ABSTRACT

SRINIVASAN, ABHINAV. Adaptive Frameless Rendering. (Under the direction of Benjamin Watson.)

We present a graphics processing unit (GPU) based implementation of Adaptive Frameless Rendering, a departure from the traditional “double-buffered” renderer that has the potential to dramatically increase rendering speed. While previous implementations of Adaptive Frameless Rendering have relied on simulated use of the GPU through shaders, our implementation uses Nvidia’s OptiX ray tracing engine to leverage potential of the GPU. Our implementation faces several new challenges as it is a fully interactive implementation of Adaptive Frameless Rendering, which has parallel sampling with competition between threads. Other challenges in our implementation include the effective use of time between the CPU and the GPU, and the interactive nature of the application, as previous implementations were much slower and non-real time. Also unique to our implementation is the parallel computation that is introduced by the use of the GPU, while sampling in previous implementations was done sequentially. Our implementation also includes reconstruction in the real time closed loop, whereas previously the sampler and reconstructor were decoupled in an open loop system.
© Copyright 2016 by Abhinav Srinivasan

All Rights Reserved
Adaptive Frameless Rendering

by
Abhinav Srinivasan

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Science

Raleigh, North Carolina
2016

APPROVED BY:

Christopher Healey
John Turner Whitted

Benjamin Watson
Chair of Advisory Committee
DEDICATION

I would like to dedicate this work to my parents, and to my brother.
BIOGRAPHY

Abhinav Srinivasan was born in Madras (Chennai), India on February 17th, 1993. He completed High School in Sri Sankara Senior Secondary School, Chennai, India, graduating in 2010. He obtained his B.Tech in Information & Communication Technology from SASTRA University, India in 2014. He began his graduate studies in North Carolina State University, Raleigh, U.S.A, in August of 2014, where his research focussed on Computer Graphics and rendering techniques under the supervision of Dr. Benjamin Watson.
ACKNOWLEDGEMENTS

I would like to thank my mother, my father, and my brother, for their constant support and encouragement.

I would also like to thank Sri Sankara Senior Secondary School - the school that I considered home for a decade, and the school that adopted me as one of its own. I would like to thank SASTRA University - the institution that shaped my outlook and instilled passions in me that I would not have been able to discover elsewhere. I would also like to thank North Carolina State University for giving me an opportunity to pursue academic excellence, and for helping me find strength in the wolfpack.

I would also like to extend my gratitude to Dr Benjamin Watson for giving me the wonderful opportunity to work on a project that was simultaneously challenging and exciting. His constant words of encouragement and his mentorship brought forth the best I had within myself. I would also like to thank Dr Christopher Healey and Dr John Turner Whitted for consenting to being a part of my thesis committee.

I would like to thank all of my friends for being companions with me on this truly marvelous journey. I would especially like to thank Adam Marrs, on whom I have relied on whenever the magnitude of the obstacles that I’ve faced have overwhelmed me. His perspective and inputs have been invaluable to me and were a critical component in this project.

Finally, I would also like to thank you. Yes, you. Thank you very much for reading this. The effort that went into this work would be meaningless if it weren’t for you.
# TABLE OF CONTENTS

**LIST OF TABLES** .......................................................... vi

**LIST OF FIGURES** ....................................................... vii

Chapter 1 Introduction to Adaptive Frameless Rendering  .................. 1
  1.1 Framed Rendering and Double Buffering  ........................... 1
  1.2 Frameless Rendering ............................................... 2
  1.3 Adaptive Frameless Rendering  .................................... 2

Chapter 2 Related Work .................................................. 3
  2.1 Single Buffering .................................................... 3
  2.2 Multiple Buffering .................................................. 4
  2.3 Frameless Rendering ................................................ 7
  2.4 Adaptive Frameless Rendering ..................................... 9

Chapter 3 Algorithm ....................................................... 11
  3.1 Overview ............................................................ 11
  3.2 Sampling ............................................................. 13
  3.3 Reconstruction ..................................................... 20
    3.3.1 Mechanism .................................................... 20
    3.3.2 Implementation ............................................... 24
  3.4 Data Structures .................................................... 26
  3.5 Pseudocode ........................................................ 27
  3.6 Implementation Details ............................................ 27
  3.7 New Problems Encountered ........................................ 29

Chapter 4 Results .......................................................... 30

Chapter 5 Discussion and Future Work ................................... 32

Chapter 6 Conclusion ...................................................... 35

References ................................................................. 36
LIST OF TABLES

Table 3.1  Pseudocode .................................................. 28
Table 4.1  Amount of time consumed in seconds by the CPU and GPU for various scenes in one iteration of the algorithm. .................. 31
LIST OF FIGURES

Figure 2.1 Image Tearing (courtesy Wikipedia) ................................. 4
Figure 2.2 A depiction of single, double, and triple buffering courtesy of [6]. In single buffering, the front buffer is always shown. In double buffering, one buffer is the front buffer, and the other the back buffer. These two buffers swap positions for each frame. In triple buffering, an additional pending buffer is present. In triple buffering, a buffer is first cleared, and rendering to it is begun (pending). Second, the system continues to use the buffering for rendering until the image has been completed (back). Finally, the buffer is shown (front). ......................... 6
Figure 2.3 Motion Blur in Frameless Rendering ............................... 8
Figure 2.4 Practically Frameless Rendering remaps how frame-buffer pixels are displayed. Image courtesy [24] ................................. 9
Figure 3.1 System Components ....................................................... 12
Figure 3.2 Tiles cluster around edges and areas of motion, thereby biasing the sampling towards that region. To prevent tiles from getting too big or too small, and thereby skewing sampling too much, it is also reasonable to restrict the maximum and minimum sizes that tiles can take. For example, the bottom left corner of this image is occupied by one very big tile. If a new object were to appear there, then it takes longer to capture the variance in that region. ......................... 14
Figure 3.3 Because tiling is biased towards edges and areas of motion, that is where sampling takes place too. In this picture, new samples are marked by a green pixel. So it’s clear that towards the edges, where the spatio-temporal variance is high, more samples are taken. And in regions with low variance, such as plain surfaces and background, very few samples are taken. ................................. 15
Figure 3.4 Crosshairs, which are stored in the deep buffer, are using to calculate the gradients. We used the symmetrical crosshairs initially, but have since modified the implementation to use the tetrahedron crosshairs. Previous of adaptive frameless rendering by [10] have used the symmetrical crosshairs. ................................. 17
Figure 3.5 Pixie dust as seen in traditional frameless rendering. A characteristic of frameless rendering is that the viewer can see pixel samples from multiple moments in time. This results in pixie dust as seen here. Theoretically, in traditional reconstruction, unless a pixel is sampled again, it continues to have old sample information. ................................. 21
Figure 3.6 Adaptive Reconstruction. At static moments, the difference between traditional and adaptive reconstruction is minute. However, once the scene becomes dynamic, then adaptive reconstruction gives a much better quality image than the traditional counterpart. This figure shows adaptive reconstruction at a dynamic moment. Notice how the pixie dusts at the trailing edge have become significantly blurred.

