Development of Multifragment Rendering Methods for 3D Graphics

Grigorios Tsopouridis

Diploma Thesis

Supervisor: Ioannis Fudos

Ioannina, September 2021

ΤΜΉΜΑ ΜΗΧ. Η/Υ & ΠΛΗΡΟΦΟΡΙΚΉΣ

ΠΑΝΕΠΙΣΤΉΜΙΟ ΙΩΑΝΝΊΝΩΝ

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

UNIVERSITY OF IOANNINA

Dedication

To my grandmother Eleni and my family.

Acknowledgements

First of all, I would like to express my gratitude to my advisor Prof. Ioannis Fudos for his invaluable assistance, advice, patience and guidance throughout this thesis.

Furthermore, I would also like to thank Dr. Andreas A. Vasilakis for his valuable help and advice that was a major aid in the creation of this thesis.

Finally, I would like to thank my family for their support, love, care and for encouraging me to pursue my goals.

Abstract

Many modern applications use effects such as transparency or translucency, which are difficult to be rendered correctly. These are just some of the uses of multifragment rendering methods, which require operations on multiple fragments located at the same pixel position to be rendered correctly. Presently, several such methods have been developed that process one to multiple fragments per pixel per rendering pass to produce such multi-fragment effects. Each method differs from the other methods in terms of performance and memory consumption, resulting in different rendering quality and performance, allowing us to choose the most appropriate method for each use case. Several well-known multi-fragment rendering methods (A-Buffer, k-Buffer) and some of their variants are presented, implemented, experimented on, and the results are then briefly presented.

Furthermore, a new method, Deep Learning Variable k-Buffer, is introduced. Deep Learning Variable k-Buffer attempts to approximate the quality of the memory-intensive A-Buffer, which provides the best visual results, combined with the low memory consumption of Variable k-Buffer by using deep learning to predict an importance value for each pixel.

Finally, a comparison between the results of the proposed method and those of other already established multifragment rendering methods is provided, to evaluate and present their rendering quality and performance.

Περίληψη

Πολλές σύγχρονες εφαρμογές γραφικών χρησιμοποιούν οπτικά εφέ που απαιτούν μεθόδους που μπορούν να επεξεργαστούν πολλά θραύσματα ανά πιξελ ανά πέρασμα στη σκηνή (rendering pass) για να αποδοθούν σωστά. Ένα από αυτά τα εφέ είναι και η διαφάνεια, και συγκεκριμένα η διαφάνεια που δεν απαιτεί την ταξινόμηση της γεωμετρίας από το κοντινότερο στο μακρινότερο αντικείμενο για να αποδοθεί σωστά (Order Independent Transparency). Μέχρι σήμερα, έχουν αναπτυχθεί πολλές τέτοιες μέθοδοι οι οποίες διαφέρουν μεταξύ τους σε επίδοση και στην ποιότητα της εικόνας που αποδίδουν. Έτσι, ο χρήστης έχει την δυνατότητα να διαλέξει την κατάλληλη μέθοδο για κάθε εφαρμογή.

Στην τρέχουσα διπλωματική εργασία, υλοποιούνται μερικές γνωστές μεθόδοι πολυθραυσματικής απόδοσης (A-Buffer, k-Buffer) όπως και μερικές παραλλαγές τους, και παρουσιάζονται οι μεθόδοι όπως και μερικά πειράματα πάνω σε αυτές.
Επιπλέον, προτείνεται μία νέα μέθοδος πολυθραυσματικής απόδοσης που χρησιμοποιεί βαθιά νευρωνικά δίκτυα, Deep Learning Variable k-Buffer. Η μέθοδος αυτή, χρησιμοποιεί την αρχιτεκτονική του Variable-k Buffer, που επιτρέπει διαφορετικό K ανά πιξελ, όπως και χαμηλή κατανάλωσης μνήμης. Η μέθοδος έχει ως στόχο να να προσεγγίσει την βέλτιση ποιότητα εικόνας, του ακριβού σε μνήμη, A-Buffer με μειωμένες απαιτήσεις μνήμης.

Τέλος, γίνεται σύγκριση και παρουσιάση των αποτελέσματων της παραπάνω μεθόδου με τα αποτελέσματα άλλων ήδη υπάρχοντων μεθόδων, όπως και αναλύση της ποιότητας και της επίδοσής της.

List of Figures

List of Algorithms

Table of Contents

Chapter 1

Introduction

Rendering is the process of generating an image from 2D or 3D geometry using a computer program. The resulting image is referred to as the render. From the above, we can easily infer that rendering is a vital part not only of computer science but our daily lives. A plethora of modern applications such as computer games, movies, TV visual effects, simulations are only some of the examples that require rendering and computer graphics in our everyday lives.

Multifragment rendering is the process of generating an image from 3D geometry through sampling multiple times in the same pixel (fragments) in a computer program. Multifragment rendering methods can store almost all fragment data for each pixel and thus exploit every detail of the geometry. This allows the creation of applications, ranging from order-independent transparency, constructive solid geometry, shadows to global illumination.

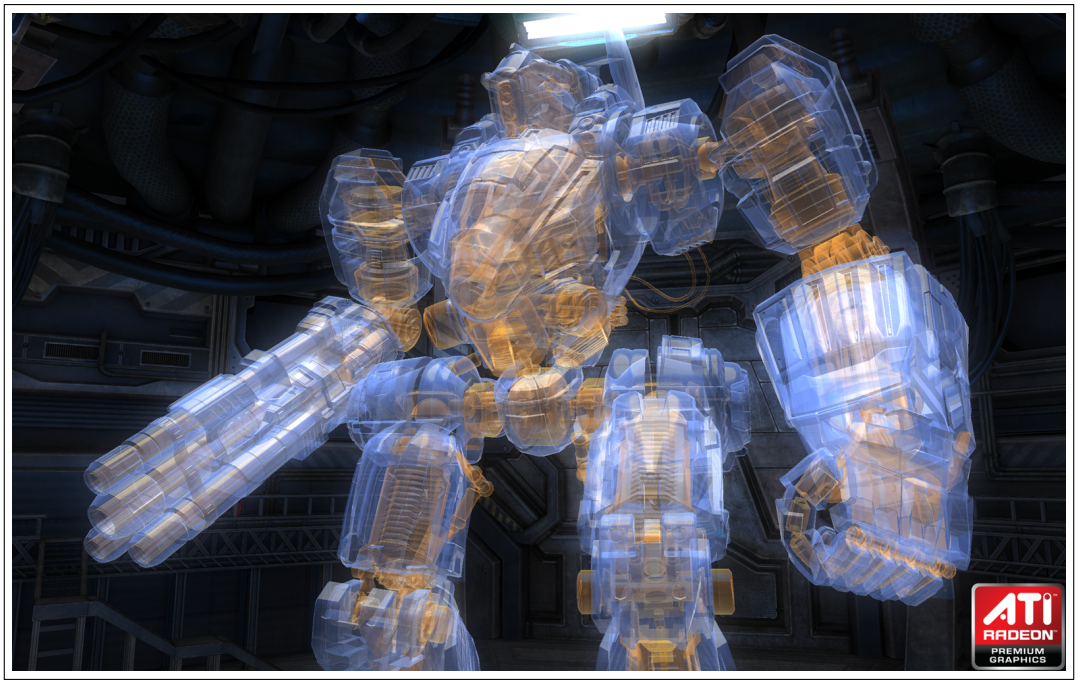

Figure 1.1: Order-Independent Transparency. Image from ATI Mecha Demo [ATI].

Today, the accessibility of this technology has improved due to advances in both hardware and software. The hardware is getting faster and cheaper, while the software and tools needed to perform the rendering process are getting better and more widely available. As a product, there are currently numerous renderers available. This makes it relatively easy for a company of almost any size to develop a product. Some renderers are integrated into larger modeling and animation packages, others are stand-alone, and many are free open-source projects. Rendering is usually associated with many different subtopics of computer graphics that are often required for an organization to develop a product, but this is not always the case. For example, a movie studio may need rendering programmers, animators, and 3D modelers, professions that belong to a broad spectrum of disciplines, from artistic to technical.

Multifragment rendering methods have been successfully used in various real-world applications. The original A-Buffer was incorporated into the REYES 3D rendering system at Lucasfilm and was used in the "Genesis Demo" sequence in Star Trek II [Car84].

In this thesis, several well-known multi-fragment rendering methods (A-Buffer, k-Buffer) and some of their variants (S-Buffer, Variable k-Buffer) are implemented, studied and tested. Each method provides different rendering performance and quality. A-buffer has the best image quality and having high memory requirements, while k-Buffer and its variants have lower memory requirements but poorer performance and quality. In addition, a simple multifragment rendering (MFR) demo application is briefly presented, with its features including the ability are to switch MFR methods on the fly, and a visualized visual difference comparison of the methods using a heatmap.

Furthermore, a new method, Deep Learning Variable k-Buffer, is introduced, and compared with other well-established multifragment rendering algorithms. Deep Learning Variable k-Buffer attempts to approximate the quality of the memory-intensive A-Buffer, which provides the best visual results, combined with the low memory consumption of Variable k-Buffer by using deep-learning to predict the optimal k for each pixel.

1.1 Related Work

There is an abundance of research work in the literature that addresses the problem of multifragment rendering, which describes the multiple sampling of a pixel and the corresponding data structures.

A very comprehensive survey [VVP20] on this topic was recently published by Vasilakis et al., which presents a plethora of multifragment rendering methods, their complexity, strengths and weaknesses, and their applications.

Primary Visibility Determination and the Z-buffer. The most well-known method for primary visibility determination is the Z- buffer [Cat78]. Independently of Ed Catmull's work, Wolfgang Straßer's dissertation described the Z-buffer algorithm as early as 1974. Depth buffers aid the rendering of a scene as they ensure that polygons are properly occluded and that the depth values closest to the camera are stored. In the Z-buffer, the depth of each fragment is compared

to the already stored depth. If it is closer than it to the camera, it replaces the stored value and updates the buffer with its depth and other information as the new closest fragment. After processing all the fragments, the Z-buffer algorithm consists of only a single layer containing the closest visible fragment per pixel. However, some techniques such as Order-Independent Transparency require more than one layer of fragment information per pixel in order to be rendered correctly.

Depth Peeling. Depth Peeling [Mam89] is an iterative multifragment rendering method that can store multiple layers of pixel information. Depth peeling renders the geometry multiple times, extracting a single fragment layer per rendering pass. It uses two Z-buffers, one that works normally, and one that sets the minimum acceptable depth at which a fragment can be rendered: the previous pass's normal Z-buffers depth. Each pass draws what is further (behind) than the previous pass. The algorithm stops either when there are no more fragments or when the maximum number of layers specified by the user has been processed. The many geometry rendering passes make it unsuitable for use in real-time applications. Depth Peeling was later implemented by Everitt [Eve01].

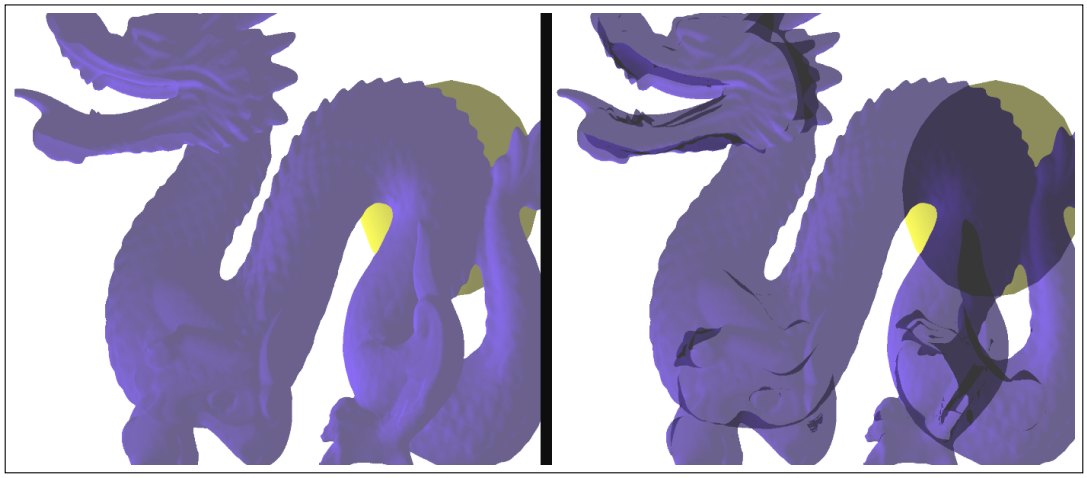

Figure 1.2: Order-Independent Transparency using Depth Peeling, 2 layers (left), 3 layers (right).

A-Buffer. Carpenter's original A-Buffer [Car84], a descendant of the well-known Z-buffer, is a popular method that captures multiple fragment layers per-pixel in multiple per-pixel linked lists. The fragments are first captured and then sorted by depth. This enables the rendering of Order-Independent Transparency and works as an antialiasing mechanism at the same time. Later, Yang et al. [YHGT10], introduce an A-Buffer variant that works on the GPU and uses a per-pixel linked list by exploiting GPU atomic operations. The process of capturing all fragments is as follows. The A-Buffer is first cleared, then all fragments are captured in per-pixel linked lists in a geometry rendering pass and finally, a post-process operation depth-sorts the linked lists, operates on the sorted fragments, and renders the result to the framebuffer. Proper memory cache utilization can also have a large impact on the overall performance. Crassin [Cra10] implements an A-Buffer variant that uses fixed-size arrays to store all fragment information, this has greater performance than per-pixel linked lists but can waste large amounts

of memory due to low occupancy. A CUDA-based implementation of fixed-sized array A-Buffer is proposed by Liu et al. [LHLW10], that allows the storage of all the fragments in fixed-sized per-pixel arrays. Variable-sized array approaches try to combine the advantages of both the linked list and fixed-sized array approaches. Vasilakis and Fudos [VF12], propose S-buffer, an A-Buffer variant, that requires two additional passes before the capturing of fragments. A geometry pass that counts the number of fragments per-pixel and, a screen-space pass that generates the per-pixel memory offsets by using a parallel prefix sum. This method allocates memory in contiguous regions and thus offering improved cache coherence and exact memory allocation.

k-Buffer. k-Buffer reduces the memory consumption by only capturing the k-best fragments per pixel, usually the k-closest to the camera, of all generated fragments in a scene in a single rasterization pass. It assumes a preassigned global k value across every pixel. k-Buffer was initially proposed by Callahan et al. [CICS05] for volumetric rendering and was later extended [BCL+07] for general use. k-Buffer suffers from flickering artifacts due to read-modify-write data hazards that arise when the generated fragments are simultaneously inserted into the buffer in arbitrary depth order. Salvi utilizes hardware-accelerated pixel synchronization [Sal13] to solve this problem but, it could potentially reduce its performance due to simultaneous contention of accessing the critical areas. Finding the optimal k for a scene is a very challenging task and usually requires an iterative approach that consumes a lot of development time. To solve this problem, a dynamic k-Buffer [VPF15] was introduced, able to automatically determine a global k value under memory constraints. Unlike previous approaches, variable k-buffer [VVPM17] introduced by Vasilakis et al., assigns a different k value for each pixel, which is calculated using an importance function. Their proposed importance function is computed by using the Depth Complexity, Fresnel, and Foveated (or periphery) heuristics.

