unbalanced point distribution adds up another layer onto the unbalanced workload between different units.

Table 5.1 shows the performance impacts of the two different point distributions illustrated in the figures. Clearly the point distribution is associated with the camera-view, and the performance is effected from frame to frame.

Table 5.1. Performance Comparison

Point Distribution	Kernel 1	Kernel 2
View 1 (Figure 5.12)	490 fps	379 fps
View 2 (Figure 5.13)	190 fps	90 fps

5.6.3 Unit configuration

The kernels are developed to make use of dynamic unit configurations. Thus, the number of threads that participate in the process of one triangle can be changed. This is more of a compensation for the irregularity of the point data structure and the unbalanced workload between the units.

In Figure 5.14, we compared a number of unit configurations when rendering the same scene. Performance figures are collected from different camera-views and with different triangle numbers. The curves drawn in different colors represent the performance trends of four different unit sizes. When the triangle number is smaller than 5000, the unit size of 256 gives the best performance. For larger triangle numbers, the unit size of 64 gives the best performance. Therefore, different unit configurations should be chosen for different cases.

Furthermore, through the investigations we made, the unit configuration switching point is different from scene to scene. It does not only depend on the triangle number but also the point distribution. Therefore, it needs to be further investigated so as to make the dynamic unit configuration more accurate.

Besides, our solution is to run a number of tests for the scene we are going to render, and pick the best unit configuration for this specific scene.

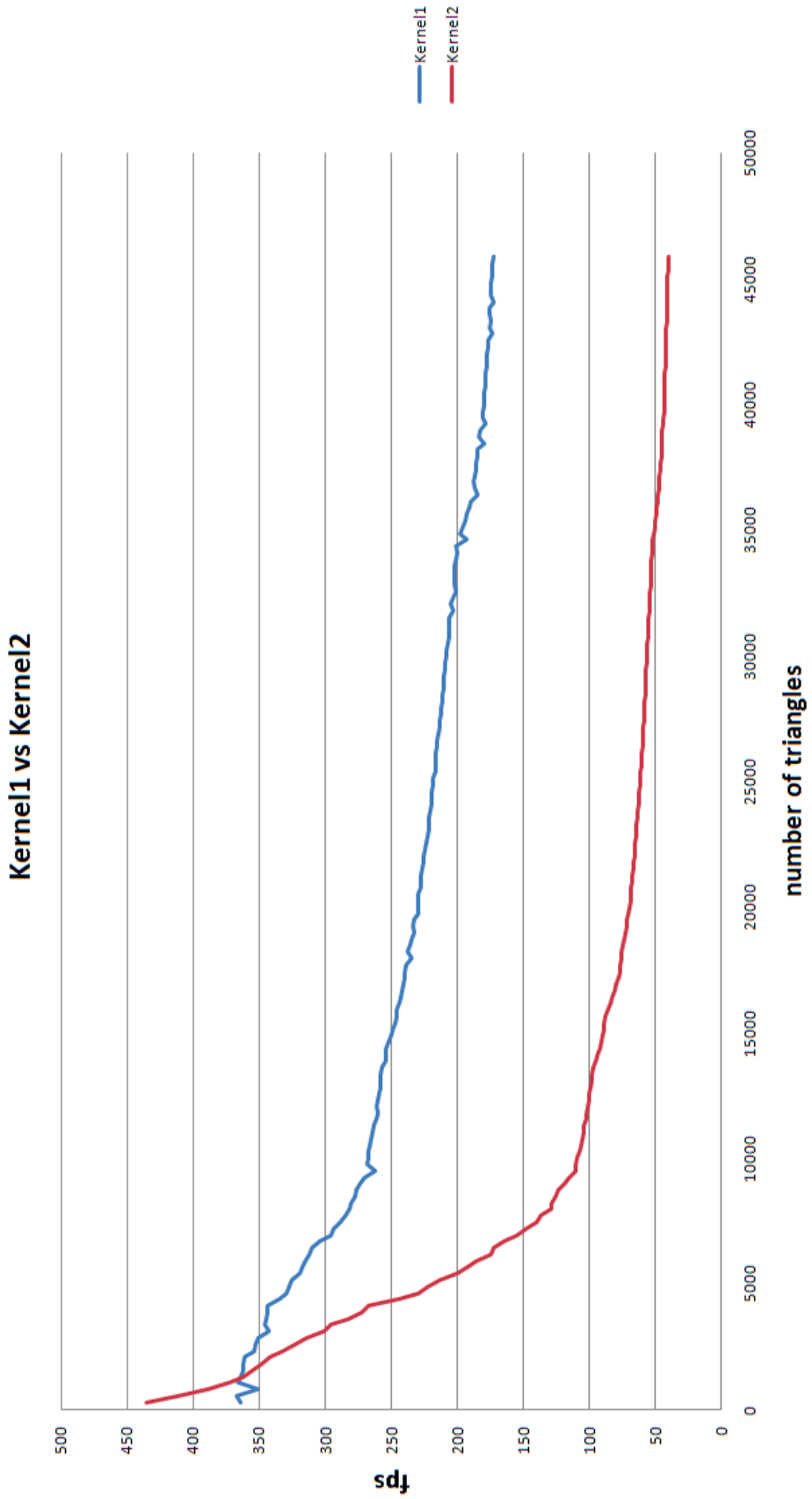
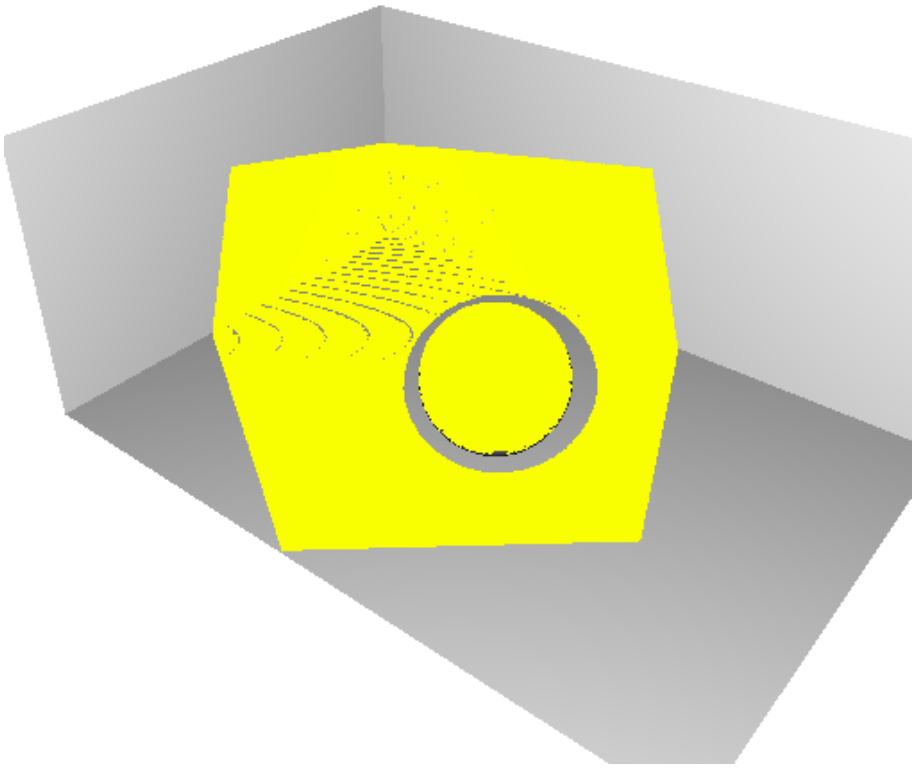
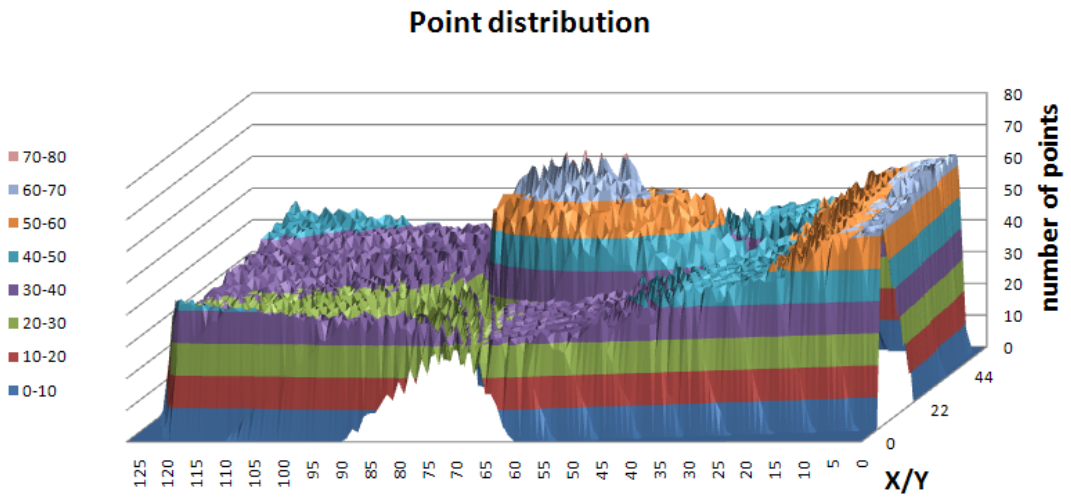