Figure 3.7 A sampling density map (left) is used by the reconstructor to determine the expected local sample volume $V_s$ and the tile gradients (right) $G_x$, $G_y$, and $G_t$. Here, $G_x$, $G_y$, and $G_t$ are shown in blue, green, and red, respectively.

Figure 4.1 Adaptive Frameless Rendering on a simple, one plane scene. The picture on the left shows the scene during a still moment, and the picture on the right shows the scene when there is motion.

Figure 4.2 Adaptive Frameless Rendering on a more complex scene. The picture on the left shows the scene at a still moment, and the picture on the right shows the scene when there is motion.
Chapter 1

Introduction to Adaptive Frameless Rendering

1.1 Framed Rendering and Double Buffering

In a framed display, the goal of the rasterizer is to assign color values to the pixels, and to render the scene correctly. However, if we only have one buffer, then the viewer can see the polygons as they are being drawn on screen, a phenomenon known as flickering. To avoid this scenario, “double buffering” is used. In double buffering, the scene is rendered off-screen, on a so-called back buffer. Once the scene is rendered in the back buffer, the contents of this buffer are swapped with the front buffer which was previously displayed on screen [6]. This swapping of the buffers usually occurs when the electron gun of the microscope cannot disturb the display, a time known as vertical retrace. However, if the buffer swap does not take place during such a time, then the electron gun of the microscope renders a portion of the screen based on the previous front buffer, and the remainder of the screen based on the values present in the new front buffer. If the changes in these buffers is significant enough, then this leads to a visual artifact called tearing. Additionally, in double buffered displays, because the previous image is shown to the user until the next image is ready to be displayed, there is always a one frame delay in the image that the user sees. In environments where a tight coupling between the user and the system is required, this can be an impediment to interactivity [9].
1.2 Frameless Rendering

The problems presented in double buffered rendering are avoided in Frameless Rendering by randomizing the order in which new samples are taken, and by using just one buffer. The use of just a single buffer means that there is no one frame delay in the image that the human viewer gets to see. The usual tearing that is associated with double buffering is also absent in frameless rendering. However, because the samples are taken at a random order, the resulting image that is produced has the disadvantage of having a noisy, speckled look. These noisy grains, called \textit{pixie dust} give an appearance similar to motion blur, but the overall image avoids the unpleasant “jerkiness” of tearing.

1.3 Adaptive Frameless Rendering

Adaptive Frameless rendering can reduce the speckled look produced by frameless rendering. Because a frameless renderer is unbiased, all regions are given equal importance when sampling. However, an adaptive frameless renderer biases the process by introducing an importance sampling method that samples more in regions of high error (such as spatial edges and regions of motion), and less in regions of low error (such as plain surfaces and regions of low activity). While most importance sampling methods are adaptive across space, adaptive frameless rendering is adaptive across both space and time [11].

Even with adaptive sampling, the noise is evident. Thus, adaptive frameless rendering also introduces an adaptive reconstruction mechanism. In frameless rendering, the most recent sample taken at any given pixel is displayed to the user, a process referred to as \textit{traditional reconstruction}. However, adaptive frameless rendering makes use of a temporally deep buffer for storing samples, and then convolves these samples with space-time filters to produce the reconstructed image [10].
Chapter 2

Related Work

In display systems, a major problem we need to address is when a scene is drawn, and when the user gets to see the scene. As our work focuses extensively on this core issue, it is important to understand the existing approaches to solving it. This section will concentrate on providing a detailed explanation of the various mechanisms, while addressing their merits and limitations. This comprehensive review will shed light on the rationale behind the advantages of Adaptive Frameless Rendering, while making a case for its necessity in low latency rendering systems.

2.1 Single Buffering

In a single buffer system, as there is only one buffer, the same buffer that is drawn into is also visible to the user at the same time. As primitives for a frame are drawn, we will see more and more of them as the monitor refreshes, which is an unconvincing effect. Single buffering presents problems even if the frame rate is equal to the monitor’s update rate. For instance, when we clear the buffer, or draw a large polygon, then we will briefly be able to see partial changes as the beam of the monitor passes through areas that are being drawn. This artifact is called tearing because it gives the appearance of the image being split into two (see Figure 2.1). However, single buffering can be useful in situations when we need to update the drawing area infrequently [6].
2.2 Multiple Buffering

Double Buffering

For the viewer to not see the actual rendering process, and instead only see the final rendered image, more than one buffer is needed. This is known as double buffering. So, in double buffering, two separate buffers are maintained, and at any given time, only one of these is shown to the viewer. The buffer that the viewer sees is called the front buffer. The buffer that is hidden from the viewer is where the actual rendering of the primitives takes place. This buffer is called the back buffer. After the primitives have been rendered in the back buffer, the graphics driver “swaps” the two buffers, thereby ensuring that the viewer only sees a completely rendered image at all times. Typically this buffer swap happens after the last scan line has been traced, when the electron gun moves to the start of the scan lines. This time is known as “vertical retrace”. Synchronising the buffer swap with the vertical retrace ensures that all pixels that the user sees are always from the same instant of time, and so is desirable in most situations [6]. Following
the buffer swap, the new back buffer receives all the graphics commands, and the new front buffer is visible to the user, thereby repeating the entire process all over again (see Figure 2.2).