1.2 Thesis Structure

The remainder of this thesis is structured as follows. Chapter 2, titled Background, covers the basics of Rendering, Multifragment rendering, Shader programming, Order-Independent Transparency, and introduces various tools and technologies used in this thesis. Chapter 3 presents the shader implementation, brief experiments, and performance of some of the most popular multifragment rendering methods. Chapter 4 introduces a new multifragment rendering method, Deep Learning Variable K-buffer. Deep Learning Variable K-buffer takes advantage of the Variable k-Buffer architecture and attempts to approximate the results of the A-Buffer using a simple neural network. A-Buffer provides the best rendering quality as it can store all the scene's fragments. A step-by-step process of creating the neural network, generating and processing the feature data and training set, and finally some of the results obtained with this method are presented. Chapter 5 presents an experimental comparative evaluation between the proposed method and some of the already introduced multifragment rendering methods. Finally, Chapter 6 concludes our work and proposes directions for future work based on this thesis.

Chapter 2

Background

In this chapter, some basic, required theoretical background is introduced. First, rendering is defined. Then the process of rasterization, the definition of the pixel, and the fragment, and fragment processing are presented. Finally, multifragment rendering, Order-Independent Transparency as well as the technology used in this thesis are introduced.

2.1 Rendering

Rendering is the process of creating a photorealistic or non-photorealistic image from a 2D or 3D geometry (scene) using a computer program. The resulting image is called render. A scene can contain multiple models, textures, lighting, shading, and information that describe a virtual space in general. Rendering is one of the most important subtopics of computer graphics, and in practice, it is always connected to the others, as it is always the last step before the visual result.

Rendering can be done in real-time or as pre-rendering (or offline rendering). Offline rendering is used to create realistic images and movies where each frame can take several hours or days to complete, while real-time rendering is often used for video games, visualization, and other interactive applications.

2.2 Rasterization

Raster image (or raster) is a matrix data structure representing an image, usually a rectangular grid of pixels, containing color and tone information that is synthesized to create the raster image.

Rasterization is the process of converting a scene described by shapes into a raster image. It loops through each primitive (polygon or triangle), projects its vertices onto the screen, determines which pixels of the image are affected, and modifies those pixels accordingly. Thus, it attempts to solve the problem of visibility. Fragments are generated as a result of rasterizing a primitive.

Rasterization is the most commonly used technique for rendering 3D models because it offers extremely fast performance compared to its alternatives, such as ray tracing. The color of each pixel is computed by shading, which can either be based on physical laws or be purely artistic.

Figure 2.1: Raster image with its pixel grid (grey and white) visible.

2.3 Pixel and Fragment

The pixel is the smallest element of a raster image. Each pixel is a sample of an original image grid and it contains its color information. The more pixels in an image (resolution), the better its perceived quality.

A fragment is the data required to create a single pixel's worth of a drawing primitive in the framebuffer. A fragment may contain data such as depth, position, color attributes, texture coordinates, alpha. When a scene is drawn, primitives are rasterized into fragments which are then shaded and combined with the existing framebuffer, creating the pixels. A pixel can be composed of multiple fragments.

2.4 Fragment Processing

Fragment processing [Khr] is a set of operations performed on fragments generated during the rasterization process. These operations include alpha testing, depth testing, stencil testing, blending, and provide numerous mechanisms for changing or discarding fragment values. Each fragments data is processed by a fragment shader. The depth test which usually is the Z-Buffer test, only stores the data of one fragment (depending on the function used). This prevents the exploitation of every detail of the geometry as it stores only the fragments closest to the camera, thus many techniques like Order-Independent Transparency are impossible to render correctly. Due to the fragment shaders executed in parallel, correct fragment ordering is not guaranteed.

2.5 Multifragment Rendering

Multifragment rendering is the process of generating an image from 3D geometry by multiple sampling of the same pixel (fragments) in a computer program. Multifragment rendering methods can store multiple fragments per pixel and thus exploit up to every detail of the geometry.

Multifragment methods include the algorithms and data structures necessary to create, maintain, process, and utilize a set of fragments associated with a single image.

2.6 Order-Independent Transparency

Order-Independent Transparency (OIT) is a computer graphics (rasterization) technique for rendering transparency in a 3D scene, where geometry does not need to be rendered in sorted order for alpha compositing.

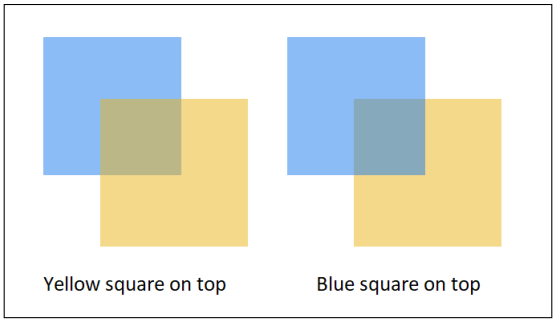

Figure 2.2: Squares drawn with 0.50 alpha value, demonstrating why fragment order matters for transparency.

Transparent 3D geometry is usually rendered by blending all surfaces into a single buffer (using alpha compositing) as each surface adds some of its color, depending on its alpha, to the existing stored color value. In short, the order in which the surfaces are blended affects the final result (Figure 2.2).

There are exact and approximate OIT methods. In exact OIT methods, transparency (the final color) is calculated accurately, so all fragments must be depth sorted. Approximate OIT techniques do not calculate an accurate result but provide faster performance.

Figure 2.3: Alpha Blending (left) compared to OIT (right). Image from ATI Mecha Demo [ATI].

As illustrated in Figure 2.3, OIT provides a result with exact transparency that can especially be perceived when comparing the Mecha's arms. On the other hand, Alpha Blending provides a faster but not accurate result with a lot of loss of detail.

2.7 Technologies Used

In this section, technologies used throughout this thesis will be briefly presented.

2.7.1 OpenGL

The programming language, the supporting libraries, and the graphics API for the development of the rendering engine which enabled us to render and experiment was a vital part of this thesis.

For this thesis, **OpenGL** API for GPU accelerated rendering tied to the **C++** programming language was used. OpenGL is used in all kinds of Computer Graphics applications, ranging from standard image rendering to 3D animation, CAD, simulations and Computer Games.

OpenGL Extension Wrangler Library (**GLEW**) was used for OpenGL extension loading, **GLFW** (Graphics Library Framework) was used for the creation of OpenGL context, windows and input. **GLM** (OpenGL Mathematics) was used as a mathematics library for graphics software and **ASSIMP** (The Open-Asset-Importer-Lib) was used for model loading. **stb** (in particular *stb_image*) was used for image loading from file and finally, **DearImGUI** was used for simple user interface.

2.7.2 Shaders

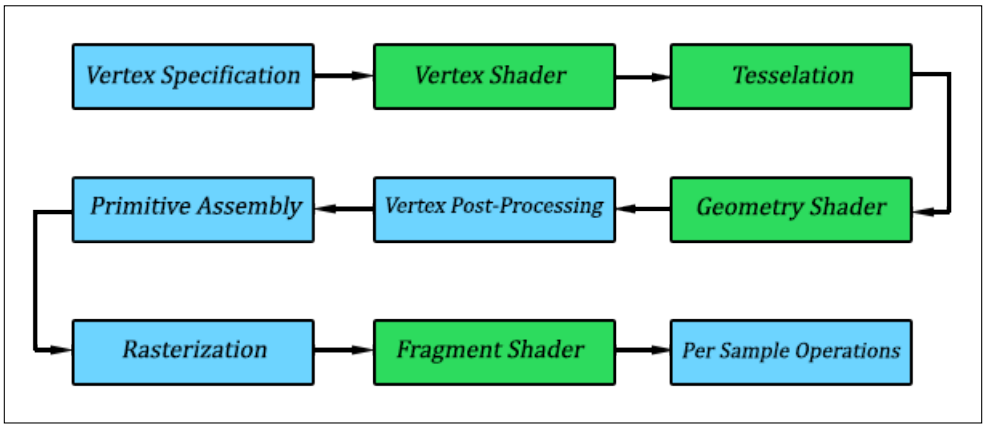

Figure 2.4: Simple diagram of the rendering pipeline. Green Boxes indicate a programmable stage.

Shaders are a type of computer program, initially used only for shading in computer graphics. They can now perform numerous functions in various sub-fields of computer graphics, or even perform functions unrelated to them. Shaders are usually run on the GPU in parallel and are divided into different types, depending on the work they perform. The types of shaders used in this thesis were the Fragment and Vertex Shaders, apart from which there are Geometry Shaders, Tesselation Shaders, Compute Shaders, and Primitive and Mesh Shaders.

The Rendering Pipeline is the sequence of steps that a renderer takes when rendering objects, some steps being programmable while others having a fixed function. Figure 2.4 displays a high-level description of the steps in the rendering pipeline. Shading Languages, such as GLSL, are used to program shaders, and thus the rendering pipeline [Khr]. Shaders replace a section of the graphics hardware called the Fixed Function Pipeline. Fixed Function Pipeline only allowed hard-coded shading and geometry transformations. Shaders provide a programmable alternative to this hard-coded approach that allows for many new customized effects.

Vertex Shaders

Vertex Shaders are called once for each vertex of the scene and have the ability to manipulate their attributes such as position, color, texture coordinates. Their purpose is to transform each vertex's 3D position in virtual space to the 2D screen coordinates as well as their depth value, used for visibility determination. Vertex Shaders enable the control over the details of each model such as position, movement, and color in any scene.

Fragment Shaders

Fragment Shaders, also known as pixel shaders, compute the color and attributes of each fragment generated during the rasterization stage. They at most affect the output of a single pixel as they only work with fragments and have no knowledge of the scene geometry. Lighting algorithms, bump mapping, shadows, and other such effects are implemented using Fragment Shaders.

Additionally, Fragment Shaders can alter and store the depth of a fragment, used for Z-Buffering, or they can store and process the data of multiple fragments, used for multifragment rendering. If the contents of a scene are passed to the Fragment Shaders through a texture, they can sample any other pixel and generate post-processing effects such as edge detection and blur.

2.7.3 Python and TensorFlow

Python is an interpreted high-level general-purpose programming language. In this thesis, Python was used as a supporting tool for the creation of Deep Learning Variable K-Buffer.

Fragments and scene data were extracted from the OpenGL renderer that was developed. Then, the data was processed with Python and was used to create a Training and Feature set for the neural network. Python scripts were created for the calculation of the Optimal-K, used as a label (or target) for the neural network. Finally, TensorFlow and Python were used to create and train the neural network.

TensorFlow is a free and open-source software library for machine learning, its Python version was used for the creation and training of a neural network.

Image quality differences are calculated with Python, using root mean squared error and the image difference evaluator ꟻLIP [ANA+20]. ꟻLIP is an image quality metric developed at NVIDIA that aims to achieve a per-pixel error map that is perceptually correct. The produced error map approximates the difference perceived by humans when alternating between two images. In this thesis, ꟻLIP mean value is used.

Chapter 3

MFR Algorithms

In this chapter, the multifragment rendering methods that are implemented in thesis are presented. A-Buffer fixed-sized array, linked list and S-buffer as well as fixed-K k-Buffer and variable-K k-buffer and a MFR algorithms demo application are presented. Every algorithm described below is used in a Fragment Shader.

3.1 A-Buffer

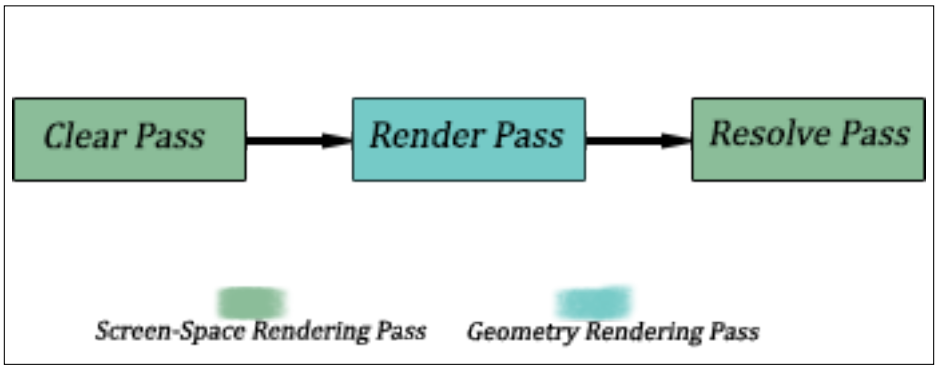

Figure 3.1: A-Buffer's rendering pipeline

A-buffer methods aim to capture all the generated fragments per-pixel. Usually, the A-Buffer stores fragment data in GPU-accelerated data structures of fixed or variable size per-pixel at a single geometry pass followed by a screen-space rendering pass (post-processing) that sorts the fragments by depth and resolves pixel colors. A-Buffer methods require a large or even an unpredictable amount of memory, as they aim to capture all generated fragments. Additionally, the depth-sorting operation may slow down performance when large amounts of fragments are captured. In general, the A-Buffer methods have a "clear" pass, a "fragment capturing" (or render) pass, and finally a "sort and resolve" (or resolve) pass.

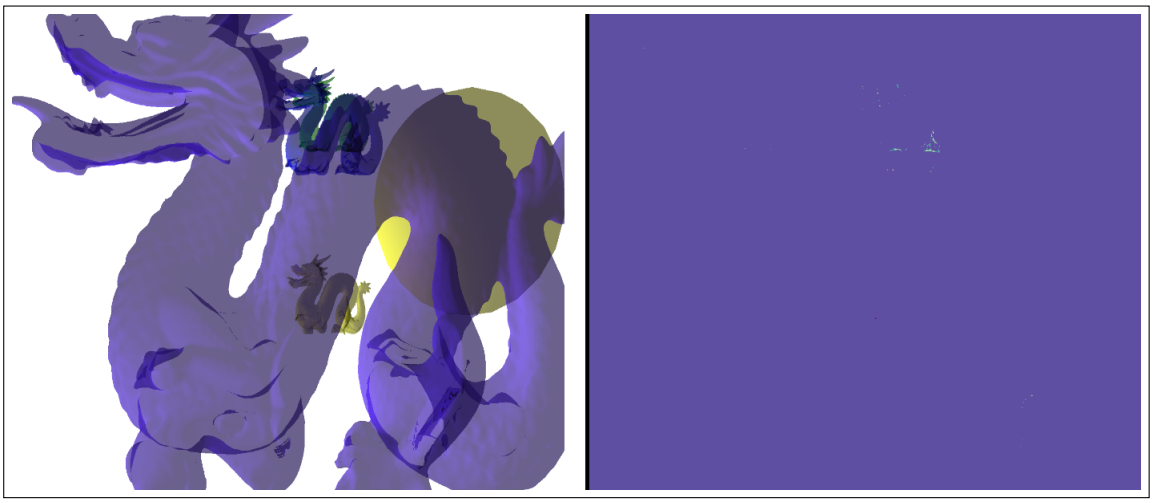

Figure 3.2: OIT rendered using A-Buffer (left). Color mapped visual difference between Fixed-Sized array (A=8) and linked-list methods (right) for the same scene.