Figure 5.11. Triangle Number Impact

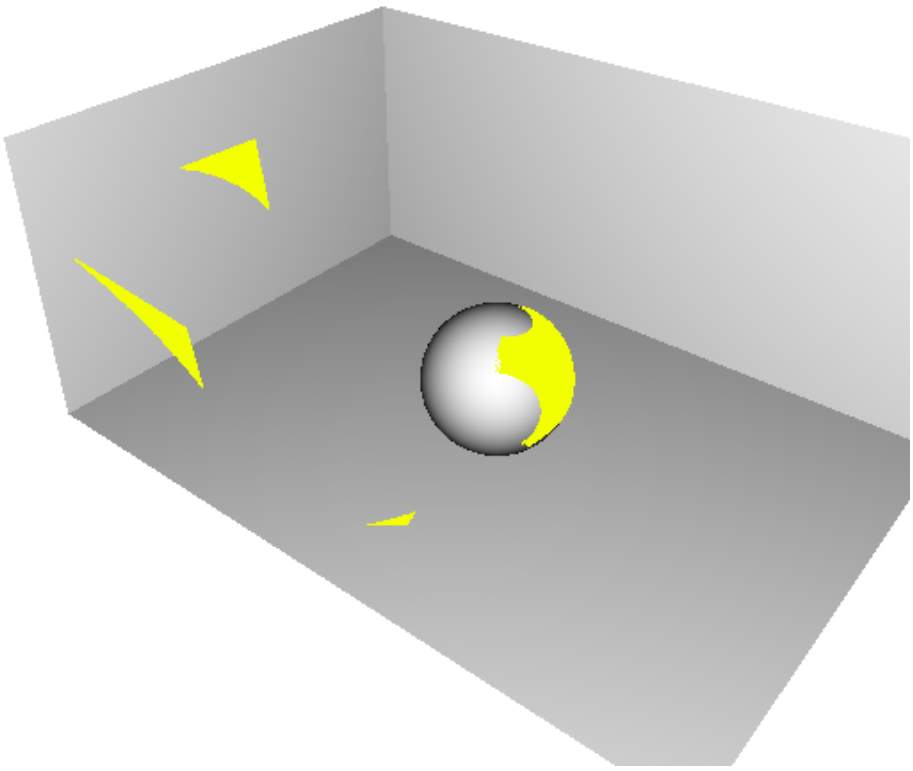


(a) Light View

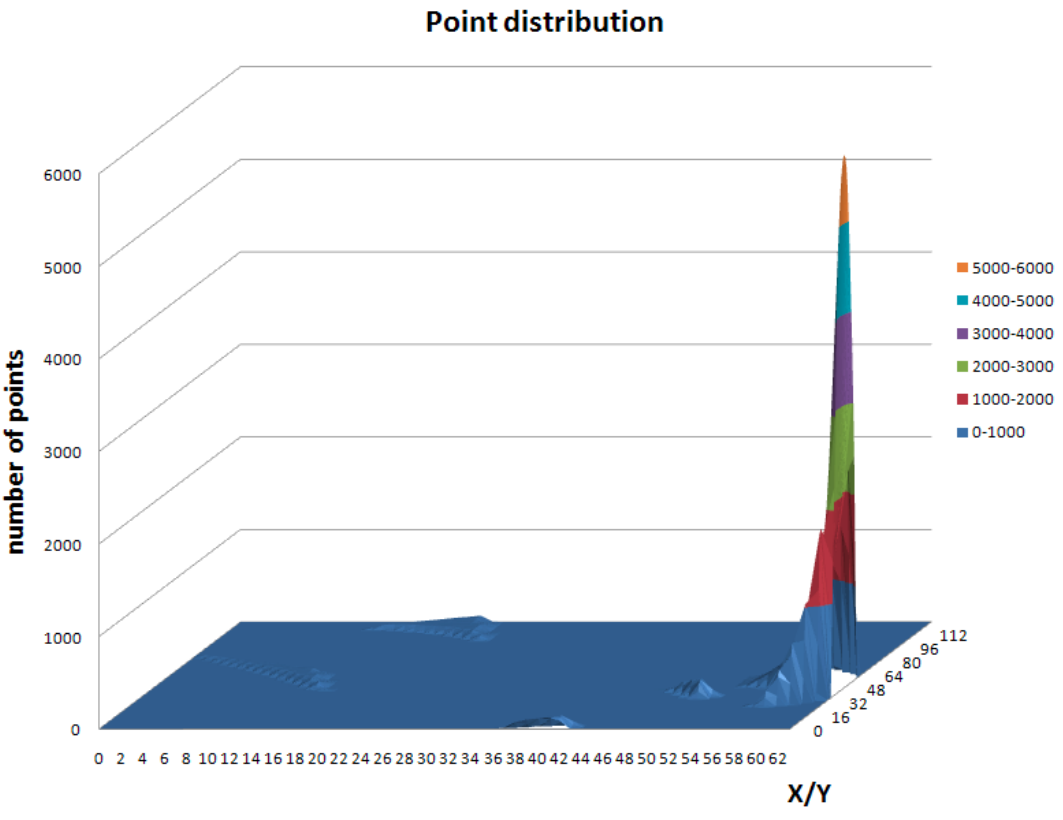


(b) Sample Points Grid Distribution

Figure 5.12. Averaged Points Distribution



(a) Light View



(b) Sample Points Grid Distribution

Figure 5.13. Unbalanced Points Distribution

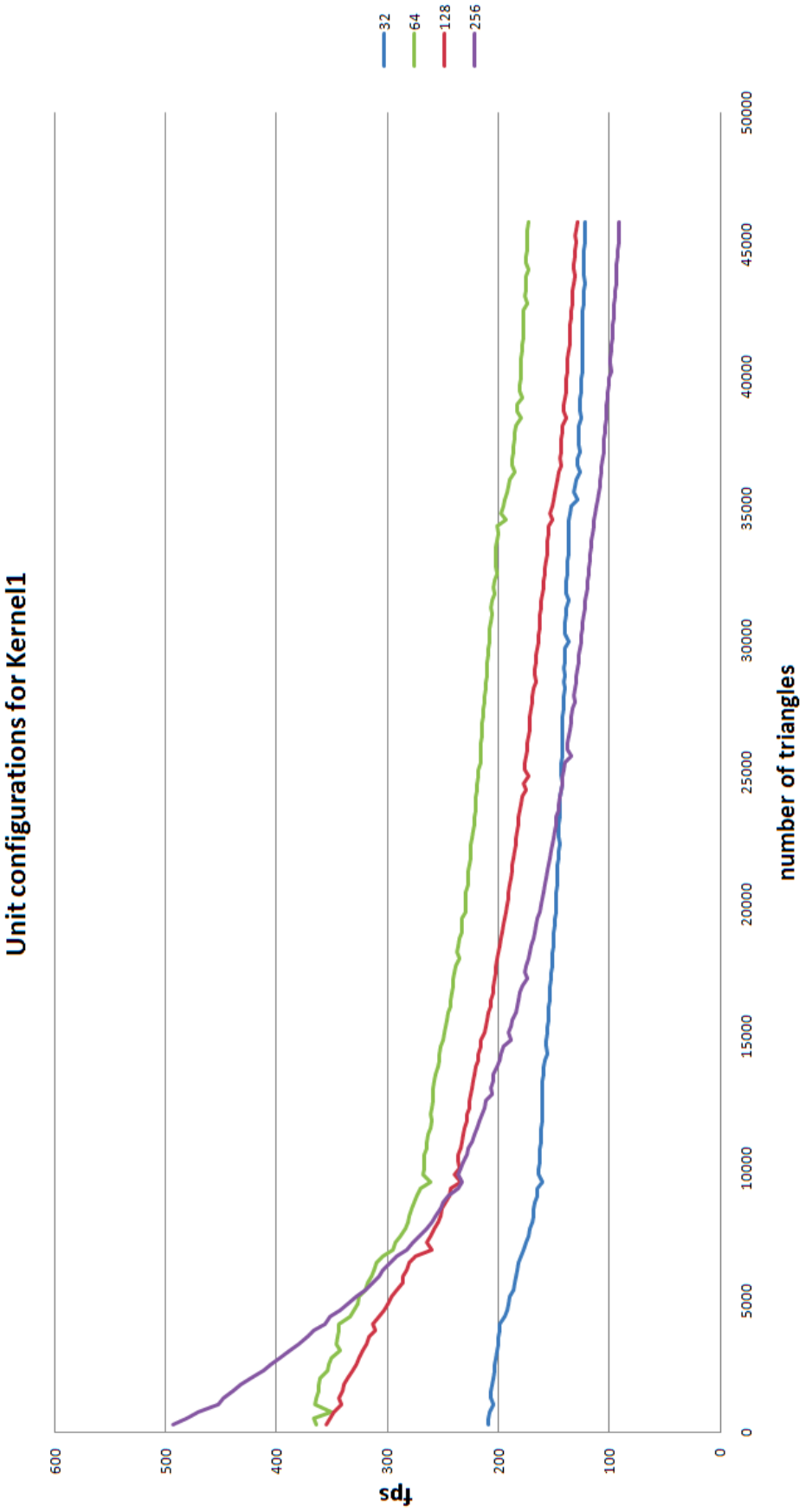


Figure 5.14. Unit Configuration

6 RESULTS

We have gone through all the different parts of our irregular shadow mapping design on CUDA. Here, we extend our discussion to a more generalized scope and conclude our work from a few different perspectives. First, we