**Triple Buffering**

Another form of buffering system worth mentioning is *triple buffering*. Just like in double buffering, the viewer only sees one buffer at a time. In triple buffering, the remaining two buffers are “off-line” i.e., not visible to the viewer. So there are two back buffers, and just one front buffer. In double buffering, after the scene has been drawn to the back buffer, the renderer has to wait for a buffer swap until it starts drawing again. During this interval, neither buffer can be accessed, and so crucial milliseconds are spent waiting for a buffer swap. In order to rectify this problem, and in order that the renderer does not wait, an additional buffer is provided, called the “pending buffer”. In a double buffered system, the graphics pipeline will be bottlenecked, waiting for a buffer swap to access a buffer. Triple buffering solves this problem by making sure that the pending buffer can be used in this interval, thereby not stalling the graphics pipeline (see Figure 2.2). Triple buffering can also lead to tearing if the monitor refresh and the buffer swap are not synchronized. So VSYNC (see below) can be used in triple buffering as well.

As attractive as triple buffering is, this introduces even more latency to the process. Between the time the scene is first drawn to the pending buffer and the viewer sees the scene, the scene has aged two frames. This delays reaction to user inputs such as keystrokes, and mouse events [6]. Theoretically, more than three buffers can be used, at the cost of increased potential latency.

**VSYNC and GSYNC**

In multiple buffering, a buffer swap can theoretically happen at any time. If a buffer swap happens in the middle of when the scene is being drawn on screen, then half the screen contains information from the old frame, and half the screen will have information from the new frame. This causes a visual artifact called *tearing*. To avoid a buffer swap from taking place in the middle of a monitor refresh, the buffer swaps are synchronized with the vertical retrace of the monitor. This procedure is called *VSYNC* for Vertical Synchronization. VSYNC works very well if the frame rate of the application is greater
Figure 2.2: A depiction of single, double, and triple buffering courtesy of [6]. In single buffering, the front buffer is always shown. In double buffering, one buffer is the front buffer, and the other the back buffer. These two buffers swap positions for each frame. In triple buffering, an additional pending buffer is present. In triple buffering, a buffer is first cleared, and rendering to it is begun (pending). Second, the system continues to use the buffering for rendering until the image has been completed (back). Finally, the buffer is shown (front).
than the monitor refresh rate. In such cases, the refresh rate is the limiting factor, and the screen is redrawn at that rate, but with the added advantage of not having any tearing.

However, if the frame rate of the application is lesser than the monitor refresh rate, then VSYNC’s disadvantages become noticeable. If we had an application updating at 60 FPS, and a monitor with a 75 Hz refresh rate, then the framebuffer is updating at 80% the refresh rate. However, if VSYNC is enabled, then the frames must be copied to the screen buffer in synchronization with the vertical retrace. So the application misses the deadline to swap buffers in every other cycle, so we’ll end up with half of the refresh rate as the framerate, i.e. 37.5 Hz. This is significantly less than the 60 FPS which we could have achieved otherwise [5].

So, VSYNC’s method of synchronizing the buffer swap in the GPU to the display device has its own limitations. To combat this, and to eliminate screen tearing while keeping input lag low, there exist display devices with GSYNC capability. In GSYNC, the display device (monitor) synchronizes its refresh rate to match the buffer swaps of the GPU, rather than the other way around [2]. So, instead of the GPU being the slave to the monitor as is the case with VSYNC, the monitor is the slave to the GPU.

### 2.3 Frameless Rendering

Frameless Rendering offers unique flexibility in terms of rendering as it can respond to change with very little delay, and at any location on the image. Although frameless rendering requires a per-sampling rendering algorithm such as real-time ray tracing, [17] makes a compelling case for a frameless renderer.

Tearing is a visible artefact as is seen in Figure 2.1 and it has distinct horizontal features. This is because the pixels are updated in a sequential fashion. Randomizing the order of the update of the pixels, however, eliminates tearing. Even in a random order update of pixels, there will be information present from different time points at screen. However, they will be dispersed across the screen and there will not be the sudden breaks as seen in Figure 2.1. So Frameless Rendering exhibits a fluid motion instead of the sudden jerkiness that is associated with double buffered rendering [9]. This results in an approximation of motion blur. Motion blur is an effect that arises when the camera shutter is open for an extended period of time, and so any motion that occurs during that time is captured in the same snapshot [21]. See Figure 2.3
Motion blur is more desirable than tearing, especially in applications which rely on a tight coupling between the user input and the system response, as is the case with virtual reality. The reason motion blur is preferable to tearing is because the impact of latency felt by the user can be reduced by displaying a crude image when the scene is changing, and progressing to higher and higher quality images as the scene stabilises [8]. This process, of moving from a coarser to a finer image, thereby converging on an ideal image, is called the “golden thread”. In Frameless Rendering, as the motion of the user and the motion of the scene slows down, the motion blur also reduces and the image finally converges on the ideal image without any artefacts.

[25] categorizes Frameless Rendering techniques based on how the technique handles aging pixels and how pixels are chosen to be updated. This includes boundary importance sampling (where each pixel maintains an approximation of gradient magnitude) and whether or not aging pixels borrow information from updated neighbors.

[24] presents a “practically frameless” rendering method, which relies on the manipulation of the frame buffer and its mapping to pixels. Instead of using an identity map which displays pixel \((x,y)\) at position \((x,y)\), all the pixels in the top left quadrant to
display at position \((2x, 2y)\) instead. See Figure 2.4 for a detailed illustration of the mapping. This mapping is achieved by either copying pixels between buffers or by address remapping. While this method is not truly frameless, it does update only a quarter of the pixels at each time step. It also merely approximates motion blur. The stochastic nature of updating samples in Frameless Rendering is absent in this method.

![Figure 2.4](image.png)

Figure 2.4: Practically Frameless Rendering remaps how frame-buffer pixels are displayed. Image courtesy [24]

### 2.4 Adaptive Frameless Rendering

Previous work on Adaptive Frameless Rendering, relied on simulating GPU side performance via the use of shaders [10]. And while [10] and [11] were both open loop systems, ours is a closed loop system, in which feedback is in real time and instantaneous.

[19] proposes a “Frameless Volume Visualization” system that uses a multi-tiered stream of samples rendered framelessly to reconstruct a high resolution and high framerate image. This effectively decouples the rendering system from the display system, which “is particularly suitable when dealing with very high resolution displays or expensive rendering algorithms where the latency of generating complete frames may be prohibitively expensive for interactive application.” But while [11] used spatial and temporal filters to improve the quality of the rendered image, [19] does not. Instead, [19] used “a high order filter over a structured grid to produce a spatially and temporally
As mentioned previously, Adaptive Frameless Rendering finds numerous applications in low-latency rendering contexts. Naturally, one such application is in virtual reality and augmented reality. The following works explore the use of Adaptive Frameless Rendering in such applications.