Table 3.1: A-Buffer Linked List and Fixed-sized array performance

2500880 total scene fragments	**Fixed-Sized (A = 8)**	**Linked List**
Memory Consumption (MB)	87.8592	30.01056
Clean Stage (ms)	0.028832	0.028960
Fragment Capturing Stage (ms)	2.238624	2.157152
Resolve Stage (ms)	0.475968	0.739200
Total Time (ms)	2.743424	2.925312

Table 3.1, presents an overview of A-Buffer's Fixed-Sized array and Linked-List variants performance based on Figure 3.2's scene. The A-Buffer fixed-sized arrays A value is set high enough to compare both methods with almost no visual difference, as indicated in the visual difference colormap, and is obtained through experimentation. Fixed-Sized array offers a 6.41% better performance but requires almost three times linked-list variants memory, due to low array occupancy and uneven fragment distribution, for the same scene. Please note, that this is a scene with a low fragment count and performance differs for different scenes. For our implementation, the fixed-sized array method uses OpenGL's *RG32F texture* (8 bytes per array location) to store color and depth values while linked-list uses a *Shader Storage Buffer* of 12 bytes per fragment.

3.1.1 Fixed-Sized Array

A-Buffer fixed-sized array approaches [LHLW10] [Cra10] manage GPU memory by allocating a fixed-sized array of length A per-pixel. While fixed-sized approaches can offer better performance in complex environments they have a much larger memory consumption compared to linked-list approaches as scenes with uneven fragment distribution can cause memory waste due to low array occupancy.

A fixed-sized array with the size A equals the maximum depth complexity for all pixels can capture every generated fragment. However, any number $\boldsymbol{A}$ smaller

than that can capture only up to $\boldsymbol{A}$ fragments per-pixel, offering a lower memory consumption, better performance but a less accurate result. As fragment order is not guaranteed, a $\boldsymbol{A}$ value smaller than the maximum depth complexity for all pixels may lead to flickering artifacts.

Fixed-sized arrays require two screen-space rendering passes and a geometry rendering pass. A per-pixel unsigned-int fragment counter texture, ***counterTexture***, and a per-pixel texture for capturing fragment information (A-Buffer texture), ***bufferTexture***, are required. The general idea is to first capture the fragments and later sort and calculate pixel colors.

First, the counter texture is cleared for every pixel at a screen-space rendering pass ("clear" pass, Algorithm 1).

Algorithm 1 Fixed-sized A-Buffer clear pass

1: $counterTexture[pixel_x][pixel_y] \leftarrow 0$
▷ Reset pixel counter

Then, up to $\boldsymbol{A}$, where A is the maximum number of fragments captured per-pixel, fragments are stored into the A-Buffer texture in a geometry rendering pass ("fragment capturing" or "render" pass, Algorithm 2).

Algorithm 2 Fixed-sized A-Buffer fragment capturing pass

1: $currentCounter \leftarrow counterTexture[fragment_x][fragment_y]$
▷ Store previous counter value
2: $counterTexture[fragment_x][fragment_y] \leftarrow currentCounter + 1$
▷ Increase the fragment counter using atomic addition
3: $fragmentColor \leftarrow calculateColor()$
▷ Calculate fragments color according to a shading algorithm
4: $bufferTexture[fragment_x][fragment_y][currentCounter] \leftarrow (fragmentColor, fragmentDepth)$
▷ Store fragment information into the A-Buffer texture

Finally, the fragments are sorted by depth, and their colors are resolved in the final screen-space rendering pass ("sort and resolve" pass, Algorithm 3). Each pass is executed in a fragment shader.

Algorithm 3 Fixed-sized A-Buffer resolve pass

1: $pixelFrags \leftarrow counterTexture[pixel_x][pixel_y]$
▷ Get total fragment count of this pixel
2: **if** $pixelFrags$ is zero **then**
3: discard fragment
4: **end if**
5: **for** i=0; i<pixelFrags; i++ **do**
6: $fragments[i] \leftarrow bufferTexture[pixel_x][pixel_y][i]$
7: **end for**
▷ Store fragments into local memory
8: sort(fragments)
9: $resolvedColor \leftarrow resolveColor(fragments)$
▷ Resolve pixel color using the sorted fragments
10: $outputColor \leftarrow resolvedColor$
▷ Return pixel color

3.1.2 Linked List

A-Buffer linked list approaches [YHGT10] use per-pixel linked lists to capture all generated fragments, by exploiting GPU atomic operations. Linked list approaches offer a lower memory consumption compared to fixed-array approaches.

Like the fixed-array approaches, linked list methods run at every frame and require three passes, two screen-space rendering passes, and a geometry rendering pass. An atomic counter containing the next available location in the global buffer, ***nextAddr***, a texture for storing each pixels head, ***pixelHeads***, and a Shader Storage Buffer, ***nodes***, containing all fragment connectivity and data are required.

In the first screen-space pass (Algorithm 3), the buffer and texture are cleared to zero, representing no fragments stored.

Algorithm 4 Linked linked A-Buffer clear pass

1: $pixelHeads[pixel_x][pixel_y] \leftarrow -1$
2: $nextAddr \leftarrow -1$
3: **clear** nodes

During the second stage (geometry pass, Algorithm 5), the lists are filled. For each fragment, the address of a new node is computed by adding one to the global address counter, using atomic operations. Then, an atomic exchange sets the new node's next pointer to the pointer currently stored in the head buffer and replaces the head pointer with the new node's address. In this stage, the fragments are stored in an unsorted manner and the linked lists are created in reverse.

Finally, the fragments are sorted by depth, their colors are resolved in the final screen-space rendering pass ("sort and resolve" pass, Algorithm 6) and their result is displayed at the framebuffer.

Algorithm 5 Linked List A-Buffer fragment capturing pass

1: **atomic addition** of nextAddr with 1 and store the previous value to localNextAddr ▷ Increase the address counter using atomic addition
2: **if** localNextAddr<length of nodes buffer **then** ▷ Check memory overflow
3: $fragmentColor \leftarrow calculateColor()$ ▷ Calculate fragments color according to a shading algorithm
4: **atomic exchange** $pixelHeads[fragment_x][fragment_y]$ with localNextAddr and store the previous value to prevAddr
5: $nodes[localNextAddr] \leftarrow (fragmentColor, fragmentDepth)$
6: $nodes[localNextAddr].next \leftarrow prevAddr$ ▷ Store fragment and connectivity data
7: **end if**

Algorithm 6 Linked list A-Buffer resolve pass

1: $index \leftarrow pixelHeads[pixel_x][pixel_y]$ ▷ Check if pixel contains fragments
2: **if** *index* is zero **then**
3: discard fragment
4: **end if**
5: $count \leftarrow 0$
6: **while** $index \neq 0$ **do**
7: $fragments[count] \leftarrow nodes[index]$
8: $count \leftarrow count + 1$
9: $index \leftarrow nodes[index].next$
10: **end while**
▷ Fetch fragment data from the linked list and store fragments into local memory
11: sort(fragments)
12: $outputColor \leftarrow resolveColor(fragments)$

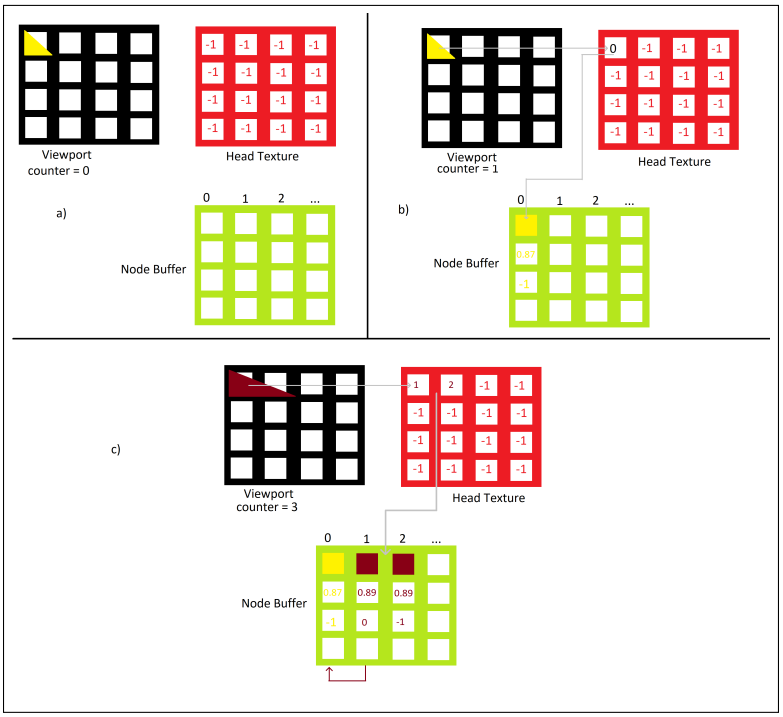

Figure 3.3: a) Yellow triangle is rasterized but not yet stored in linked lists. b) Yellow triangle linked lists are created. c) Brown triangle linked lists are created, with 1 overlapping fragment location.

3.1.3 S-Buffer

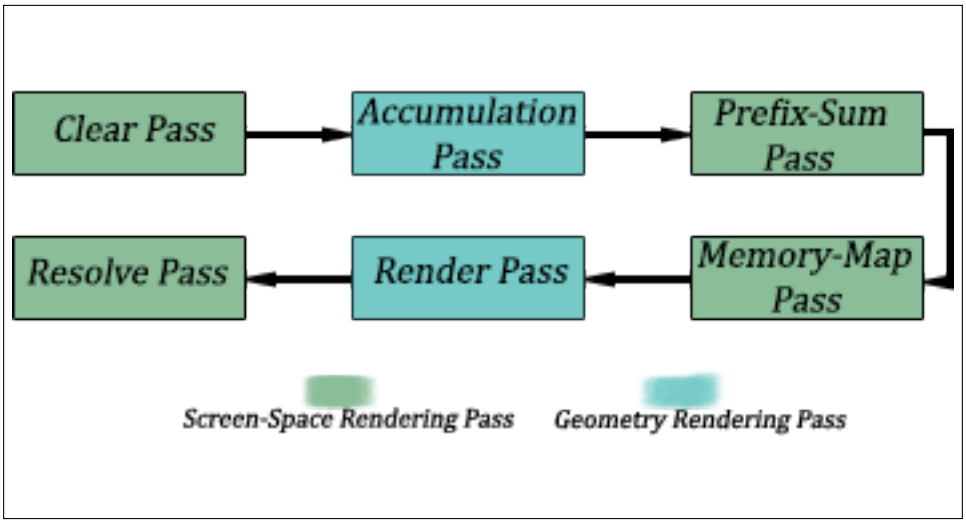

Figure 3.4: S-Buffer's rendering pipeline

S-buffer [VF12], is a memory-friendly GPU-accelerated A-buffer variant. S-buffer's memory is organized into contiguous regions for each pixel, thus avoiding the limitations set of linked-lists and fixed-array techniques. It exploits fragment distribution and pixel sparsity thus allowing for precise memory allocation and the

computation the memory offsets for each pixel in parallel. To reduce the congestion created from all pixels trying to update the same memory location (group) at the same time, S-buffer applies S user-defined multiple GPU-accelerated shared counters C, thus non-empty pixels are decomposed into multiple non-uniform groups by using a simple hash function. Each counter is associated with one pixel group and parallel prefix sums can be performed for all groups. It requires four screen-space rendering passes ("clear","prefix sum", "memory map","resolve and sort pass") and, two geometry rendering passes ("fragment accumulation" and "fragment capturing" passes"). S-Buffer performs worse than other A-Buffer variants due to the need for an extra geometry pass.

S-buffer requires a texture for counting fragments per-pixel, ***counterTexture***, a texture for pixel head address, ***addressTexture***, a Shader Storage Buffer, ***data***, for storing fragment information and a Shader Storage Buffer, ***addressMap***, for storing group addresses. ***groupNum***, is the user-specified number of group counters C.

In the first pass, a screen-space pass, the S-buffer is cleared ("clear" pass, Algorithm 4).

Algorithm 7 S-buffer clear pass

1: $counterTexture[pixel_x][pixel_y] \leftarrow 0$
2: **for** i=0;i<groupNum;i++ **do**
3: $\quad addressMap[i] \leftarrow 0$
4: **end for**

▷ Reset all groups

In the second pass, a geometry rendering pass, the accumulation of the fragments which influence a pixel is computed ("accummulation" pass, Algorithm 8).

Algorithm 8 S-buffer accummulation pass

1: **atomic addition** of $counterTexture[pixel_x][pixel_y]$ with 1

The third pass, a screen-space pass, aims to pack same pixel fragments in adjacent memory positions by computing a prefix sum of counters, per-pixel group, and then setting the pixels head address. Each pixel's group is calculated using a simple hash function:

$$H(P) = (P.x + P.y * width)\%S$$

With S being the number of counters and P.x, P.y the position of the pixel (Algorithm 9).

The fourth pass, a screen-space pass, aims to map each group of pixels to its own memory space by performing a prefix sum (forward or inverse) on the final values of the shared counters (Algorithm 10).

$$C_{pr}(i) = \sum_{n=0}^{i-1} C_{pr}(n)$$

Where $C_{pr}(i)$ is the i-th groups prefix sum.

Algorithm 9 S-buffer prefix sum pass

1: $fragNum \leftarrow counterTexture[pixel_x][pixel_y]$
▷ Check if pixel contains fragments
2: **if** $fragNum$ is zero **then**
3: discard fragment
4: **end if**
5: $pixelGroup \leftarrow H(pixel_x, pixel_y)$
▷ Calculate pixel group using the hash function.
6: **atomic addition** to $addressMap[pixelGroup]$ with fragNum and store its previous value to $pxAddress$
7: $addressTexture[pixel_x][pixel_y] \leftarrow pxAddress$
▷ Set head address for this pixel

An inverse mapping technique, that was not implemented, can be applied to boost by a factor of two the latter prefix sum process, by using total fragment information. The counters are split into two halves and forward prefix sum is applied to the first half (C_1) and inverse prefix sum for the latter half (C_2). The inverse prefix sum starts accumulating from the end of the processing set towards the start. The memory offset for each pixel P is computed using the following equation:

$$Offset(P) = \begin{cases} address(P) + C_{pr}(H(P)) & if P \in C_1 \\ totalFragments - 1 - address(P) + C_{pr}(H(P)) & otherwise \end{cases}$$

Algorithm 10 S-buffer memory map pass

1: $pixelGroup \leftarrow H(pixel_x, pixel_y)$
▷ Calculate pixel group using the hash function.
2: **if** pixelGroup <groupNum **then**
3: **for** i=0;i< pixelGroup; i++ **do**
4: $sum \leftarrow sum + addressMap[i]$
5: **end for**
▷ Only forward prefix sum shown here as inverse prefix sum was not implemented.
6: $addressMap[pixelGroup] \leftarrow sum$
▷ Store the calculated group prefix sum.
7: **end if**

The fifth pass, a geometry rendering pass, captures all generated fragments by calculating each fragments position in the global memory buffer using the pixel offset and storing it in its pixel's contiguous memory area ("fragment capturing" or "render" pass, Algorithm 11).