discuss the issues we have encountered and their proposed solutions. This is followed by the results on shadow quality and performance figures. Finally, we will have a few words on the future prospection of our work.

6.1 Artifacts

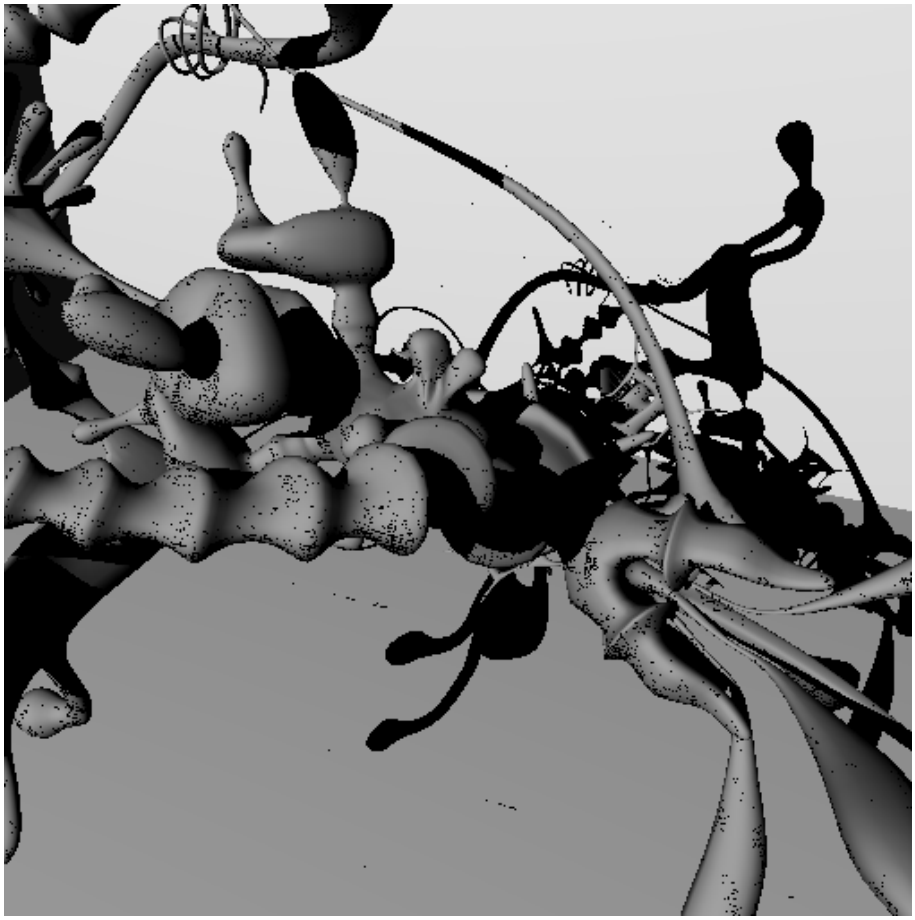


Figure 6.1. Artifacts

Conceptually, the irregular Z-Buffer algorithm does not introduce artifacts, but in a real implementation it is not really the case. Figure 6.1 shows the artifacts generated by our irregular rasterizer before any optimization has been made. Unexpected black dots appear on the geometry surfaces where no shadow should exist.