While most low latency rendering techniques have been applied to displays with standard video interfaces such as VGA, DVI, or HDMI, [26] explores low latency rendering on head-mounted AR displays. By maintaining an estimate of the image that the user currently perceives, and by reducing the difference between that image and the newly generated image from the tracking data, their system creates an image that is “neither here nor there”, but is constantly approaching the moving target.

[14] examines the behaviour of frameless renderers in terms of latency, specifically in the context of Virtual Environments. By using a custom frameless renderer in combination with an Oculus Rift DK2, and with the help of a high and low speed video of the system being used, they were able to confirm a latency of $\sim 1$ ms.

[22] used some concepts from [11] such as spatial and temporal coherence exploitation in order to implement a render cache and the edge-and-point image directly on graphics hardware. While not a frameless renderer, their system used a multi-pass CPU/GPU hybrid to improve framerates while freeing up resources in the CPU side in order to perform higher-quality shading such as global illumination.
Chapter 3
Algorithm

3.1 Overview

The Adaptive Frameless Renderer is split between the CPU and the GPU. The CPU
is primarily responsible for managing the tiling system and for displaying the output
buffer via OpenGL. The GPU is responsible for actually shooting rays and performing
reconstruction. Another role that the GPU plays is in sample reprojection. The variance
calculations, which are across both space and time, are based on the samples stored in a
temporally deep buffer. The variance calculation was initially implemented on the CPU,
but has since been moved to the GPU for faster performance. In order to keep track of
areas of edges and motion, each tile’s error measure is computed across both space and
time. This error measure is the variance of the luminance value, calculated according to
the NTSC formula [3],

\[
luminance = 0.299 * R + 0.587 * G + 0.114 * B \tag{3.1}
\]

where R, G, and B are the red, green, and blue channels of the color respectively.

To make the frameless rendering adaptive, we make use of the variance calculations
mentioned above, the idea being that regions of high variance need to be sampled a lot,
and regions of low variance need only be sampled sparsely [16]. This is achieved by the
use of a quadtree. The root of the quadtree is the entire image, and the leaves of the
quadtree are individual pixels (or even sub-pixels, in case of supersampling) [10]. The
number of leaf tiles remains a constant (in our case, 256) throughout the program, but
Figure 3.1: System Components
the size of each individual leaf tile will vary based on the error of that tile. A very large leaf tile indicates that the region has very little error, and a very small leaf tile indicates a region with very high error across space-time, as shown in Figure 3.2. The number of samples per tile, i.e. the number of rays shot per tile by the GPU in one dispatch, remains a constant. So, a leaf tile spanning, say 16 by 16 pixels, and another leaf tile that is 4 by 4 pixels in size, will both be sampled the same number of times, in a random order within the tile. This ensures that the high error 4 by 4 tile will have more up-to-date pixels than the low error 16 by 16 pixel tile, thereby biasing the sampling towards the edges, motion, and areas of high spatio-temporal error, as shown in Figure 3.3. As we sample all the tiles in one GPU dispatch, we still remain sensitive to newly emerging details from the larger tiles too.

3.2 Sampling

GPU

On the GPU, each thread is responsible for managing one leaf tile. Each thread first shoots a fixed number of rays according to Optix’s mechanism in a random order to the tile it is assigned. When a thread tries to shoot a ray to a pixel, it first checks to see if the pixel is being used by another thread. If the pixel is free, a ray is shot through that pixel. If that pixel is not free, however, then the thread proceeds to find another pixel to sample, thereby not waiting on that busy pixel. When a ray is shot, the color value of that point is then noted, and then assigned to the front of the deep buffer to which it corresponds. The oldest value of the deep buffer is erased, and the new value takes its place. This process, of updating the deep buffers makes it essential that the deep buffer be a circular list to avoid spending excessive amounts of time in performing updates. There is a semaphore buffer that is maintained that contains the per pixel semaphore to indicate whether a pixel is free or busy. There is also a G-buffer that is maintained, to indicate the position of the object (in world coordinates) that the ray has just hit. The G-buffer, and the stencil buffer are updated when the output buffer has been written to, after a ray shoot is successful. Now, the thread proceeds to perform re-projection.

The number of re-projections performed per tile, again, is the same for all tiles irrespective of their size. The ratio of number of new samples to the number of reprojected samples is 1 : 3, i.e. for every new sample, there are three times as many reprojected
Figure 3.2: Tiles cluster around edges and areas of motion, thereby biasing the sampling towards that region. To prevent tiles from getting too big or too small, and thereby skewing sampling too much, it is also reasonable to restrict the maximum and minimum sizes that tiles can take. For example, the bottom left corner of this image is occupied by one very big tile. If a new object were to appear there, then it takes longer to capture the variance in that region.
Figure 3.3: Because tiling is biased towards edges and areas of motion, that is where sampling takes place too. In this picture, new samples are marked by a green pixel. So it’s clear that towards the edges, where the spatio-temporal variance is high, more samples are taken. And in regions with low variance, such as plain surfaces and background, very few samples are taken.
samples. For reprojection, the thread tries to access a pixel within the tile to which it is assigned, checking to see if it is free or not. If it isn’t free, it moves onto another pixel to acquire a lock. Once a lock is acquired, the thread has exclusive access to that pixel. The values from the color buffer, and G-buffer are read. Based on the G-buffer values, and the current location of the camera, the reprojected point is calculated in world space. Based on this, we derive the pixel to which it has to be reprojected to. The thread then proceeds to acquire the lock for that pixel, this time, waiting until it does so, and finally writing the pixel value to the front of the deep buffer when the lock is acquired. After this process, the thread then proceeds to free up both the locks, thereby freeing up both the source and the destination pixels.