The final pass, a screen-space rendering pass, sorts all stored fragments by depth and resolves pixel color ("sort and resolve" pass, Algorithm 12).

Algorithm 11 S-buffer fragment capturing pass

1: $pixelGroup \leftarrow H(pixel_x, pixel_y)$
2: **atomic addition** of $addressTexture[fragment_x][fragment_y]$ with 1 and storage of its previous value into fragmentAddress
▷ Add 1 in order to have the address of the next fragment of the pixel next time.
3: $memoryOffset \leftarrow fragmentAddress + addressMap[pixelGroup]$
▷ Inverse prefix sum could be used here
4: $fragmentColor \leftarrow calculateColor()$
▷ Calculate fragments color according to a shading algorithm
5: $data[memoryOffset] \leftarrow (fragmentColor, fragmentDepth)$
▷ Store the fragment information into the appropriate position of the global data buffer

Algorithm 12 S-buffer resolve pass

1: $pixelFrags \leftarrow counterTexture[pixel_x][pixel_y]$
▷ Get total fragment count of this pixel
2: **if** $pixelFrags$ is zero **then**
3: discard fragment
4: **end if**
5: $pixelGroup \leftarrow H(pixel_x, pixel_y)$
6: $pixelAddress \leftarrow addressTexture[pixel_x][pixel_y]$
7: $memoryOffset \leftarrow fragmentAddress + addressMap[pixelGroup]$
▷ Inverse prefix sum could be used here
8: **for** i=0; i<pixelFrags; i++ **do**
9: $fragments[i] \leftarrow data[sum - i]$ ▷ As every pixels head pointer was set at their last fragment position in the previous pass, we go backwards one-by-one, from the last fragment to the first pixel fragment, in order to access all fragments and store them locally
10: **end for**
▷ Store fragments into local memory
11: sort(fragments)
12: $resolvedColor \leftarrow resolveColor(fragments)$
▷ Resolve pixel color using the sorted fragments
13: $outputColor \leftarrow resolvedColor$
▷ Return pixel color

3.2 K-Buffer

K-Buffer aims to capture the k best fragments per-pixel. The k-best fragments in a pixel are defined as the k-subset of a pixel's fragments that minimize the value of a specific cost function. For example, the k best fragments could be defined as the k fragments closest to the camera. k-Buffer reduces the memory cost by only capturing the k-best fragments per-pixel of all generated fragments in a scene in a single geometry pass while still ensuring the correct fragment depth order.

Similarly to the A-Buffer, k-Buffer can store fragment data in a fixed-sized per-pixel array, using linked lists or into contiguous regions for each pixel similarly to S-Buffer [VF12], allowing for a per-pixel different k. k-Buffer approaches either assume a user-defined [BCL+07] or dynamic [VPF15] global k for all pixels or a per-pixel variable k [VVPM17] that is calculated using an importance distribution function.

k-Buffer suffers from flickering artifacts due to read-modify-write data hazards that arise when the generated fragments are simultaneously inserted into the buffer in arbitrary depth order. This can be solved by utilizing hardware-accelerated pixel synchronization [Sal13].

3.2.1 Fixed K

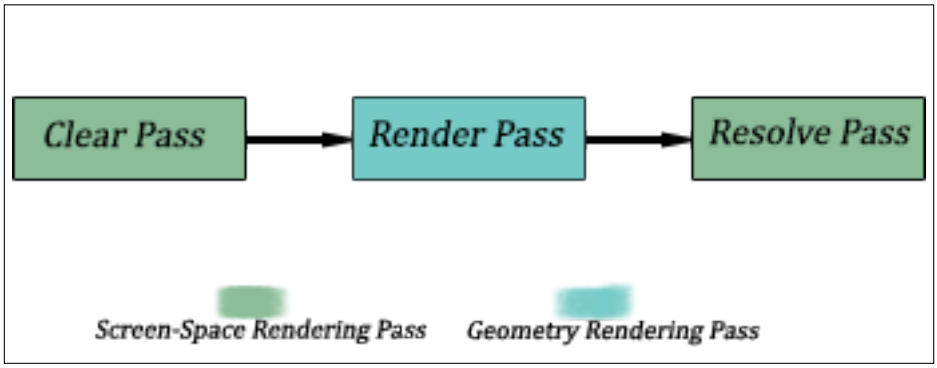

Figure 3.5: Fixed k k-Buffer's rendering pipeline

Fixed k [BCL+07] approaches assume a global, user-defined k for all pixels. Usually, a specific subset of the total fragments is enough for achieving plausible simulation of effects, such as transparency or translucency. Finding the best k for a single scene can be proven to be a very challenging task. On one hand, setting k to a small number can result in visual artifacts as more than k fragments might be required for some pixels to correctly simulate the desired effect. On the other hand, setting k value too high can result in wasteful memory allocation, as large and potentially unused storage is allocated for pixels that contain less than k fragments.

Table 3.2, presents an overview of k-Buffer's Fixed-Sized array performance based on Figure 3.6's scene and k values. It is clearly shown that a high enough

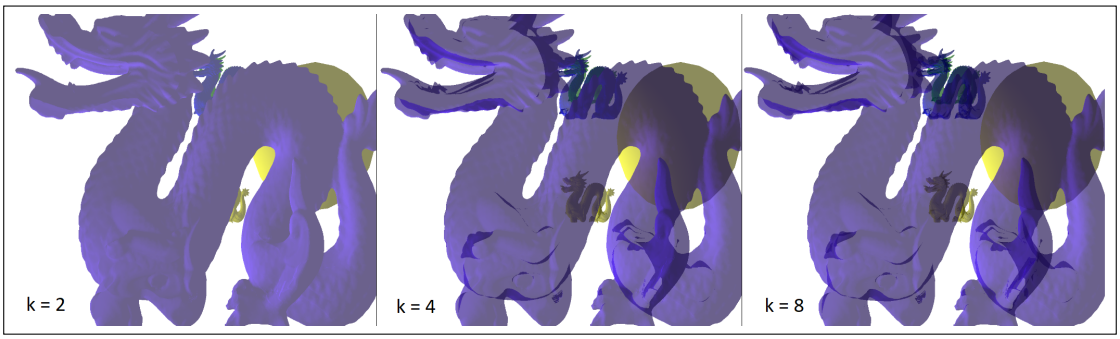

Figure 3.6: OIT using different K values

Table 3.2: k-Buffer fixed-sized array performance with different k values

2500880 total fragments	**k = 2**	**k = 4**	**k = 8**
Memory Consumption (MB)	21.9648	43.9296	87.8592
Clean Stage (ms)	0.053248	0.107136	0.222496
Fragment Capturing Stage (ms)	2.217984	2.652576	4.136224
Resolve Stage (ms)	0.092160	0.135808	0.256000
Total Time (ms)	2.363392	2.895520	4.614720

k value is required in order to simulate OIT adequately (k = 4). For every application, the correct k must be chosen depending on the expected performance and memory consumption. For our implementation, the fixed-sized array method uses OpenGL's *RGBA16F texture* (8 bytes per array location) to store color and depth values. *RGBA16F textures* have a similar size to fixed-array A-Buffer's *RG32F texture* implementation and serves for a better method comparison.

In this thesis, k-buffer fixed k using per-pixel fixed arrays was implemented. An OpenGL RGBA16F texture, namely ***kBufferTexture***, was used for storing fragment information. A geometry pass ("fragment capturing" or "render") pass and two screen-space ("clear" and "resolve" pass) are required.

In the first pass (Algorithm 13), the k-buffer texture is cleared, representing no fragments stored.

Algorithm 13 K-Buffer clear pass

1: **reset** kBufferTexture

In the second, geometry rendering pass, fragments are shaded and inserted into the k-buffer by using insertion-sort, in order to guarrantee that only the k-best fragments are captured in a single pass (Algorithm 14).

In the final, screen-space, pass the final pixel color is calculated using the stored fragments.

Algorithm 14 K-Buffer fragment capturing pass

1: **delimit the following code as critical section by utilizing hardware-accelerated pixel synchronization**
▷ This will guarantee that the critical section of the fragment shader will be executed for only one fragment at a time.
2: **for** i=0; i<k; i++ **do**
3: $fragments[i] \leftarrow bufferTexture[pixel_x][pixel_y][i]$
4: **end for**
▷ Store already sorted fragments into local memory
5: $thisFragmentData \leftarrow (fragmentColor, fragmentDepth)$
6: insertionSort(thisFragmentData, fragments)
▷ Add current fragment to locally stored fragment array if possible
7: **for** i=0; i<k; i++ **do**
8: $bufferTexture[fragment_x][fragment_y][i] \leftarrow fragments[i]$
9: **end for**
▷ Store fragments back into k-buffer texture
10: **end critical section**
▷ Return pixel color

Algorithm 15 K-Buffer resolve pass

1: **for** i=0; i<k; i++ **do**
2: $fragments[i] \leftarrow bufferTexture[pixel_x][pixel_y][i]$
3: **end for**
▷ Store already sorted fragments into local memory, a check could be inserted here in case total pixel fragments are less than k
4: $resolvedColor \leftarrow resolveColor(fragments)$
▷ Return pixel color

3.2.2 Variable K

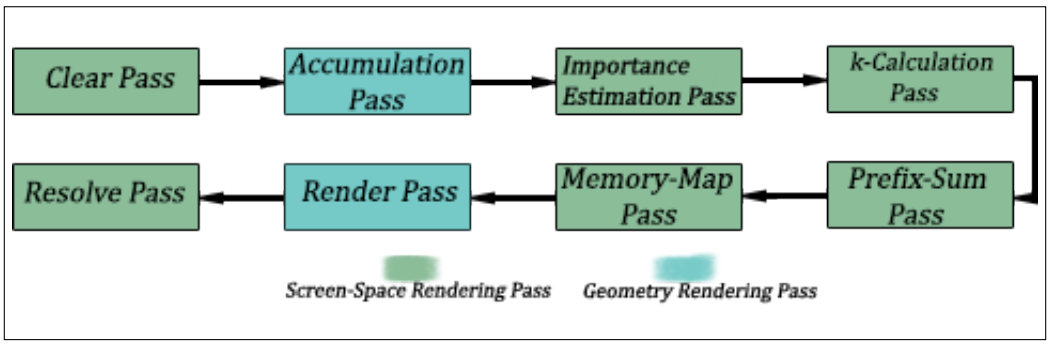

Figure 3.7: Variable k-Buffer's rendering pipeline

Variable k k-Buffer approaches aim to set a different k value for every pixel. In this thesis, Variable k-buffer using Importance Maps [VVPM17] was implemented. While fixed k k-Buffer assumes a pre-assigned, global k value for every pixel, Variable k-buffer using Importance Maps works under a strict memory budget

and aims to set a different k value for every pixel according to an importance distribution function, which determines each pixels k value based on its estimated importance and without exceeding the pre-allocated memory budget.

The importance distribution is calculated in screen-space, stored in an importance map (texture), and passed to the k-buffer before the generation of the fragment data. Depth Complexity and Periphery heuristics were implemented, as described in [VVPM17]. The memory is fixed and pre-allocated, and similarly to the S-buffer [VF12], linearly organized into variable contiguous regions of length equals to the per-pixel k for each pixel.

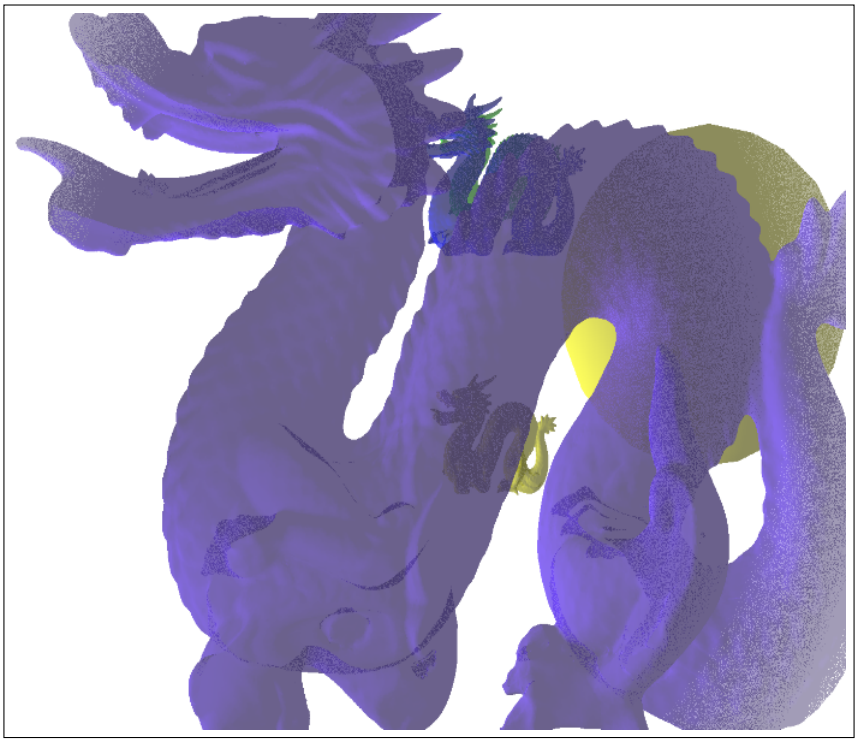

Figure 3.8: OIT rendered using variable k-Buffer using importance maps with 20MB of allocated memory.

Figure 3.8 illustrates rendering a scene using Variable k-Buffer. While only allocating 20MB of memory, it offers a better visual result compared to fixed k-buffer with k = 2 (Figure 3.6) while requiring less memory (Table 3.2). Pixels with higher complexity (depth complexity heuristics) and pixels around image's center (periphery heuristic, focusing on image center) have higher k values.

Variable k-Buffer builds on S-Buffer's [VF12] architecture, allowing for a different k per-pixel. It requires two geometry rendering passes, one that counts the fragments per-pixel ("Accumulation" pass) and one that captures the fragments ("Fragment Capturing" or "render" pass). Additionally, six screen-space passes are required, two are the S-Buffer's "memory map" and "prefix sum" passes that calculate each pixels groups prefix sums and map their fragments to their respective pixel's contiguous memory area, a pass for calculating per-pixel importance, a pass for calculating per-pixel k and, a final "resolve" pass. A "clear" pass could also be used to reset each pixel's k every frame.