6.1.1 Floating Point Precision Error

We suspect that this is caused by the single precision floating point errors. In the world space, the irregular sample points should be lying right on the triangle surfaces where they have been reconstructed from, meaning ideally, in the light space, the depth of the sample point and the interpolated depth of the triangle on the sample point position should be the same. Due to the precisions lost in the floating point arithmetic operations, these two depth values become different in an unpredictable way. So we might not be able to determine the shadows correctly by comparing the depth values between the sample points and the triangles.

6.1.2 Biasing

Similar to the common fix we used in conventional shadow mappings, these artifacts can be easily eliminated by applying a biasing during the depth comparison. In our case, only a very small biasing value is needed, because we only need to compensate the small precision errors, and there are no spatial deviations in each sample of our irregular shadow map [AL04].

Not like the side affects we get by biasing the depth comparison in conventional shadow mappings where the shadows are sometime pushed off the object as if it is floating, the side effects potentially introduced by the biasing are barely perceivable in our shadows.

6.1.3 View Frustum Adjustment

Another aspect to keep the shadow generation in a good shape is to properly adjust the camera-view frustum.

The reconstruction of the sample points is dependent on the depth values generated in the camera space. The depth values are mapped to the range between 0 and 1 in a logarithmic scale. In order to let the sample points take up a bigger range of the depth value so they can be reconstructed more precisely, the far and near planes of the camera-view frustum need to be put close to each other as much as possible.

Especially when the far plane is exposed in the camera-view, if the far plane is unnecessarily located far behind the rest of the scene, the sample point distribution becomes terribly unbalanced. Not only the sample points' reconstruction becomes unreliable, but also the performance of the rasterizer is jeopardized.

6.1.4 Is Single Precision Floating Point Enough?

The entire rasterization process is implemented using single precision floating point operations. Although it causes artifacts, they can be easily removed. So there is no need to employ double precision floating point operations, which will lead to considerable performance degradation.

6.2 Shadow Quality

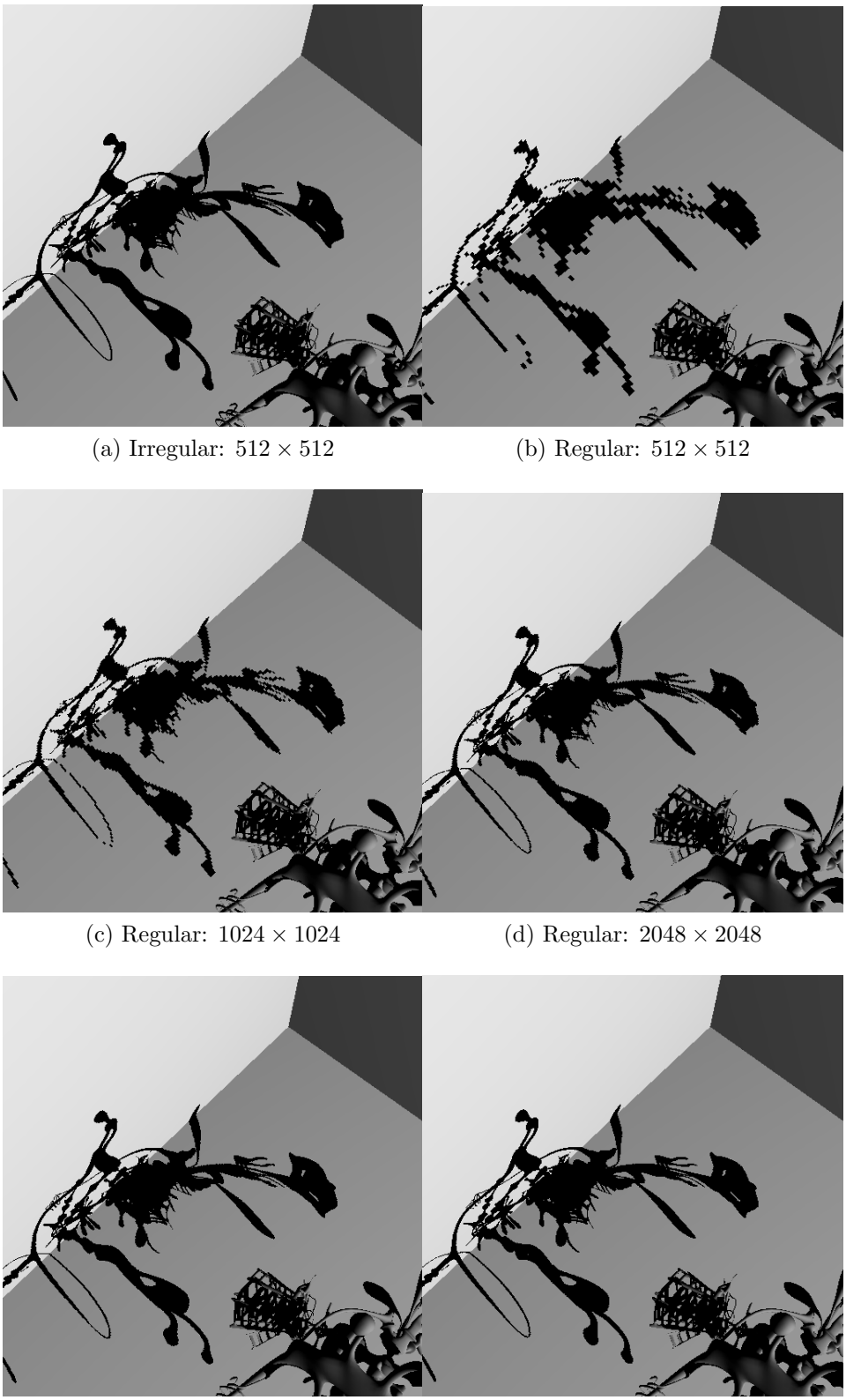
We simply want to demonstrate the rendering quality of our irregular shadow mapping by making a comparison with the conventional shadow mapping. Figure 6.2 shows a few pictures rendered by two different shadow generation approaches. All pictures are rendered from the same camera position in the same scene, and the same light position is shared by both methods. In order to improve the shadow quality of the conventional shadow mapping, different shadow map resolutions are used and compared. One can tell from the picture until the resolution of the conventional shadow map has increased to its maximum size possible (8192×8192), the rendering qualities of the two methods seem to be comparable (Figure 6.2a to Figure 6.2f).