As mentioned earlier, to calculate the luminance by using Eq. 3.1, we must store the RGB values in the deepbuffer. We use the deepbuffer to store crosshairs of samples. This use of crosshairs helps us in supporting calculating the color gradients in a frameless context. So each deepbuffer cell consists of a crosshair of samples instead of just a single sample. A crosshair consists of 6 separate samples - a central sample, two samples along the same vertical axis but different horizontal, two samples along the same vertical axis but on different verticals, and another central sample a slightly different time stamp (Figure 3.4). When we sample at location \((x, y)\), we also make sure to sample at \((x, y + 1), (x, y - 1), (x + 1, y),\) and \((x - 1, y)\). These 5 samples can be used to determine the gradients along the \(x,\) and \(y\) axes. We still need to introduce a temporal difference in our sampling mechanism to calculate the temporal gradient. To accomplish this, the pixel is then marked as being sampled at the current time step. During the next GPU dispatch, all pixels marked as such are sampled once again, ensuring that the sample is taken at the exact location within the pixel. This introduces the temporal difference we need to calculate the gradient.

We now use these crosshair samples to calculate the \(x, y,\) and \(t\) gradients (respectively \(g_x, g_y,\) and \(g_t\)) as follows:

\[
g_x = \frac{1}{2} \left[ \frac{|lum_{left} - lum_{center}|}{x_{center} - x_{left}} + \frac{|lum_{center} - lum_{left}|}{x_{right} - x_{center}} \right] \tag{3.2}
\]

\[
g_y = \frac{1}{2} \left[ \frac{|lum_{bottom} - lum_{center}|}{y_{center} - y_{bottom}} + \frac{|lum_{center} - lum_{top}|}{y_{top} - y_{center}} \right] \tag{3.3}
\]
Figure 3.4: Crosshairs, which are stored in the deep buffer, are used to calculate the gradients. We used the symmetrical crosshairs initially, but have since modified the implementation to use the tetrahedron crosshairs. Previous of adaptive frameless rendering by [10] have used the symmetrical crosshairs.
CPU

The only work that is done on the CPU is the retiling procedure. To handle the tiles, we use a quadtree datastructure. The root of the quadtree is the entire image, and the leaf tiles of the quadtree are the individual tiles that, together, span the entire image. A quadtree works well for our purpose because of its inherently recursive nature. While the previous implementation of adaptive frameless rendering by [11] and [10] used a kd-tree for handling tiling, we preferred a quadtree instead. In a quadtree, each node in one level is guaranteed to have the same $x$ and $y$ dimensions, a property that we found desirable, and absent in a kd-tree. However, it is possible that a kd-tree would fit the data much better, and we leave an implementation of adaptive frameless rendering using a kd-tree for future work. We additionally maintain a list of leaf tiles, and a list of parent tiles. A leaf tile is a tile which has no children, and a parent tile is a tile that has exactly four children, each of which is a leaf tile. We perform the retiling procedure at every CPU dispatch, however, it must be noted that performing the retiling on the GPU could be faster and would remove the dispatch overheads that we currently face. We leave a GPU implementation of retiling for future work. After mapping all the buffers that contain information about the variance of the tiles, we perform the retiling process. This process consists of two separate operations - the merge operation, and the split operation - which are described in detail next. Together, we call this one split-merge pair.

**Merge**

First, we identify the parent tile which has the least error (measured in terms of variance). Then, we remove its four children from the list of leaf tiles that we maintain. The number of leaf tiles now decreases by 4. Then, we proceed to call the `mergeTile` function which deletes from memory information about the children, and sets to null corresponding pointers. Now that the parent tile with least error has no children, it becomes a leaf tile. So, in the list of leaf tiles that we maintain, we add this tile. We also remove this tile from the parent tile list we maintain. Now, the number of leaf tiles increases by 1, and is 3 less than when we started the retiling procedure. Now, if the parent of this newly-added leaf

\[ g_t = \frac{|l_{um_{temporal}} - l_{um_{center}}|}{t_{center} - t_{temporal}} \]  

(3.4)
tile contains 4 children, all of which are leaf tiles, then we add that to the list of parent tiles that we maintain.

**Split**

The split operation can be thought of as the mirror image of the merge operation. First, we identify the leaf tile with the maximum error. Then, we call the *splitTile* function. This function create 4 new children for this tile, allocates memory for them, and also handles corresponding pointers. We now add the newly created children into the list of leaf tiles that we maintain. We also remove the tile from the list of leaf tiles, and add it to the list of parent tiles. We then remove its parent from the list of parent tiles. This results in 4 tiles being added to the leaf tiles, and one tile being removed. So, in all, the split operation results in 3 new tiles being added to the leaf tiles, thereby maintain the same number of leaf tiles.

**A note on retiling**

A couple of things have to be kept in mind when performing the retiling procedure. Recollect that the purpose of the retiling procedure is to bias the sampler towards areas of the scene with high error. However, if we are not careful, we could end up biasing the sampling towards such regions too much. This could lead to oversampling of those regions, while undersampling other regions. To avoid this, we restrict the size of the tiles that are split or merged. When choosing a candidate parent tile to be merged, we ensure that we do not select a tile that is any more than 64 px * 64 px in size. In our 512 px * 512 px scene, this corresponds to 12.5 % the size of the scene. By doing this, we ensure that no tile gets too big, and so isn’t undersampled.

It must be noted that performing only one split-merge pair per CPU iteration will lead to the tiling not adapting fast enough to the dynamic nature of the scene. However, performing multiple split-merge pairs per CPU iteration is computationally expensive, as the retiling procedure on the CPU now becomes a bottleneck. In our experience of experimenting with different number of split-merge pairs per CPU iteration, we’ve observed 5 split-merge pairs leads to desirable results. However, the exact number is heavily dependent on the nature of the dynamic motion we wish to introduce in the scene. If we know the scene is going to change very rapidly, a higher number of split-merge pairs is needed. However, if the scene only has small, incremental and slow changes, then a
low number of split-merge pairs is preferred. The number of tiles, and the number of split-merge operations to perform can be varied by the use of gain control. We leave this for future work.

3.3 Reconstruction

The original frameless rendering work in [9] simply displayed the most recent samples at any given pixel. This is a strategy referred to by [10] as “traditional reconstruction”. The result of traditional reconstruction is an image that is noisy, and has a lot of “pixie dust” (Figure 3.5). This gives the appearance of an image that sparkles and scintillates as the underlying scene changes [10].