Our implementation requires a texture for counting fragments per-pixel, ***coun-***

terTexture, a texture for pixel head address, ***addressTexture***, a Shader Storage Buffer, ***data***, for storing fragment information and a Shader Storage Buffer, ***addressMap***, for storing group addresses. A texture for storing the calculated per-pixel k values, ***kValueTexture***, and a texture for saving per-pixel importance, ***pixelImportanceTexture***, an atomic counter for counting total fragments, ***totalFragments***, and an atomic counter for storing the sum of all per-pixel importances, ***importanceSum***, are also required.

In the first pass, a geometry pass similar to the S-Buffer's "accumulation" pass, that counts all fragments per-pixel is performed (Algorithm 8).

The second pass, a screen-space pass, calculates the per-pixel importance values. Vasilakis et al. [VVPM17] describe an unnormalized importance distribution function based on the expected depth complexity, the distance of pixel from a point of interest **o** (periphery heuristic) and the Fresnel term of the nearest fragment. Only the periphery and depth complexity heuristics were used in our implementation.

Periphery heuristic [GFD+12] gradually lowers the significance of pixels towards the edges of the framebuffer by assuming that the information near the point of interest **o** (usually where the user gazes) is perceptually more important.

$$I_{per} = 1 - (\frac{||p - o||}{\sqrt{w^2 + h^2}})^d$$

With **p** being the pixel's position, **o** the point of interest and **d** controlling the rate of importance drop towards the periphery.

Depth-complexity heuristic is the ratio of the number of visible layers N_{layers} that should be drawn at pixel p against a anticipated maximum layer count of a scene N_{limit}.

$$I_d = \alpha + (1 - \alpha)min(1, \frac{N_{layers}}{N_{limit}})$$

With α being a range bias that assists pixels with very low depth complexity and also dampens the prevalence of pixels with very high layer count in the importance distribution. N_{layers} can be set as the total fragments of the pixel and N_{limit} as roughly the maximum of the scenes total fragments per-pixel.

Finally, the per-pixel importance is calculated as product of the above heuristics with an added noise bias, to mask potentially visible abrupt transition zones across otherwise smooth image regions.

$$I_{px}(p) = max(0, I_{per} * I_d + \lambda * (\rho - 0.5))$$

With $\rho \in [0,1]$ being a uniformly distributed scalar, and λ the being the noise level set to 0.1.

The third pass, a screen-space pass, calculates the per-pixel k values by using the stored total importance value I_{tot}, stored in an atomic counter. With **M** being the desired memory size and **k(p)** being the k-value of pixel **p**, **P(p)** being the importance-based probability, $P(p) = \lfloor \frac{I(p)}{I_{tot}} \rfloor$, each pixel's k value is calculated by:

$$k(p) = \lfloor M * P(p) \rfloor$$

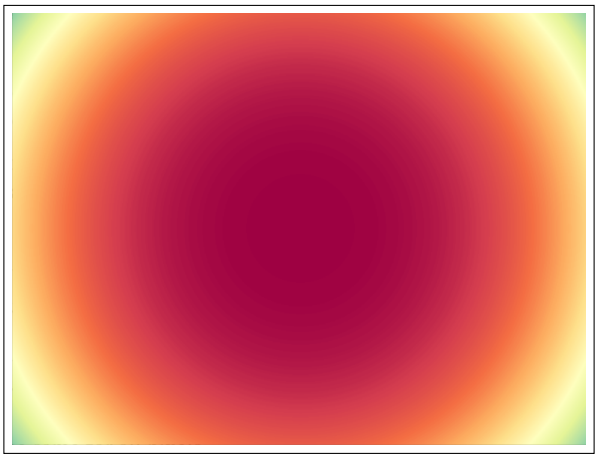

Figure 3.9: Heatmap of the Periphery Heuristic with the point of interest set to the center of the image. Brighter red colors indicate higher importance value.

Algorithm 16 Variable k-buffer importance calculation pass

1: $pixelImportance \leftarrow I_{px}(pixel_x, pixel_y)$
▷ Calculate pixel's importance using the importance distribution function
2: $pixelImportanceTexture[pixel_x][pixel_y] \leftarrow pixelImportance$
▷ Store per-pixel importance
3: **atomic addition** of pixelImportance to importanceSum atomic counter

Or by expressing the memory **M** as $M = k_{avg} * width * heigh$, with **width** and **height** indicating the framebuffer's size and, k_{ave} the average per-pixel k, each pixel's k value could also be calculated by:

$$k(p) = \lfloor k_{avg} * width * heigh * P(p) \rfloor$$

Algorithm 17 Variable k-buffer k-calculation pass

1: $totalImportance \leftarrow$ value of importanceSum atomic counter
2: $pixelImportance \leftarrow pixelImportanceTexture[pixel_x][pixel_y]$
3: **if** $pixelImportance \neq 0$ **then** ▷ Ensure that pixel has at least one fragment
4: $kValueTexture[pixel_x][pixel_y] \leftarrow k(p)$
▷ Calculate pixels k value
5: **end if**
6: **atomic addition** of pixelImportance to importanceSum atomic counter

The fourth and fifth passes (Algorithms 18, 19), screen-space passes, are similar to the S-Buffer's "prefix-sum" and "memory-map" stages, but instead of using the number of fragments per-pixel, the calculated **k(p)** is used.

Afterwards, a geometry rendering pass ("fragment capturing" or "render" pass, Algorithm 20), captures the k-best available fragments.

Finally, the pixel's color is calculated ("resolve" pass).

Algorithm 18 Variable k-buffer prefix sum pass

1: $fragNum \leftarrow kValueTexture[pixel_x][pixel_y]$
▷ Check if pixel contains fragments
2: **if** $fragNum$ is zero **then**
3: discard fragment
4: **end if**
5: $pixelGroup \leftarrow H(pixel_x, pixel_y)$
▷ Calculate pixel group using the hash function.
6: **atomic addition** to $addressMap[pixelGroup]$ with fragNum and store its previous value to $pxAddress$
7: $addressTexture[pixel_x][pixel_y] \leftarrow pxAddress$
▷ Set head address for this pixel

Algorithm 19 Variable k-buffer memory map pass

1: $pixelGroup \leftarrow H(pixel_x, pixel_y)$
▷ Calculate pixel group using the hash function.
2: **if** pixelGroup <groupNum **then**
3: **for** i=0;i< pixelGroup; i++ **do**
4: $sum \leftarrow sum + addressMap[i]$
5: **end for**
▷ Only forward prefix sum shown here as inverse prefix sum was not implemented.
6: $addressMap[pixelGroup] \leftarrow sum$
▷ Store the calculated group prefix sum.
7: **end if**

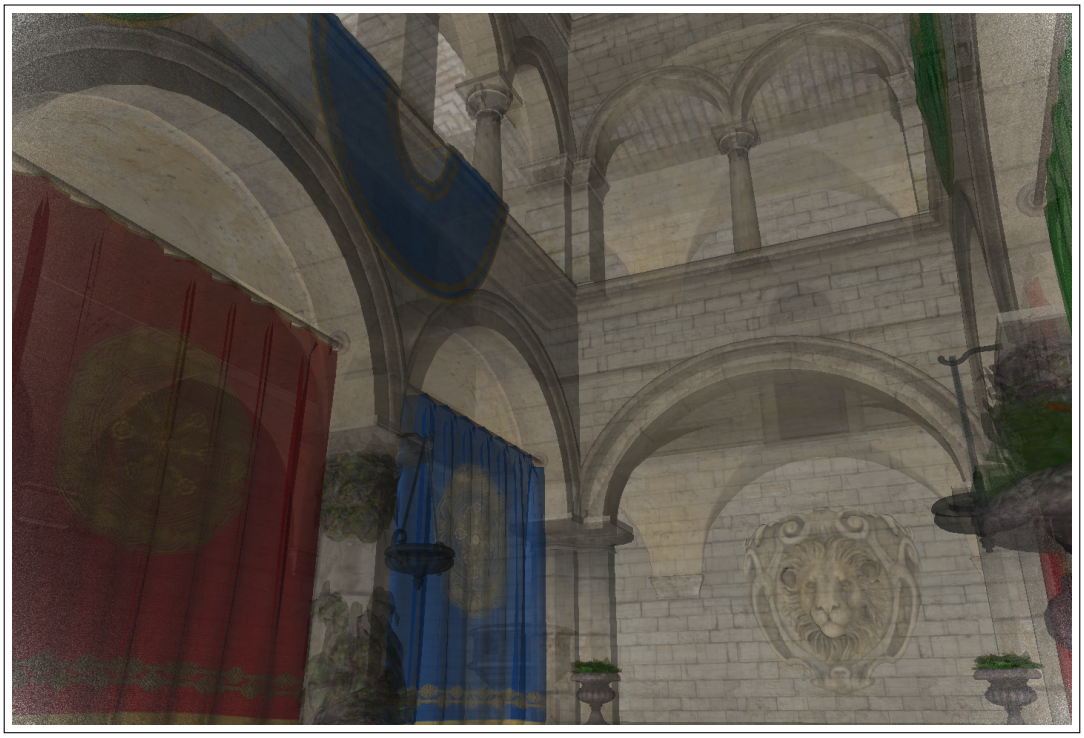

Figure 3.10: Crytek Sponza OIT rendered with [VVPM17], Sponza model downloaded from Morgan McGuire's Computer Graphics Archive [McG17]

Algorithm 20 Variable k-buffer fragment capturing pass

1: $pixelGroup \leftarrow H(pixel_x, pixel_y)$
2: $fragmentAddress \leftarrow addressTexture[fragment_x][fragment_y]$
▷ Get pixels head address
3: $pixelK \leftarrow kValueTexture[fragment_x][fragment_y]$
4: $memoryOffset \leftarrow fragmentAddress + addressMap[pixelGroup]$
▷ Inverse prefix sum could be used here
5: $fragmentColor \leftarrow calculateColor()$
6: **delimit the following code as critical section by utilizing hardware-accelerated pixel synchronization**
7: *A check could be added here in order to keep track of how many pixel's fragments have already been stored, if less than pixelK have been stored, we can just store the fragment and then sort the array, similarly to the A-Buffer*
8: **for** i=0; i<pixelK; i++ **do**
9: $fragments[i] \leftarrow data[memoryOffset + i]$
10: **end for**
▷ Locally store the currently stored fragments
11: $thisFragmentData \leftarrow (fragmentColor, fragmentDepth)$
12: insertionSort(thisFragmentData, fragments)
▷ Add current fragment to locally stored fragment array if possible
13: **for** i=0; i<pixelK; i++ **do**
14: $data[memoryOffset + i] \leftarrow fragments[i]$
15: **end for** ▷ Store fragment values back into global memory
16: **end of critical section**

Algorithm 21 Variable k-buffer resolve pass

1: $pixelGroup \leftarrow H(pixel_x, pixel_y)$
2: $fragmentAddress \leftarrow addressTexture[fragment_x][fragment_y]$
▷ Get pixels head address
3: $pixelK \leftarrow kValueTexture[fragment_x][fragment_y]$
4: $memoryOffset \leftarrow fragmentAddress + addressMap[pixelGroup]$
5: **for** i=0; i<pixelK; i++ **do**
6: $fragments[i] \leftarrow data[memoryOffset + i]$
7: **end for**
8: $resolvedColor \leftarrow resolveColor(fragments)$
9: $outputColor \leftarrow resolvedColor$

3.3 MFR Algorithms Demo Application

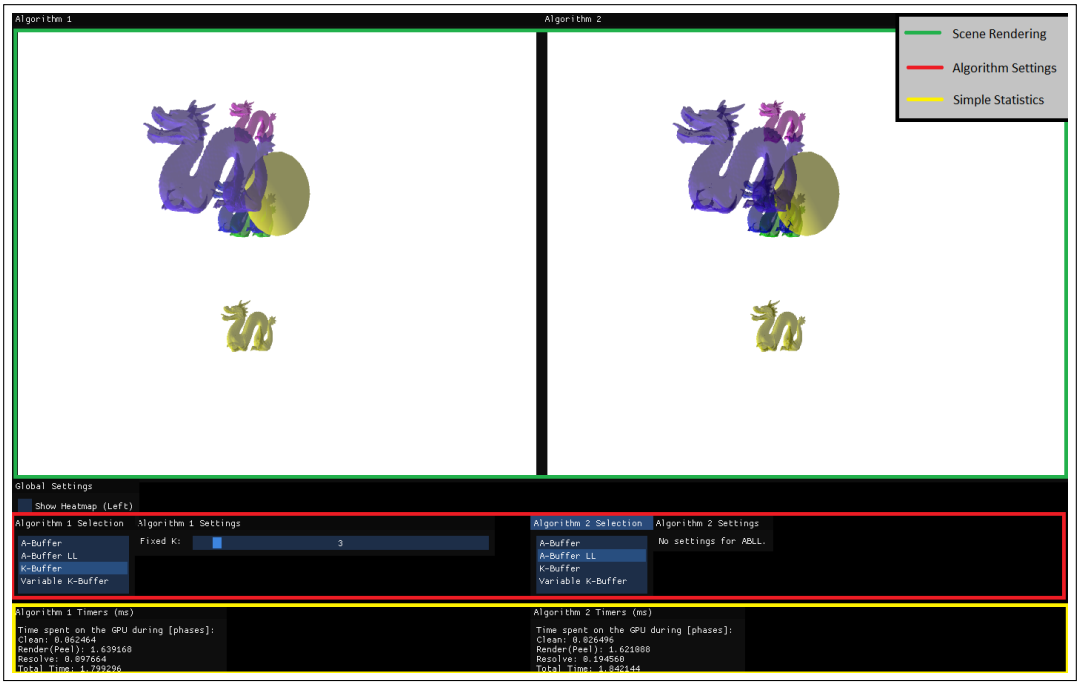

Figure 3.11: Overview of the application's GUI

A rendering engine, capable of rendering simple 3D using multifragment rendering methods has been developed for this thesis. Every experiment and scene render used it. As a side-product, it can also be used as a multifragment rendering methods demo application. It offers a simple and intuitive GUI, simultaneous side-by-side rendering of two algorithms, interactive algorithm and setting (such as global k, memory M) modification, MFR method difference using a heatmap and, simple performance statistics.