But can we always achieve a comparable shadow quality in conventional shadow mappings by increasing the resolution? Actually, if we zoom in to the shadows, there is always a difference. Figure 6.3 shows two pictures rendered by the two shadow mapping approaches for the same scene but at a different camera position. The camera is moved closer to a small part of the shadow we just showed. Because irregular shadow maps are always adapted to the view, the quality of the shadow is always maintained regardless where we are looking at. Conventional shadow maps are always fixed. So if only we zoom in closely enough, artifacts are always noticeable unless the shadow map is perspectively adjusted to the camera view dynamically. It is always possible to construct cases where conventional shadow maps will fail for a finite resolution.

Thoughts on Shadow Volumes

We have not talked about shadow volumes [Cro77] so far, but it is definitely worth to mention. The shadow volumes algorithm takes advantage of the hardware rasterizer on GPUs, but the overheads are the detection of the silhouette edges and the building of the shadow volumes each frame.

One interesting fact is that the Z-pass or Z-fail tests performed in shadow volumes are actually equivalent to the irregular rasterization in irregular shadow mapping. It is exactly how the irregular rasterization looks like from the camera point of view. For both approaches, the basic ideas of the visibility tests are the same. There-



(a) Irregular: 512×512 (b) Regular: 512×512

(c) Regular: 1024×1024 (d) Regular: 2048×2048

(e) Regular: 4096×4096 (f) Regular: 8192×8192

Figure 6.2. Shadow Quality Comparison

(a) Irregular 512×512 (b) Regular 8192×8192

Figure 6.3. Shadow Quality Comparison: Zoom in

fore, in comparison with irregular shadow mapping, the shadow volumes algorithm is essentially more of a compromise to make use of the regular rasterization hardware; the downside of irregular shadow mapping is the lack of support from existing hardware.

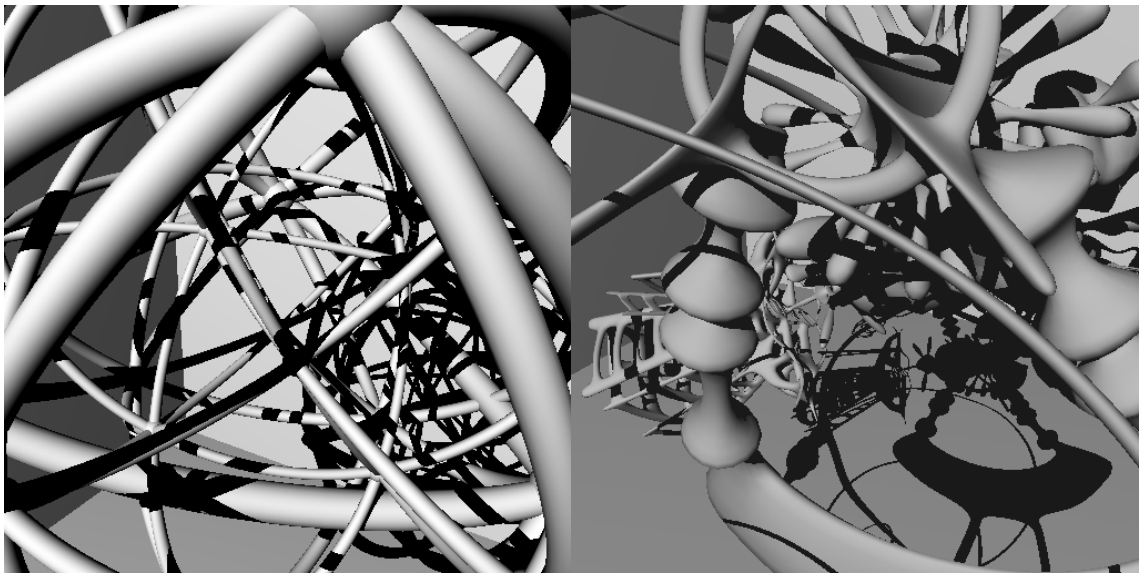
Then it comes to the question:“ For what cases does one solution outperform the other?” It is hard to draw a concrete conclusion here. But for very complex shadows, it could be complicated to compute the right silhouette edges and build corresponding shadow volumes; irregular shadow mapping is more of a concern about the load balancing and the performance of the software rasterizer.