With adaptive sampling, the image quality improves a little, as the areas that have high change such as edges and regions of motion are sampled more than static areas with low noise such as plain surfaces. This biasing of the sampling enhances the visual aesthetics of the image, reducing the amount of scintillations and noise we observe. However, even with adaptive sampling, these problems persist, and merely sampling adaptively can not overcome these visual artefacts. So, a new approach is necessary to restore image quality to acceptable levels for interactive and real time settings. This approach, called “adaptive reconstruction” (Figure 3.6) relies on a deep buffer. This deep buffer ensures that we can store information for every pixel across time as well. It must be observed that using a deep buffer to store information from old samples presents problems of its own. For example, when the scene is static, the old samples contain information that is valid and should influence the color of neighboring pixels. However, if the scene is dynamic, then the old samples contain “stale” information that should contribute minimally to the neighboring pixels, whereas newer samples must contribute heavily. Filtering in adaptive reconstruction is done across both space and time, and the deep buffer plays a key role in that process. The details of that process are described in the coming sections.

3.3.1 Mechanism

A natural question with respect to the filtering process used in adaptive reconstruction is what sort of filter to use. What should determine the shape and the size of the filter? As pointed out by [10] a filter that is spatially broad, but temporally narrow will give more weight to the spatial neighbors of the pixel. Because the importance given to old
Figure 3.5: Pixie dust as seen in traditional frameless rendering. A characteristic of frameless rendering is that the viewer can see pixel samples from multiple moments in time. This results in pixie dust as seen here. Theoretically, in traditional reconstruction, unless a pixel is sampled again, it continues to have old sample information.
Figure 3.6: Adaptive Reconstruction. At static moments, the difference between traditional and adaptive reconstruction is minute. However, once the scene becomes dynamic, then adaptive reconstruction gives a much better quality image than the traditional counterpart. This figure shows adaptive reconstruction at a dynamic moment. Notice how the pixie dusts at the trailing edge have become significantly blurred.
samples is less than that given to spatial neighbors, such a filter is well suited for scenarios when the underlying image is moving rapidly. Similarly, a temporally broad but spatially narrow filter would give more weight to older samples, and less weight to spatial neighbors. Such a filter is well suited for scenarios where the underlying image is not moving much, or is static. It is very much possible that within a given scene, certain regions exhibit a dynamic characteristic, while other regions remain static. Because of this, adaptive reconstruction should accommodate a mechanism that encompasses varying filter extents in space and time for different areas of the image.

**Filter Size and Shape**

The only filter shape we have experimented with is the Gaussian filter. It must be noted that although the Gaussian filter kernel is computationally heavy, it gives a relatively clearer image. Although [10] have experimented with other filter shapes such as the classic Mitchell-Netravali filter (two cubic polynomials defined over adjoining intervals), and the inverse exponential filter, we have not.

![Figure 3.7: A sampling density map (left) is used by the reconstructor to determine the expected local sample volume $V_s$ and the tile gradients (right) $G_x$, $G_y$, and $G_t$. Here, $G_x$, $G_y$, and $G_t$ are shown in blue, green, and red, respectively.](image)
Filter Type

The filter size is determined by the local sampling density map and the local space-time gradients (Figure 3.7). As mentioned in [11] and [10], the size of the filter support can be interpreted as a space-time volume. Given a local sampling rate $R_l$, expressed in samples per pixel second, let $V_s$ be the expected space-time volume occupied by a single sample.

$$V_s = \frac{1}{R_l} \quad (3.5)$$

The units of $V_s$ is pixel-seconds. As pixels have area, the product of pixels and seconds is a volume. The filter is then constructed at this location with space-time support proportional to $V_s$. As mentioned in [10], the filters are restricted to be axis aligned to the $x$, $y$, and $t$ dimensions for the sake of simplicity. Now, we choose filter extents $e_x$, $e_y$, and $e_t$ in such a way that they span equal areas of change of color, as determined by the estimates of $x$, $y$, and $t$ gradients. So the product of the extent with the corresponding gradient is same for all the three dimensions. [10]

$$e_x G_x = e_y G_y = e_t G_t \quad (3.6)$$

This being subject to the total volume constraint $V_s$ gives us

$$V_s = e_x e_y e_t \quad (3.7)$$

So, the filter extents are given by

$$e_x = \sqrt[3]{\frac{V_s G_y G_t}{G_x^2}}, \quad e_y = \sqrt[3]{\frac{V_s G_x G_t}{G_y^2}}, \quad e_t = \sqrt[3]{\frac{V_s G_x G_y}{G_t^2}} \quad (3.8)$$

3.3.2 Implementation

As mentioned by [11] and [10] the reconstruction process can be thought of in two different ways. A gather process loops over the pixels, searches the neighborhood of each pixel for nearby samples, and evaluates the contribution of those samples to that pixel. Reconstruction can also be implemented as a scatter process, wherein we loop through each sample, project it onto the image plane, and evaluate its contribution to all the pixels within some footprint.
Gather Process

Each GPU dispatch is technically two separate dispatches - one for sampling, and one for reconstruction, occurring one after the other sequentially. It is critical that the reconstruction happen after all the sampling is finished to ensure that the reconstructor works correctly by using the most recent information that every pixel can offer. This is possible only after the sampling process is complete.

In reconstruction as a gather process, we assigned every pixel its own thread. Each thread is responsible for reconstructing the pixel it is assigned based on the calculations that the filter shape and sizes mentioned previously dictate. First, the size of the filter support extents in space and time are calculated in terms of number of pixels in the space dimension, and number of previous samples taken in the time dimension. Based on this, the thread loops over the neighboring pixels in space by accessing the color buffer, and the old samples corresponding to them by accessing the deep buffer. While the thread loops over these samples, weights are assigned to their underlying color by the use of the Gaussian function

$$G(x, y) = \frac{1}{2\pi\sigma^2}e^{-\frac{x^2+y^2}{2\sigma^2}} \quad (3.9)$$

While multiple threads can access a pixel to read its value at the same time, this does not cause a race condition. This is because OptiX allows for multiple simultaneous reads to take place on a resource without exceptions being triggered. The final color of the pixel is then computed as the weighted sum of all the samples that we have looped over. The weights assigned are proportional to the distance from the center pixel in space and in time, as calculated by Eq. 3.9. To calculate the final color of the pixel, we take the weighted average of the colors we have gathered from.

$$color_{final} = \frac{\sum_i w_i color_i}{\sum_i w_i} \quad (3.10)$$

As the color in any given pixel is just its corresponding RGB values, this can be written as

$$r_{final} = \frac{\sum_i w_i r_i}{\sum_i w_i} \quad (3.11)$$
Scatter Process

A scatter process is theoretically an inverse of the gather process. While in a gather process each pixel has to loop over its neighbors to calculate its final color, in a scatter process, new samples scatter information to neighboring pixels. Each pixel then calculates its final color value by again taking a weighted average of the colors. The only difference is the manner in which the samples were presented to the pixel. In a gather process, the pixel loops over the neighboring samples, irrespective of whether or not they are new samples. In a scatter process, only the new samples updated the weighted sum of colors and the sum of weights for the pixel as and when they arrive. Because the number of new samples scattering to the pixel will be lesser than the total number of samples in the neighborhood of the pixel, it is conceivable that the scatter process would be faster than the gather process. However, we have not experimented with the scatter process yet, and leave it as future work for the moment.