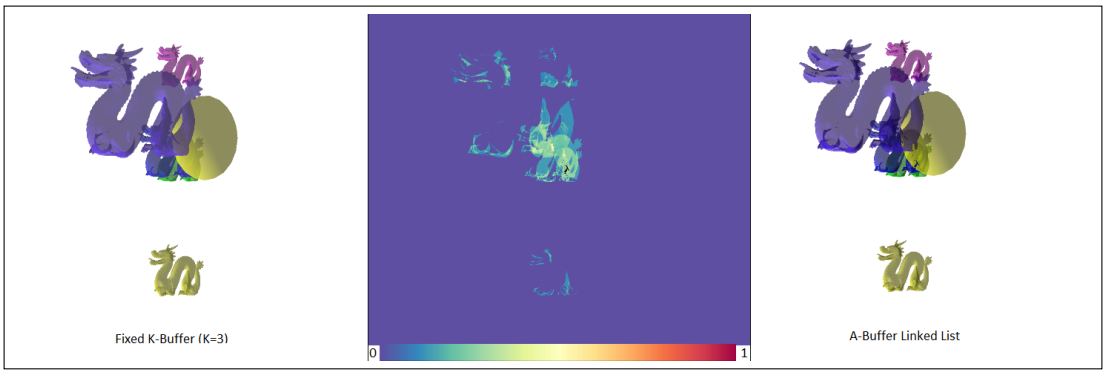

Figure 3.12: Heatmap visually indicating MFR method differences

Chapter 4

Deep Learning Variable K-Buffer

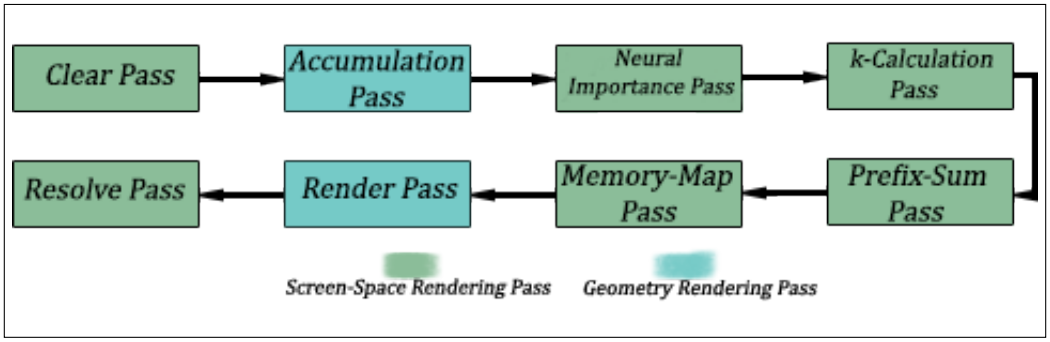

Figure 4.1: Deep Learning Variable K-Buffer pipeline

Deep Learning Variable K-buffer takes advantage of the Variable k-Buffer architecture, using the S-Buffer to set per-pixel variable k values, and attempts to approximate the quality of the A-Buffer by using a simple neural network. A-Buffer provides the best rendering quality as it can fully exploit a scene's geometry and is consequently set as a reference point. Deep Learning Variable k-Buffer, replaces Variable k-Buffer using Importance Maps [VVPM17] unnormalized importance distribution function ("Importance Estimation" pass) with a per-pixel unnormalized predicted importance using a neural network ("Neural Importance" Pass).

Deep Learning Variable K-Buffer aims to improve rendering quality by providing a per-pixel predicted k value closer to the A-Buffer than ones provided using importance maps and heuristics, while still maintaining a strict memory budget. Optimal K values were used as a target (label) for the neural network. Optimal K values were calculated using a greedy algorithm that gradually removes fragments with the least visual impact, when removed, compared to the A-Buffer resolved colors, until a pre-specified fragment count is reached, representing the desired memory size.

Deep Learning Variable K-Buffer requires the same resources and rendering passes as Variable k-Buffer using Importance Maps, with only the "Importance Estimation" pass being different, as an importance distribution function is not used. The neural network takes 12 features as input and returns an unnormalized pixel importance.

As each pixel's importance is predicted independently, adjacent pixel predictions may result to greatly different k values, thus creating visual artifacts.

Algorithm 22 Deep Learning Variable k-buffer importance calculation pass

1: $pixelImportance \leftarrow$ Neural Network's predicted value
▷ Calculate pixel's importance using the neural network
2: $pixelImportanceTexture[pixel_x][pixel_y] \leftarrow pixelImportance$
▷ Store per-pixel importance
3: **atomic addition** of pixelImportance to importanceSum atomic counter

4.1 Network Architecture

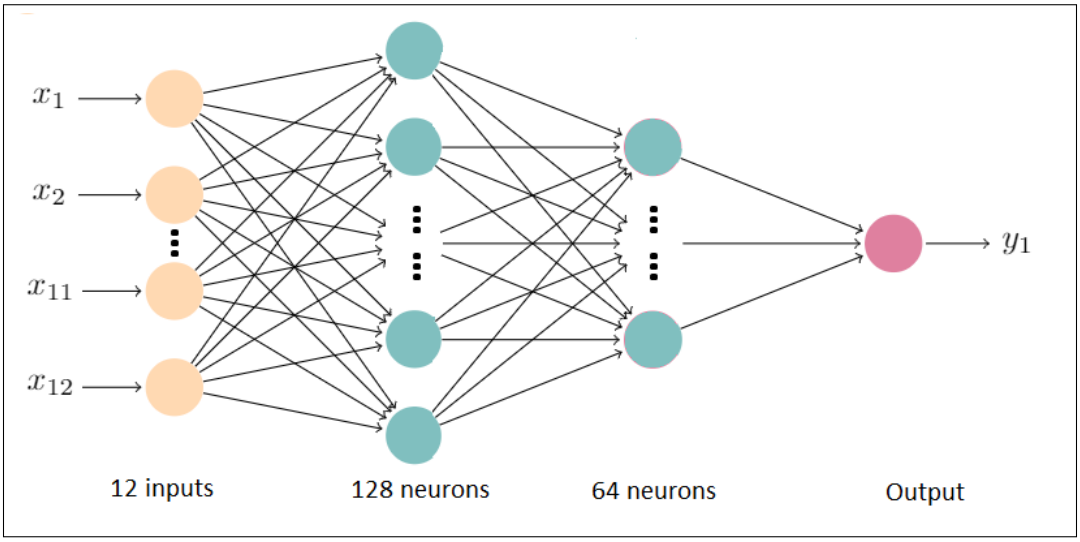

Figure 4.2: Neural network architecture

The neural network is composed of an input layer that takes 12 features as input, two dense hidden layers of 128 and 64 neurons respectively and an output layer that returns the per-pixel importance. The hidden layers use *ReLU* as an activation function while the output layer uses a *Sigmoid* activation function as it returns a value $x \in [0, 1]$, therefore it is a suitable choice for predicting a per-pixel importance value. Stochastic gradient descent (SGD) is used as an optimizer with a learning rate of *0.001* and mean-squared-error (MSE) is used as a loss function. Finally, *L1,L2* regularizers (0.3,0.35) are used for the hidden layers with 128,64 neurons respectively.

4.2 Feature Data and Training Set

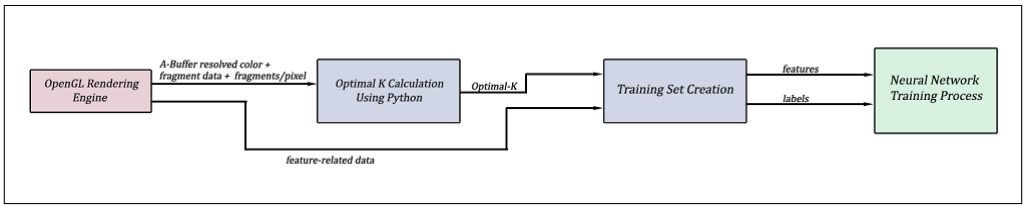

Figure 4.3: Data creation and training process

A-Buffer's data are exported from the renderer into .txt or .csv files and processed with Python, creating a Training Set for the neural network.

The neural network takes 12 per-pixel features as inputs and returns an importance value $i \in [0, 1]$. The features are described as:

1. **pixelFragments/TotalFragments**, with pixel fragments being the pixels fragments and total fragments being the scenes total fragments.
2. **reservedFragments/TotalFragments**, with reservedFragments being the allocated memory size.
3. **local min-z**, the pixel's nearest fragment depth.
4. **local max-z**, the pixel's farthest fragment depth.
5. **scene min-z**, the scene's nearest fragment depth.
6. **scene max-z**, the scene's farthest fragment depth.
7. **blend color RGB**, the RGB color of all fragments added and divided by fragment count (3 features).
8. **nearest color RGB**, the RGB color of the nearest (3 features).

4.2.1 Exporting Data from the Engine

Multifragment rendering method's shaders could be modified to calculate and store the desired neural network's input features into different textures. Every "resolve" pass can calculate and store each of the 12 features as it can access the data of every stored fragment. Any geometry rendering pass could be also used to calculate the features, as they do not require the fragments to be sorted and can be calculated with simple operations. In Deep Learning Variable K-Buffer, features can be calculated with atomic operations during the "accumulation" pass, stored into different textures, and then used during the "Neural Importance" pass as inputs for the neural network. In this thesis, the A-Buffer was used as a reference and for exporting the feature data to be processed into a Training Dataset.

For the creation of the dataset, after calculating and storing the features using the GPU, they are read on the CPU side, using C++ and OpenGL, and then exported into different files for further processing.

4.2.2 Finding the Optimal K

We refer to "Optimal K" as the k values, calculated using a greedy algorithm, whose blending result's quality is closer to the A-Buffer's quality when compared with k values calculated using importance maps (Figure 4.5). The "Optimal K" offers the same fragment count as when using importance maps, thus having the same advantage of reducing memory requirements when compared to A-Buffer or K-Buffer methods. All sorted fragment data, fragments per pixel, and the A-Buffer's resolved colors per pixel are required for the computation of Optimal K values, as a multifragment rendering pipeline is simulated on the CPU using Python, making it possible to blend fragments and find the color differences as well as operate on fragment-like data structures.

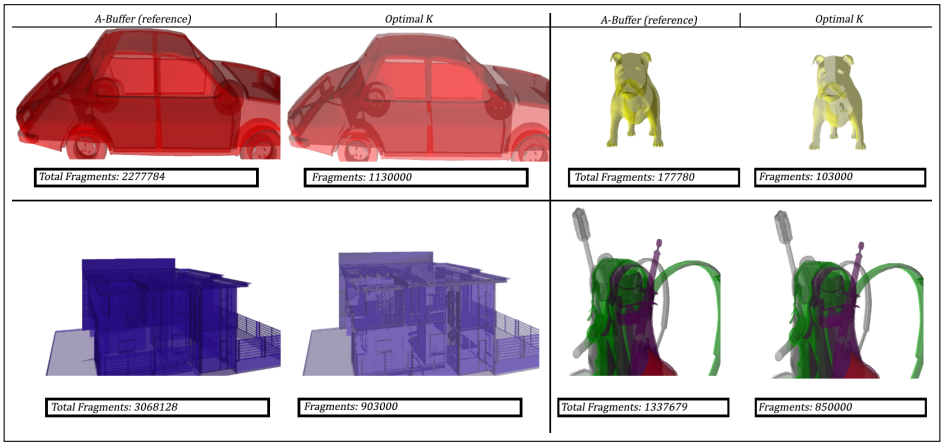

Figure 4.4: OIT using Optimal K as k values

The algorithm works in a backward fashion, starting from all fragments A per pixel (A-Buffer) and slowly removing fragments until we reach our desired fragment count. A pixel is eligible to have its fragments reduced if the pre-specified memory budget has not yet been reached and if it has more than one fragment left.

The algorithm works as follows, first assign the per-pixel optimal-k value equal to all generated pixel fragments A for all pixels, once at the beginning. Then, for every eligible pixel, resolve the color with a fragment removed ($optimal_k - 1$), estimate the change in color compared to A-Buffer that is caused by removing one fragment from the pixel, and insert the result cd(p) and the pixel's coordinates into a min-heap, sorted by cd(p). Each heap node will then contain $(cd(p), pixel_x, pixel_y)$. Any other similar data structure could also be used.

$$cd(p) = \sqrt{(r_{old} - r_{new})^2 + (g_{old} - g_{new})^2) + (b_{old} - b_{new})^2}$$

With **cd(p)** being the per-pixel color difference with the A-Buffer, $(r_{new}, g_{new}, b_{new})$ being the color blended after removing one fragment from the pixel and, $(r_{old}, g_{old}, b_{old})$ being the A-Buffer's exported resolved values for the same scene and pixel.

Afterwards, permanently remove a fragment from the pixel with the minimum contribution to the A-Buffer, i.e. the pixel with the minimum cd(p), by setting its optimal-k value as its current optimal-k minus one. The above process is repeated, gradually removing fragments with the lowest contribution one by one, until the pre-specified fragment count is reached or if no more fragments can be removed.

After this process, each pixels k value is divided by the scenes total fragment count and exported. This results in small per-pixel values, $v \in [0, 1]$ that could be interpreted as a per-pixel importance.

The final calculated per-pixel importance from the "Optimal K" is used as a **label** for the neural network.

Color blending (resolve) is done in the same way as the A-Buffer thus the fragments must be sorted in the simulated pipeline to ensure correct blending. This can be done by either by exporting pre-sorted fragment data from the rendering engine or by later sorting them in the simulated pipeline. In Algorithm 23, **currentMemory** refers to current fragment count, **totalMemory** refers to scene's total fragment count, **wantedMemory** refers to pre-specified, wanted fragment count (allocated memory) and, **optimalK** is a 2D array storing each pixels optimalK values.