6.3 Performance

Besides, a better performance is the real goal of our work. It has to be fast enough to be used in real-time rendering. Otherwise our work is of no interest to people.

We have discussed many aspects that may affect the performance of our system. Due to the irregularity of the data structure and the unbalanced workload, the overall performance of our irregular rasterizer varies as the view changes. We have, however, tried to minimize the fluctuation as much as we possibly can by applying techniques like triangle culling and dynamic unit configuration.

Figure 6.4 shows two models we tested with our irregular rasterizer. The resolution remains the same (512×512), atomic operation binning is used, and `kernel1` is chosen in order to present the best performance we have. The first model (Figure 6.4a)



(a) 24 Cell Tesseract $123K$ Tris 90 fps (b) Yeah Right! $190K$ Tris 65 fps

Figure 6.4. Irregular Rasterizer Performance on Geforce GTX 260

is a 24 cell tesseract which contains $123K$ triangles. The average performance we achieved was around 90 fps. If we put these two numbers together, the throughput of our irregular rasterizer was 11.1 million triangles per second. The second one (Figure 6.4b) has around $190K$ triangles, and the average performance we had was 64 fps, which give a throughput of 12.4 million triangles per second.

By looking at the performance figures pointed above, we are convinced that our irregular rasterizer is capable of carrying out real-time rendering tasks.

6.4 Soft Shadows

Up until now, we have been focused on hard shadows. Here, we discuss the possibilities to extend our work to generate soft shadows.

Gregory S. Johnson, the original author of the irregular Z-buffer algorithm, and others have discussed about soft shadow generations using irregular shadow mapping [JHH+09]. The basic idea can be summarized as follows (Figure 6.5): when rasterizing one sample point, we keep track of the minimum and maximum depth values of the shadow occluder on the sample point position. By comparing the two depth values, we can somehow detect the silhouette of the occluder, and determine whether the sample point is in the penumbra or the umbra of the shadow. Then instead of generating a stencil value, a factor indicating the darkness of the shadow can be given for each sample point. When all the darkness factors of all the sample points are collected, soft shadows can be rendered eventually.

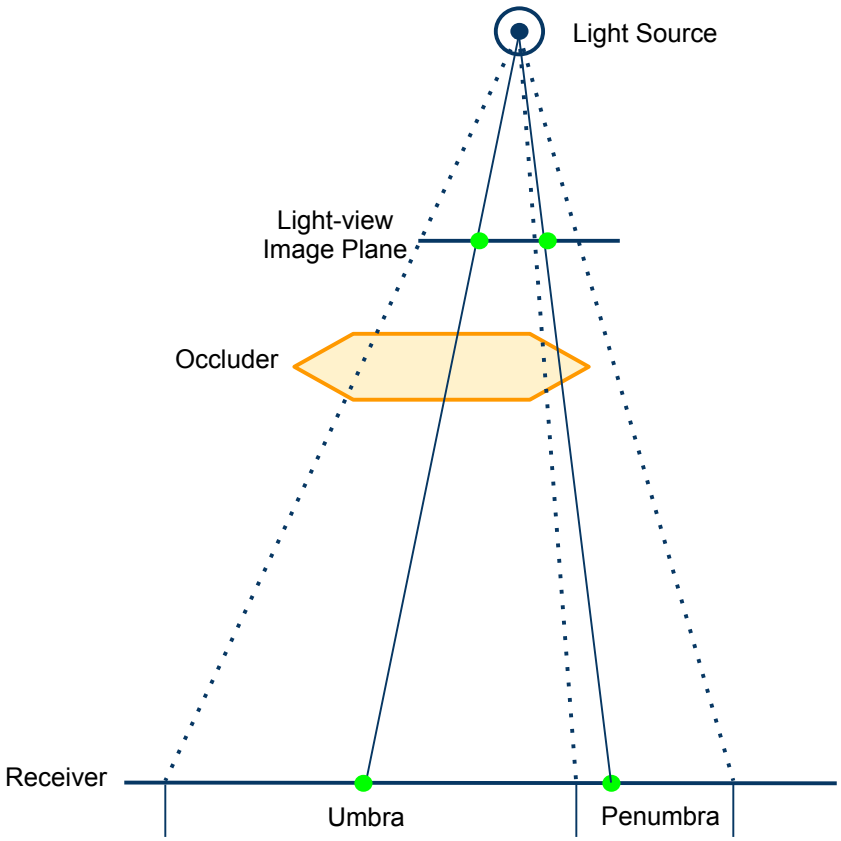


Figure 6.5. Soft Shadows

Comparing to `kernel1`, `kernel2` does not provide the best performance but is easier to extend for soft shadow generations. And that was the main reason why we kept `kernel2` as an alternative solution. In `kernel2`, the granularity of each thread within a unit covers the entire rasterization process of the sample point set assigned to this thread. Therefore, each pair of the minimum and maximum depth values can be calculated internally by a thread.

In addition, due to the time limitation of our work, `kernel2` is not fully optimized yet. For instance, we could potentially build the same 2D uniform grid for the triangle data, and the grid dimensions should be matching with the one we built for the point data. By doing this, the triangle loops within each unit are further reduced.

We propose the extension of soft shadow generations and further optimizations of `kernel2` as future work of our master thesis.