3.4 Data Structures

Our algorithm uses multiple buffers, each serving their own purpose. Based on their purpose, buffers can be accessed by either the CPU or the GPU, or by both. Information about surface properties is maintained in a G-buffer, and the actual color of the surface is stored in a Color buffer. The color in which we render need not be the same as the color of the surface (for example, when we want to visualize rendering activity, only newly colored pixels are rendered in a particular color, but variance calculations still have to happen based on the underlying color of the surface). To accommodate this, there is also an Output buffer. This is the buffer that is used by the OptiX context to finally render the scene. There is an additional Deep buffer, that stores old pixel information to make the renderer adaptive to variations in space as well as time.

In addition to this, the tiles are stored in a quadtree datastructure, with the root tile
being the entire image. The leaf tiles are also placed in a max-heap with the tile variance being the key. Similarly, the parent tiles are placed in a min-heap with the tile variance being the key. At this point, it is worth noting that it is useful to place certain restrictions on the size of leaf tiles. If a leaf tile gets too small, then it will be oversampled, thereby resulting in an inefficient use of the GPU resources, as it is possible that the number of pixels in the leaf tiles is lesser than the number of samples we take within that tile. Similarly, if a leaf tile gets too big, then it will be undersampled. So although there may be newer samples present in that leaf tile, they will not be enough in number to increase the variance of the tile to an extent that it gets split in the retiling procedure. This problem is especially prevalent with the parts of the scene that have a constant color, such as the background.

3.5 Pseudocode

See Table 3.1.

3.6 Implementation Details

**Platform.** We generated all results on a 32-bit Windows machine with an Intel Core 2 Duo CPU running at 2.4 GHz, with a 4 GB RAM, and an Nvidia GeForce GTX 465 graphics card with 1024 MB of video memory. We coded in CUDA C and C++ using OptiX, Nvidia’s ray tracing engine [4] [18]. The GeForce GTX 465 has eleven tessellation engines, 352 CUDA cores, and 1 GB of memory [1]. We implemented the retiling procedures on the CPU and the sampling, reprojection, and reconstruction procedures on the GPU. While this hardware is old, it is a good testbed for adaptive frameless rendering, since it will more quickly reach the limits of performance with traditional framed rendering when compared with more up-to-date hardware.

**OptiX.** Nvidia’s OptiX is a general purpose ray-tracing API. The OptiX engine is composed of two symbiotic parts - the host API that defines ray-tracing based data structures, and the CUDA C based programming system that handles all rays [4]. The host API is also capable of handling collision detection and certain A.I. queries. The core of OptiX is a domain specific just in time compiler that generates custom ray tracing kernels by com-
Table 3.1: Pseudocode

<table>
<thead>
<tr>
<th>GPU</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>CreateLeafTiles ()</td>
<td></td>
</tr>
<tr>
<td>CreateParentTiles ()</td>
<td></td>
</tr>
<tr>
<td>Loop</td>
<td></td>
</tr>
<tr>
<td>For each leaf tile</td>
<td></td>
</tr>
<tr>
<td>Shoot rays at a random point within the tile</td>
<td></td>
</tr>
<tr>
<td>Using information from point, update buffers</td>
<td></td>
</tr>
<tr>
<td>End for</td>
<td></td>
</tr>
<tr>
<td>For each thread</td>
<td></td>
</tr>
<tr>
<td>Reproject</td>
<td></td>
</tr>
<tr>
<td>End for</td>
<td></td>
</tr>
<tr>
<td>Repeat 5 times</td>
<td></td>
</tr>
<tr>
<td>Split leaf tile with max. variance</td>
<td></td>
</tr>
<tr>
<td>Merge parent tile with least variance</td>
<td></td>
</tr>
<tr>
<td>End Repeat</td>
<td></td>
</tr>
<tr>
<td>End loop</td>
<td></td>
</tr>
</tbody>
</table>

Bining user supplied programs for ray generation, material shading, object intersection, and scene traversal [18]. The OptiX engine also focuses on the fundamental computations required for ray tracing and avoids embedding rendering-specific constructs. The OptiX engine presents the entire mechanism based on ray-geometry interactions. As a consequence, there are no specific built-in concepts of shadows, lights, or reflectance.

The previous implementation of Adaptive Frameless rendering by [11] and [10] relied on OpenRT to handle ray-tracing. OpenRT uses a binary plug-in interface to provide surface, light, camera, and environmental shaders [12]. However, OptiX achieves a system that strives for generality [18].

28
3.7 New Problems Encountered

Because our implementation of Adaptive Frameless Rendering is GPU based, an approach that differs from previous implementations, we also faced new challenges and problems that we must elaborate upon.

One such problem we faced was in determining and achieving an effective CPU/GPU balance. Of course, in our current implementation, the retiling procedure is done on the CPU, while sampling and reconstruction are done on the GPU. The CPU is also responsible for displaying the buffer that the GPU program returns to it using OpenGL [18]. We had initially implemented the variance calculations on the CPU, but later moved it to the GPU, thereby achieving a faster dispatch speed. We believe that a further improvement in dispatch time can be achieved if even the retiling procedure takes place on the GPU.

Another problem we encountered was in terms of the interactivity of the system. The previous adaptive frameless rendering implementation by [10] and [11] was much slower. The interactive nature of our implementation also made us face several challenges in terms of speed. One of these was how to efficiently incorporate reconstruction into the real time closed loop. While the previous implementation was an open loop system, where the sampler and reconstructor were decoupled, our system is a closed loop system (see Figure 3.1). Because reconstruction had to be present in the real time closed loop, our system also had to be structured in such a manner that no single portion of the closed loop (sampling, variance calculation, gradient calculation, reconstruction, and retiling) acted as a bottleneck.