Algorithm 23 Optimal K calculation

1: Set every pixels optimalK equal to their total fragment count
2: $currentMemory \leftarrow totalMemory$
3: **for** every eligible pixel **do**
4: $newColor \leftarrow$ blend color calculated by reducing pixels optimalK by 1 fragment ▷ Blending is done in the same way as in the A-Buffer, up to OptimalK fragments
5: $cd(pixel_x, pixel_y) \leftarrow$ color difference between A-Buffer ▷ Calculate color difference using the above formula.
6: Add $(cd(pixel_x, pixel_y), pixel_x, pixel_y)$ into a min-heap
7: **end for**
8: **while** currentMemory $\neq$ wantedMemory **do**
9: Extract pixel with minimum cd from heap
10: $optimalK[pixel_x][pixel, y] \leftarrow optimalK[pixel_x][pixel, y] - 1$
11: $currentMemory \leftarrow currentMemory - 1$
12: **if** one more fragment can be reduced from this pixel **then**
13: $newColor \leftarrow$ blend color calculated by reducing pixels optimalK by 1 fragment
14: $cd(pixel_x, pixel_y) \leftarrow$ color difference between A-Buffer
15: Add $(cd(pixel_x, pixel_y), pixel_x, pixel_y)$ into the min-heap
16: **end if**
17: **end while**
18: **for** every pixel **do**
19: $optimalK[pixel_x][pixel_y] \leftarrow optimalK[pixel_x][pixel_y]/totalMemory$
20: **end for**

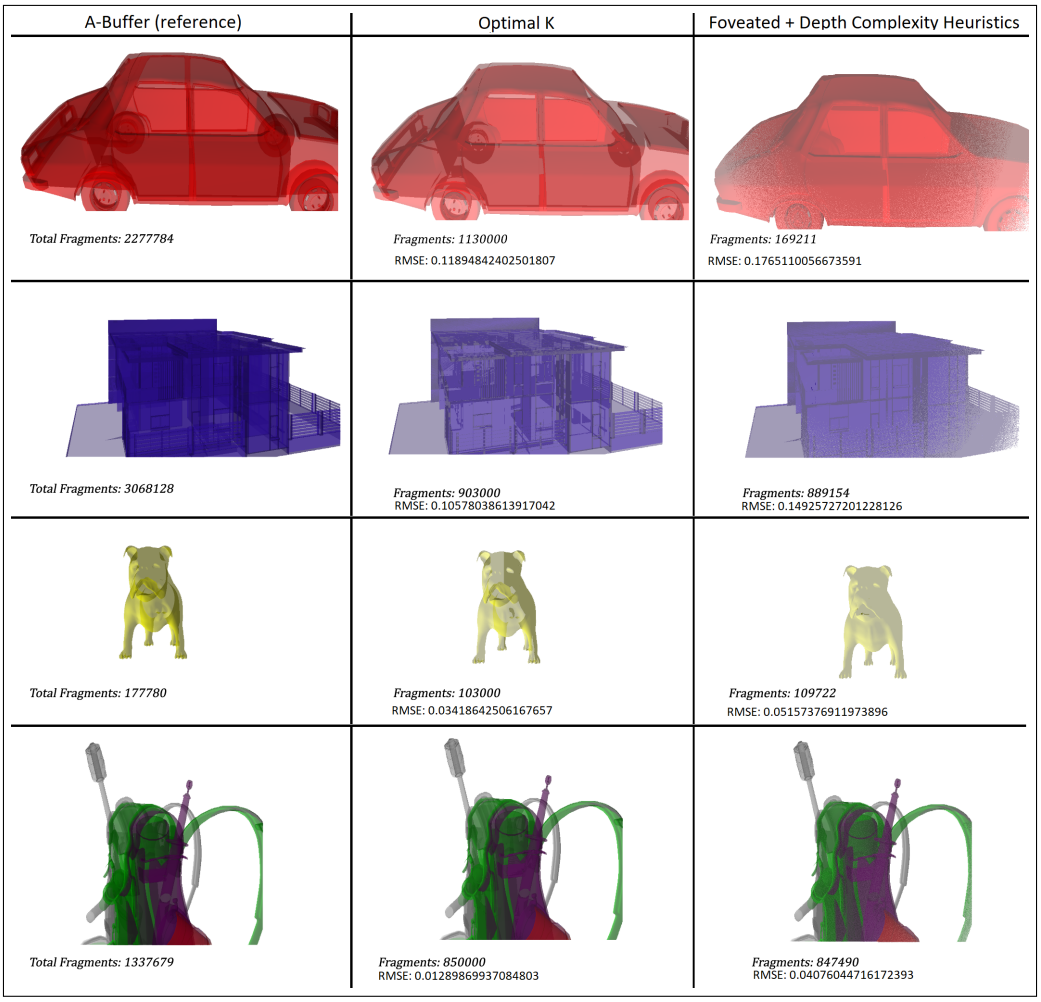

Figure 4.5: OIT rendered with Variable K-buffer, calculating k values using importance maps [VVPM17] and Optimal K, results are compared using RMSE with the A-Buffer set as reference.

4.2.3 Creating and Exporting the Training Data

In total, 8 scene's, with 8 views per scene, data were exported from the OpenGL renderer. For each view, five different variations with different memory budget of Optimal K are computed. In total,

$$framebufferWidth * framebufferHeight * scenes * views$$

pixels are exported.

All fragments' feature data and optimal k are imported into a Python script, uniformly at random sampled, and 13% of total data is selected, and then exported into multiple files.

4.3 Training the Network

The training pipeline is created using Python, TensorFlow and Keras. The Dataset is loaded and split into training (75%), validation data (15%) and test data (10%) with 512 batch size.

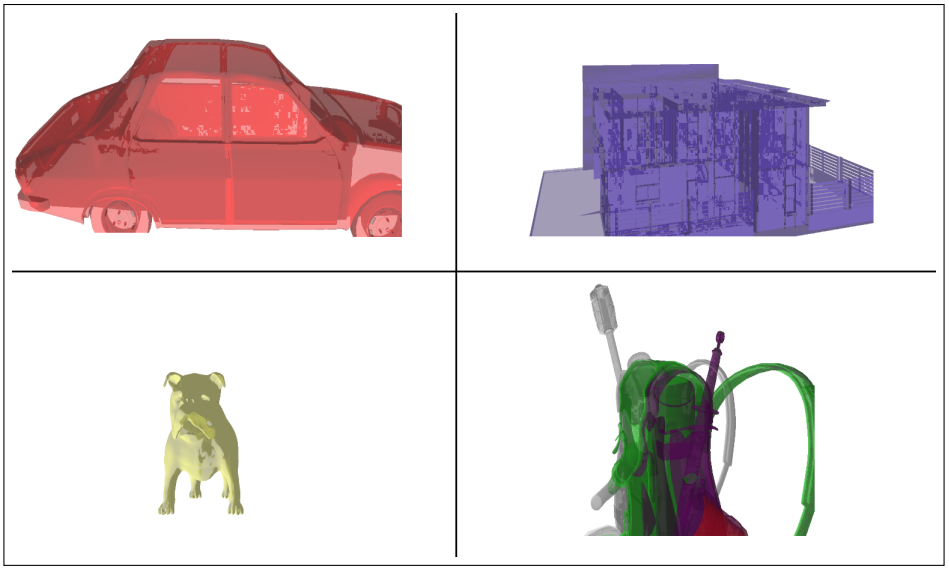

Figure 4.6: OIT rendered with Variable K-buffer, calculating k values using the neural network.

Chapter 5

Experimental Comparative Evaluation

In this Chapter, we present an experimental evaluation of the Deep Learning Variable k-buffer approach when compared with Variable k-Buffer using Importance Maps [VVPM17], excluding fresnel heuristic, with a focus mainly on image quality and comparison with the A-Buffer's (reference) quality. All experiments were conducted using a 1430x960 viewport with varying memory demands M, on an NVIDIA GTX1660 Super GPU.

All methods were implemented in a simple modern OpenGL renderer by integrating order-independent transparency effects. ꟻLIP [ANA+20] is an image quality metric developed that aims to achieve a per-pixel error map that is perceptually correct. The produced error map approximates the difference perceived by humans when alternating between two images. ꟻLIP is used with its default settings.

The visual quality of the three methods is showcased in Figure 5.2 and Figure 5.3, including the ꟻLIP error maps for the Variable-k buffer methods when compared with the A-Buffer. Table 5, presents quality measurements with the A-Buffer as a reference, the mean ꟻLIP value (the lower the better) and RMSE, as well as the fragment distribution under a fixed memory budget in all test scenes.

Quality. The importance maps approach produced lower quality images, when compared to the A-Buffer, due to its k-calculation strategy. Our variable k-buffer offers a better result due to its ability to calculate and assign a per-pixel importance value that is closer to the A-Buffer, but it may suffer from visual artifacts due to neighboring uneven fragment distribution (Figure 5.1).

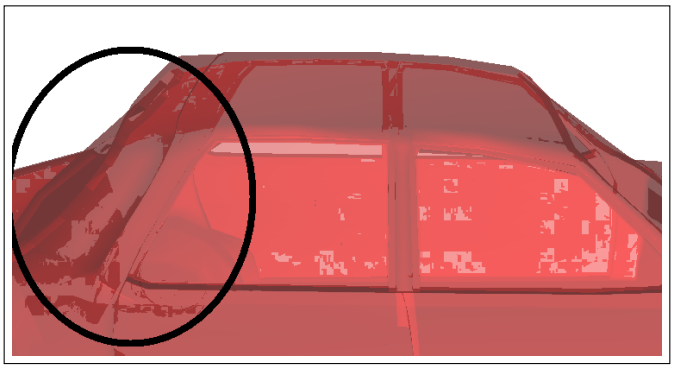

Figure 5.1: Deep Learning Variable-k buffer, artifacts caused when there is a big difference in adjacent pixel k-values.

Table 5 presents quality measurements, RMSE and ꟻLIP mean value, as well as each scenes fragment distribution and the method's reserved fragments. Deep Learning variable k-Buffer appears to be closer to the A-Buffer's quality.

Performance. Both methods are expected to have around the same performance and the same memory requirements as they use the same architecture. The only performance difference could be noticed in the "Importance Estimation" pass due to neural network prediction time.

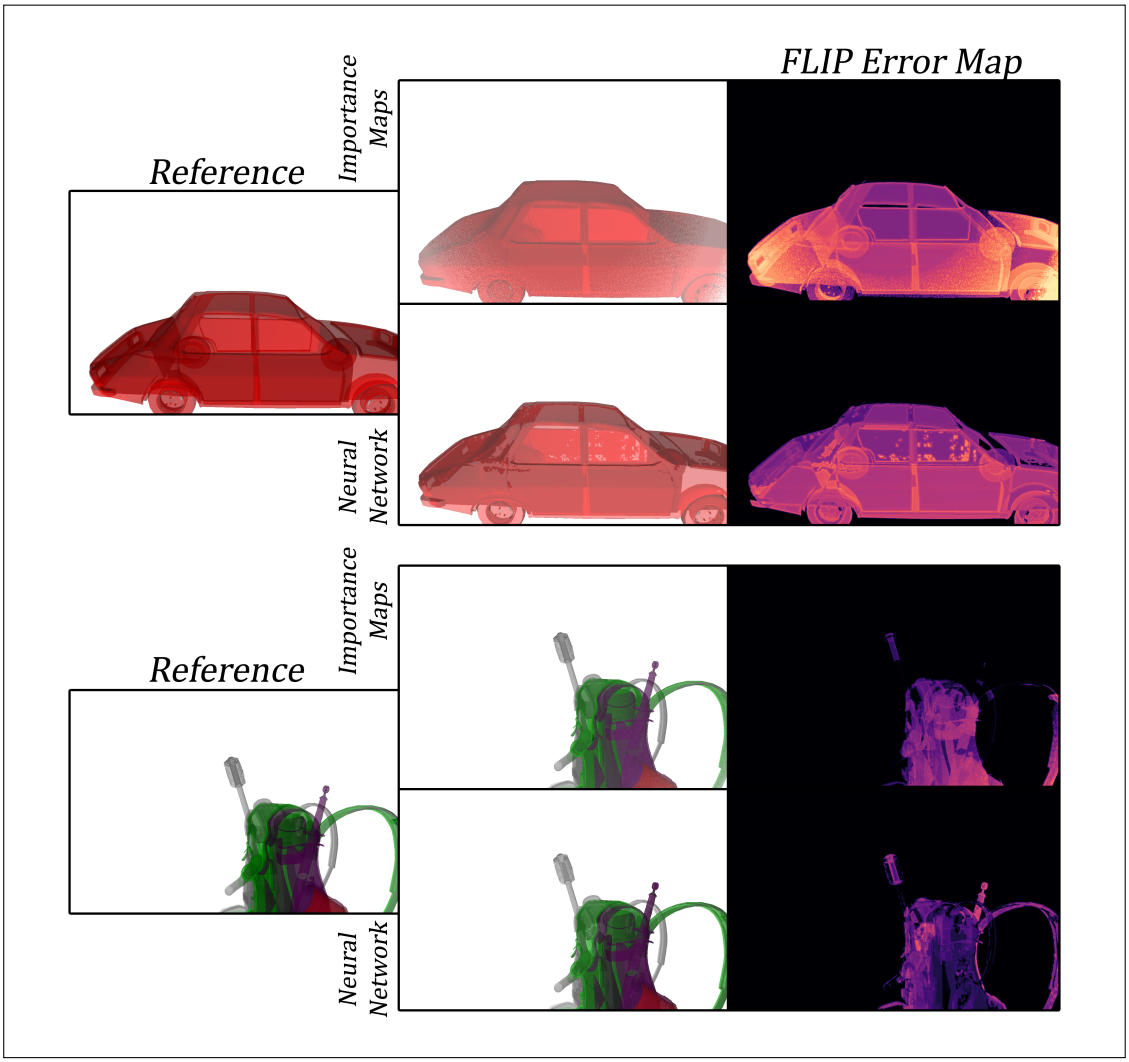

Figure 5.2: Quality evaluation between the two variable k-buffer approaches in two different scenes (car and backpack). ꟻLIP error map of both methods compared with the A-Buffer.

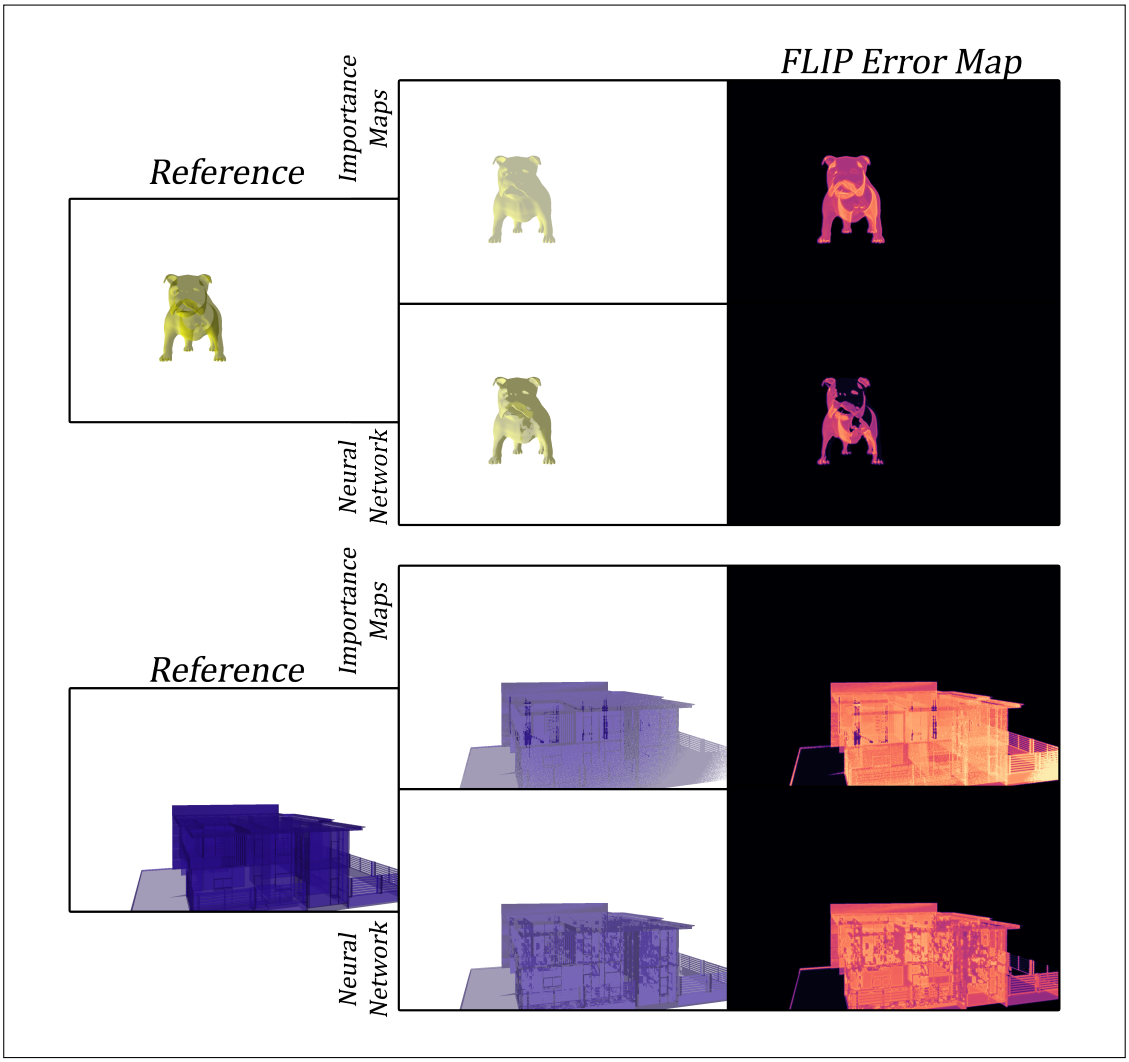

Figure 5.3: Quality evaluation between the two variable k-buffer approaches in two different scenes (dog and house). ꟻLIP error map of both methods compared with the A-Buffer.