6.5 Conclusion

In this master thesis work, we managed to implement a fast triangle rasterizer using an irregular Z-Buffer algorithm entirely on GPUs. We were able to overcome various issues we met and designed the rendering system that generates high quality shadows with a satisfactory performance. Potential extension of our work is also discussed.

6.6 Acknowledgments

We are very thankful to our examiner and supervisor, Associated Professor Ulf Assarsson at Chalmers University of Technology, who gave us the chance to work on this project.

We also would like to express our appreciation to Erik Sintorn and Markus Billeter from The Computer Graphics Research Group at Chalmers University of Technology, who also offered great help for our work.

Bibliography

[Ahn08] Song Ho Ahn. Opengl frame buffer object (fbo). Online Tutorial, 2008. Available online.

[AL04] Timo Aila and Samuli Laine. Alias-free shadow maps. In *Proceedings of Eurographics Symposium on Rendering 2004*, pages 161–166. Eurographics Association, 2004.

[AM07] Tomas Akenine-Möller. Mobile graphics hardware. Course notes for the course Mobile Computer Graphics given at Lund University, September 2007. Available online (101 pages).

[Cro77] Franklin C. Crow. Shadow algorithms for computer graphics. *SIGGRAPH Comput. Graph.*, 11(2):242–248, 1977.

[Eng06] Wolfgang Engel. Cascaded shadow maps. In Wolfgang Engel, editor, *ShaderX5: Advanced Rendering Techniques*, pages 197–206. Charles River Media, Cambridge, MA, 2006.

[FLB+09] Kayvon Fatahalian, Edward Luong, Solomon Boulos, Kurt Akeley, William R. Mark, and Pat Hanrahan. Data-parallel rasterization of micropolygons with defocus and motion blur. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*, pages 59–68, New York, NY, USA, 2009. ACM.

[Gre08] Simon Green. Cuda particles. White paper, NVIDIA CUDA SDK, June 2008. Available online (12 pages).

[Gro08] Khronos Group. Opengl specification. Version 3.0, July 2008. Available online.

[Har07] Mark Harris. Optimizing parallel reduction in cuda. White Paper, November 2007. NVIDIA Developer Technology.

[HB] Jared Hoberock and Nathan Bell. thrust — c++ template library for cuda. http://code.google.com/p/thrust/. Version 1.2.

[HOS+] Mark Harris, John D. Owens, Shubho Sengupta, Stanley Tzeng, Yao Zhang, and Andrew Davidson. Cuda data parallel primitives library. http://code.google.com/p/cudpp. Version 1.1.

[JHH+09] Gregory S. Johnson, Warren A. Hunt, Allen Hux, William R. Mark, Christopher A. Burns, and Stephen Junkins. Soft irregular shadow mapping: fast, high-quality, and robust soft shadows. In *I3D '09: Proceedings of the 2009 symposium on Interactive 3D graphics and games*, pages 57–66, New York, NY, USA, 2009. ACM.

[JMB04] Gregory S. Johnson, William R. Mark, and Christopher A. Burns. The irregular z-buffer and its application to shadow mapping. Technical Report TR-04-09, April 2004. The University of Texas at Austin, Department of Computer Sciences.

[KBR06] John Kessenich, Dave Baldwin, and Randi Rost. The opengl®shading language. GLSL Specification, 2006. Language Version 1.20.

[KS09] Javor Kalojanov and Philipp Slusallek. A parallel algorithm for construction of uniform grids. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*, pages 23–28, New York, NY, USA, 2009. ACM.

[LGQ+08] D. Brandon Lloyd, Naga K. Govindaraju, Cory Quammen, Steven E. Molnar, and Dinesh Manocha. Logarithmic perspective shadow maps. *ACM Trans. Graph.*, 27(4):1–32, 2008.

[LGS+09] Christian Lauterbach, Michael Garland, Shubhabrata Sengupta, David Luebke, and Dinesh Manocha. Fast bvh construction on gpus. *Comput. Graph. Forum*, 28(2):375–384, 2009.

[LHLW10] Fang Liu, Meng-Cheng Huang, Xue-Hui Liu, and En-Hua Wu. Freepipe: a programmable parallel rendering architecture for efficient multi-fragment effects. In *I3D '10: Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, pages 75–82, New York, NY, USA, 2010. ACM.

[NVI10] NVIDIA. Nvidia cuda™programming guide. Version 3.0, Feburary 2010. Available online (101 pages).

[OLG+07] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Tim Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, March 2007.

[SD02] Marc Stamminger and George Drettakis. Perspective shadow maps. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 557–562, New York, NY, USA, 2002. ACM.

[SEA08] Erik Sintorn, Elmar Eisemann, and Ulf Assarsson. Sample based visibility for soft shadows using alias-free shadow maps. *Computer Graphics Forum (Proceedings of the Eurographics Symposium on Rendering 2008)*, 27(4):1285–1292, 2008.

[SHG08] Shubhabrata Sengupta, Mark Harris, and Michael Garland. Efficient parallel scan algorithms for gpus. NVIDIA Technical Report NVR-2008-003, December 2008.

[SHG09] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore gpus. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society.

[Wil78] Lance Williams. Casting curved shadows on curved surfaces. *SIGGRAPH Comput. Graph.*, 12(3):270–274, 1978.

[ZHWG08] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-time kd-tree construction on graphics hardware. *ACM Trans. Graph.*, 27(5):1–11, 2008.