Sampling in our system is also parallel, whereas in previous implementations it was sequential. This also throws up new challenges in term of competitions between threads. Handling multiple threads having access to the same resources was a problem not present before, and in parts of our system, like those that handle reprojection, we have made decisions based on these new problems that seem logical at this juncture. However, it must be noted that further research and a more rigorous analysis of the best approach to solve these problems is needed, and we leave such experimentation for future works.
Chapter 4

Results

Figure 4.1: Adaptive Frameless Rendering on a simple, one plane scene. The picture on the left shows the scene during a still moment, and the picture on the right shows the scene when there is motion.
Figure 4.2: Adaptive Frameless Rendering on a more complex scene. The picture on the left shows the scene at a still moment, and the picture on the right shows the scene when there is motion.

Table 4.1: Amount of time consumed in seconds by the CPU and GPU for various scenes in one iteration of the algorithm.

<table>
<thead>
<tr>
<th>Scene Type</th>
<th>GPU</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple Scene, Double Buffering</td>
<td>0.003</td>
<td>0.000</td>
</tr>
<tr>
<td>Simple Scene, AFR</td>
<td>0.052</td>
<td>0.006</td>
</tr>
<tr>
<td>Whitted Scene, Double Buffering</td>
<td>0.016</td>
<td>0.000</td>
</tr>
<tr>
<td>Whitted Scene, AFR</td>
<td>0.074</td>
<td>0.005</td>
</tr>
<tr>
<td>Complex Scene, Double Buffering</td>
<td>0.172</td>
<td>0.000</td>
</tr>
<tr>
<td>Complex Scene, AFR</td>
<td>0.124</td>
<td>0.007</td>
</tr>
</tbody>
</table>
Chapter 5

Discussion and Future Work

Given that this is the first implementation of Adaptive Frameless Rendering on the GPU, there exists a lot of potential for improvement. Although we have successfully implemented a version of Adaptive Frameless Rendering on the GPU, there are noticeable points of difference in the specifics when compared to the previous implementation by [11] and [10].

One of the differences is our use of a quadtree [20] to manage tiling, while [11] and [10] used a kd-tree [7] [13]. While a quadtree offers us numerous advantages in terms of simplicity of implementation, the use of a kd-tree is an option worth exploring. The use of a kd-tree could remove certain restrictions that are presented by a quadtree, such as the need for tiles to always be perfect squares. By removing this restriction using a kd-tree, positioning and orientating of the splitting lines would be more fine grained, and the tiling mechanism would therefore direct the sampler more precisely towards regions of high noise.

Another interesting implementation of adaptive frameless rendering would be one in which the retiling is done by the GPU. In our implementation, the retiling is done by the CPU, which then acts as a bottleneck. Letting the GPU handle retiling would come a long way in removing that bottleneck, thereby making the scene more conducive for interactive settings. To take this idea one step further, we could dispense with the CPU side of the system altogether, and make an adaptive frameless renderer that runs only on the GPU. Currently, dispatches are sent to the GPU via the CPU by the use of the `rtContextLaunch` API function that OptiX provides. This hands over control to the GPU for executing the ray tracing kernel. Once the execution is finished, the resulting
data is used by the application, and the output buffer is then displayed by the use of OpenGL [18]. An implementation of adaptive frameless rendering that is solely on the GPU without any barriers or dispatches would definitely be an area of improvement.

When the scene to be rendered is highly dynamic, the changes in the desired output image will also be more rapid. In such cases, fixed delays in response to such changes could lead to wild increases in error [10]. For example, if the retiling does not adapt fast enough to accommodate changes in regions of motion, the sampler will correspondingly lag behind. One way to address this problem is to use the control theoretic concept of gain control [15]. Using gain control to dampen or amplify the impact could lead to a better utilization of computational resources. This sort of compensation can be used to adjust the number of tiles, and also for managing the number of split-merge operations we need to perform.

As far as the reconstruction portion of adaptive frameless rendering is concerned, we have only managed to implement a gather based reconstruction (see section 3.3). A gather based reconstruction mechanism loops over every pixel in the neighborhood of the pixel that we filtering. However, a scatter based reconstruction mechanism reduces the number of primitive operations we need to perform by inverting the process. Only the newly sampled pixels need to scatter to their neighborhood, and since the number of newly sampled pixels is only a fraction of the total number of pixels, a scatter based implementation could run faster than a gather based implementation.

Implementing adaptive frameless rendering on better hardware would also be interesting. While our hardware is relatively old, it was a good testbed for adaptive frameless rendering since older hardware will quickly reach their performance limits as opposed to newer hardware. A more rigorous evaluation metric would also be an area to improve upon. While traditionally performance speed in graphics applications are measured in terms of framerate, it is meaningless to use the same standard of evaluation for a frameless setting. A study with the break-even points that describes when adaptive frameless rendering outperforms traditional framed renderers will also be helpful. A system that allows for partial image updates instead of needing a full screen update would also be of interest as it would be truly frameless. It may also be worth exploring frameless renderers and their relationship to G-Sync [2].

While we have experimented only with frameless primary rays (or eye-rays) it may be worth experimenting with frameless secondary rays (reflection, refraction, and shadow
rays) to calculate their contribution to the ray tree [23].
Chapter 6

Conclusion

We present a GPU based implementation of Adaptive Frameless Rendering, a departure from double-buffered rendering that has the potential to increase rendering speeds. Our implementation is a real time closed loop sampler reconstructor system whereas previous implementations of adaptive frameless rendering have been open loop systems that decouple the sampler and the reconstructor. The main advantages of this work are obtained by leveraging the potential of the GPU, introducing parallelism into the sampling and reconstruction process, and the real-time nature of the application. However, discussion suggests that our current implementation has not reached its full potential due to implementation specific details. While our implementation relies on the CPU to handle the retiling procedure, an implementation of adaptive frameless rendering in which the retiling procedure is done on the GPU could potentially improve upon speed, thereby making the application more interactive. Extending the context of framlessness to secondary rays could also be an interesting avenue of exploration to improve upon this work.
REFERENCES