Table 5.1: A quality comparison between the two methods.

Method	Deep Learning	Importance Maps [VVPM17]
Car Scene, 2277784 scene fragments, 1130000 reserved fragments		
RMSE:	0.122616	0.176511
FLIP:	0.184187	0.226326
Backpack Scene, 1337679 scene fragments, 850000 reserved fragments		
RMSE:	0.028977	0.040760
FLIP:	0.034880	0.043894
Dog Scene, 177780 scene fragments, 103000 reserved fragments		
RMSE:	0.035283	0.051573
FLIP:	0.019285	0.030001
House Scene, 3068128 scene fragments, 903000 reserved fragments		
RMSE:	0.110599	0.149257
FLIP:	0.168478	0.202734

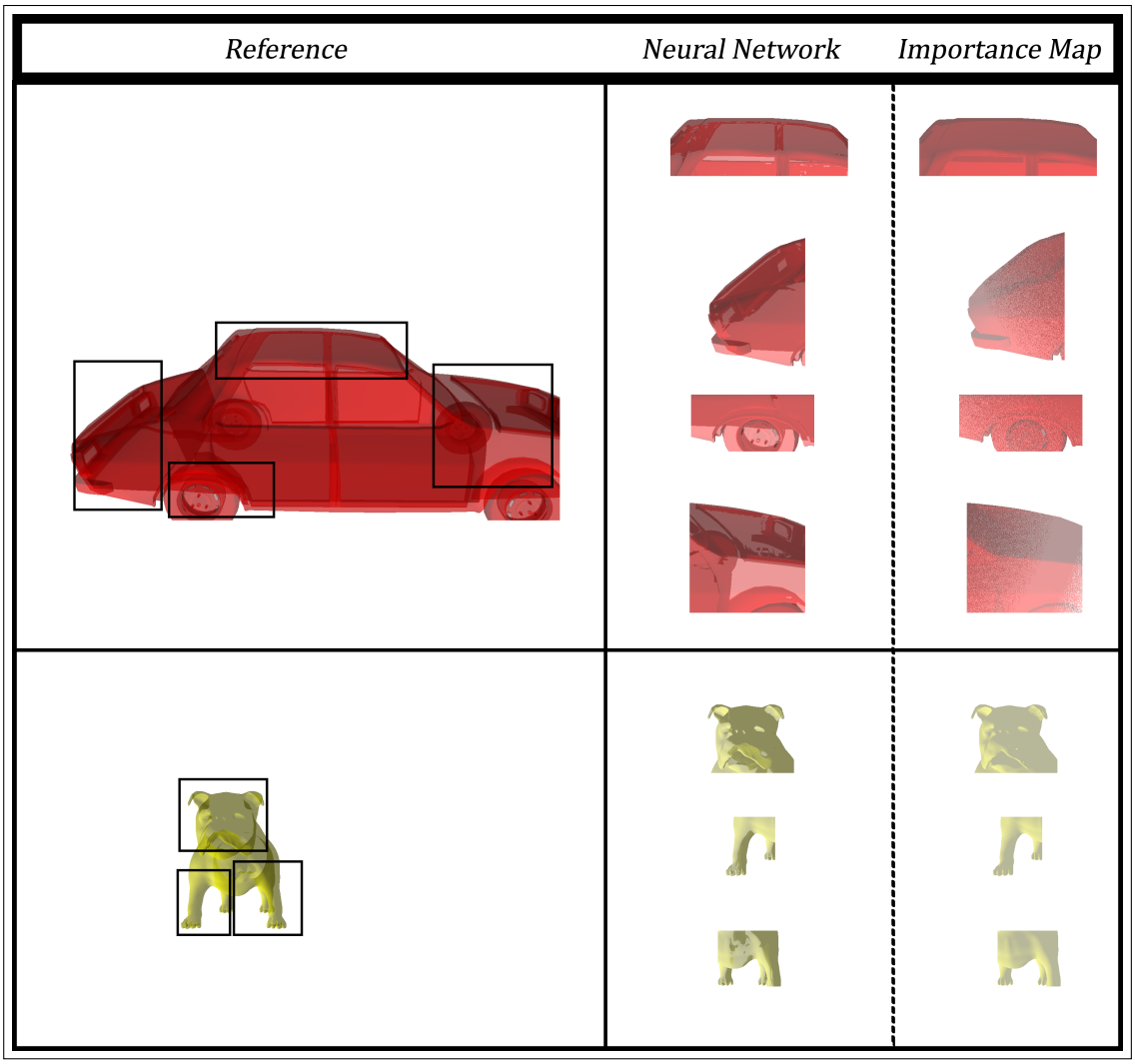

Figure 5.4: Quality evaluation between the two variable k-buffer approaches in two different scenes (car and dog). Indicated areas, are areas where neural network offers a better result.

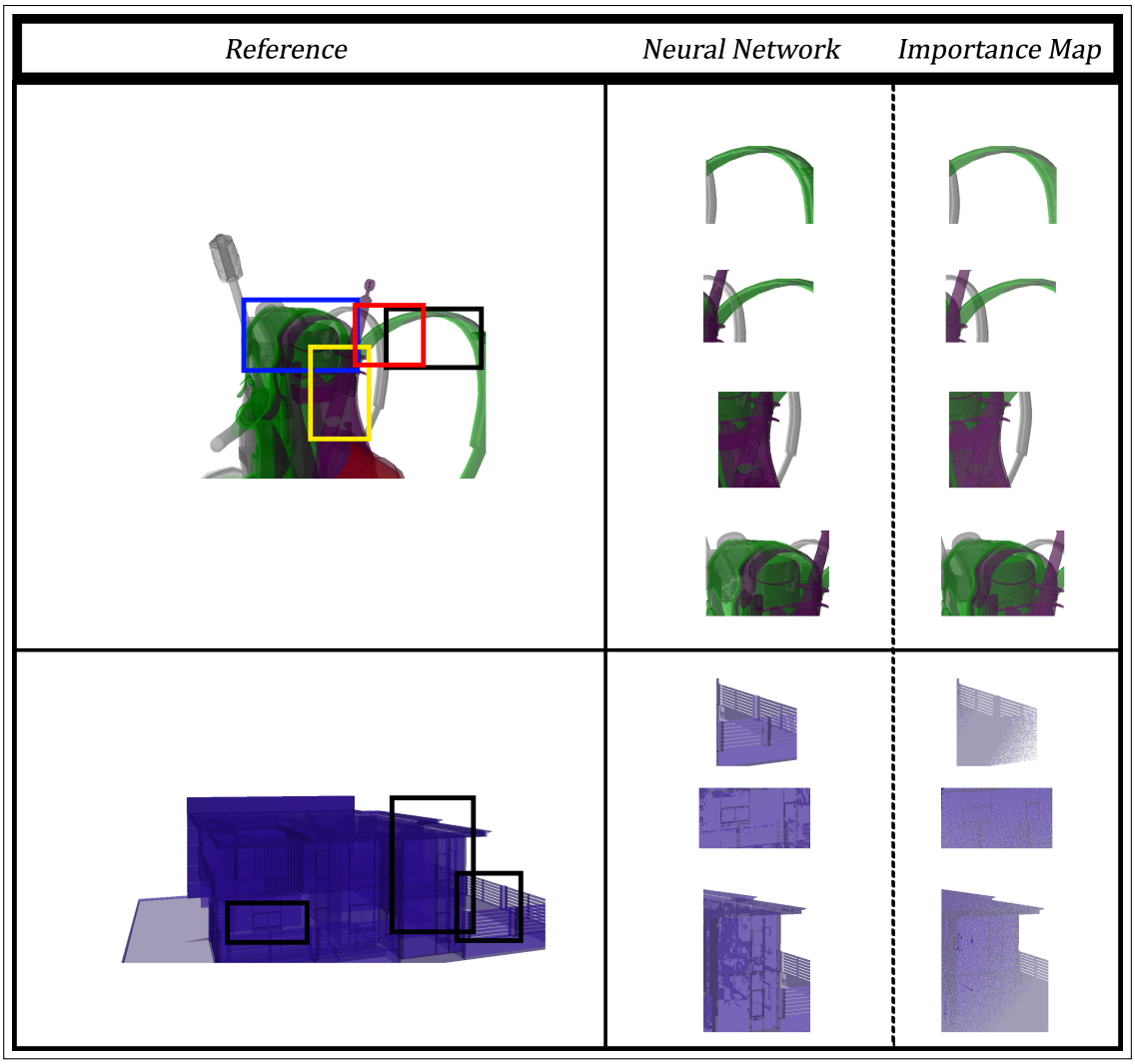

Figure 5.5: Quality evaluation between the two variable k-buffer approaches in two different scenes (backpack and house). Indicated areas, are areas where neural network offers a better result.

Chapter 6

Conclusion and Future Work

In this thesis, several multifragment rendering methods are implemented, experimented on, and the results are then briefly presented. The MFR method data structures and their shader implementations on current hardware and an MFR demo application are also presented. Additionally, a variable k-Buffer method, Deep Learning Variable k-Buffer, that assigns a variable per-pixel k, aiming for visual results closer to the A-Buffer's quality under a fixed memory budget is introduced. This approach assigns a per-pixel k according to a per-pixel importance predicted by a neural network. Finally, the neural network and the creation process of its features and training dataset are presented in a step-by-step process. Deep Learning Variable k-Buffer presents a quality closer to the A-Buffer compared to importance map approaches [VVPM17].

Further directions may be explored for tackling the problem of multifragment rendering using neural networks. While Deep Learning Variable k-Buffer lacks in certain respects (e.g. visual artifacts, slow performance), the results look promising and future work could further improve this method. The visual artifacts caused by vastly different adjacent predicted per-pixel k values should be further explored. Additional experiments showing the performance and quality differences between the various methods should be also conducted. Finally, another challenge is to improve performance by modifying the variable k-Buffer pipeline by using data from the previous frames.

Bibliography

[ANA+20] Pontus Andersson, Jim Nilsson, Tomas Akenine-Möller, Magnus Oskarsson, Kalle Åström, and Mark D. Fairchild. ꟻLIP: A Difference Evaluator for Alternating Images. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 3(2):15:1–15:23, 2020.

[ATI] ATI. ATI Mecha Demo Order-Independent Transparency. [Online; accessed July 30, 2021 at `https://gpuopen.com/archived/radeon-hd-5000-series-graphics-real-time-demos/`].

[BCL+07] Louis Bavoil, Steven P. Callahan, Aaron Lefohn, João L. D. Comba, and Cláudio T. Silva. Multi-fragment effects on the gpu using the k-buffer. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, I3D '07, page 97–104, New York, NY, USA, 2007. Association for Computing Machinery.

[Car84] Loren Carpenter. The a -buffer, an antialiased hidden surface method. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '84, page 103–108, New York, NY, USA, 1984. Association for Computing Machinery.

[Cat78] Edwin Catmull. A hidden-surface algorithm with anti-aliasing. In *Proceedings of the 5th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '78, page 6–11, New York, NY, USA, 1978. Association for Computing Machinery.

[CICS05] S.P. Callahan, M. Ikits, J.L.D. Comba, and C.T. Silva. Hardware-assisted visibility sorting for unstructured volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 11(3):285–295, 2005.

[Cra10] Crassin, Cyril. Fast and Accurate Single-Pass A-Buffer using OpenGL 4.0+, 2010. [Online; accessed July 31, 2021 at `https://blog.icare3d.org/2010/06/fast-and-accurate-single-pass-buffer.html`].

[Eve01] C. Everitt. Interactive order-independent transparency. 2001.

[GFD+12] Brian Guenter, Mark Finch, Steven Drucker, Desney Tan, and John Snyder. Foveated 3d graphics. *ACM Trans. Graph.*, 31(6), November 2012.

[Khr] Khronos Group. Online; accessed August 5, 2021 at `https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview`.

[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 *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '10, page 75–82, New York, NY, USA, 2010. Association for Computing Machinery.

[Mam89] A. Mammen. Transparency and antialiasing algorithms implemented with the virtual pixel maps technique. *IEEE Computer Graphics and Applications*, 9(4):43–55, 1989.

[McG17] Morgan McGuire. Computer graphics archive, July 2017. `https://casual-effects.com/data`.

[Sal13] Salvi, Marco. Advances in Real-Time Rendering in Games: Pixel Synchronization: Solving old graphics problems with new data structures. In ACM SIGGRAPH 2013 Courses (New York, NY, USA, 2013), SIGGRAPH '13, ACM, 2013.

[VF12] Andreas A. Vasilakis and Ioannis Fudos. S-buffer: Sparsity-aware Multi-fragment Rendering. In *Eurographics 2012 - Short Papers*. The Eurographics Association, 2012.

[VPF15] Andreas-Alexandros Vasilakis, Georgios Papaioannou, and Ioannis Fudos. k+-buffer: An efficient, memory-friendly and dynamic k-buffer framework. *IEEE Transactions on Visualization and Computer Graphics*, 21(6):688–700, 2015.

[VVP20] A. A. Vasilakis, K. Vardis, and G. Papaioannou. A survey of multifragment rendering. *Computer Graphics Forum*, 39(2):623–642, 2020.

[VVPM17] A. A. Vasilakis, K. Vardis, G. Papaioannou, and K. Moustakas. Variable k-buffer using importance maps. In *Proceedings of the European Association for Computer Graphics: Short Papers*, EG '17, page 21–24, Goslar, DEU, 2017. Eurographics Association.

[YHGT10] Jason C. Yang, Justin Hensley, Holger Grün, and Nicolas Thibieroz. Real-time concurrent linked list construction on the gpu. *Computer Graphics Forum*, 29(4):1297–1304, 2010.