INTERACTIVE RAY TRACING INFRASTRUCTURE

A Thesis

by

HAO LUO

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirement for the degree of

MASTER OF SCIENCE

Chair of Committee,	Ergun Akleman
Committee Members,	Tim McLaughlin
	Weiling He
Head of Department,	Tim McLaughlin

December 2019

Major Subject: Visualization

ABSTRACT

In this thesis, I present an approach to develop interactive ray tracing infrastructures for artists. An advantage of ray-tracing is that it provides some essential global illumination (GI) effects such as reflection, refraction and shadows, which are essential for artistic applications. My approach relies on massively paralleled computing power of Graphics Processing Unit (GPU) that can help achieve interactive rendering by providing several orders of magnitude faster computation than conventional CPU-based (Central Processing Unit) rendering. GPU-based rendering makes real time manipulation possible which is also essential for artistic applications. Based on this approach, I have developed an interactive ray tracing infrastructure as a proof of concept.

Using this ray tracing infrastructure, artists can interactively manipulate shading and lighting effects through provided Graphical User Interface (GUI) with input controls. Additionally, I have developed a data communication between my ray-tracing infrastructure and commercial modeling and animation software. This addition extended the level of interactivity beyond the infrastructure.

This infrastructure can also be extended to develop 3D dynamic environments to obtain any specific art style while providing global illumination effects. It has already been used to create a 3D interactive environment that emulates a given art work with reflections and refractions.

DEDICATION

To my families

ACKNOWLEDGMENTS

I would like to strongly express my appreciation to my committee chair, Dr. Ergun Akleman, and my committee members, Professor Tim McLaughlin and Dr. Weiling He, for their guidance, support and help throughout the course of conducting this research and writing this thesis.

Thanks also go to my friends and colleagues and the Viz Lab faculty, staff and alumni for making my time at Texas A&M University a memorable and extremely rewarding experience. My gratitude also extends to Sony Picture ImageWorks, Pixar Animation Studios, Electronic Arts and Nvidia Corporation for providing scholarships, research funding and instruments, and to all the instructors and students in Visualization Department who created such a wonderful study atmosphere and research environment.

Finally, I would like to acknowledge my mother, father, sister and brother-in-law for their endless support and encouragement and to my wife and my sons for their infinite patience and love.

NOMENCLATURE

2D	2 Dimensional
3D	3 Dimensional
API	Application Programming Interface
CG	Computer Graphics
CUDA	Compute Unified Device Architecture
GPU	Graphics Possessing Unit
MEL	Maya Embedded Language
NPR	Non-Photorealistic Rendering
OpenGL	Open Graphics Library
USB	Universal Serial Bus
VFX	Special Visual Effects
VR	Virtual Reality

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

In the realm of Computer Graphics (CG), artists have created shading and lighting effects often painstakingly went through long processes of texturing, shading, lighting and rendering sometimes even involved modeling. In order to create high quality image frames and/or photo-realistic imagery, in CG production, shading, lighting and rendering amongst those processes consume tremendous amount of computational resources. The majority of computing power are allocated to those rendering related processes and few are allocated to other equally important aspects with respect to the prevailing production pipelines. Recent years have seen an increase in demand for the efficient creation of higher image quality and faster rendering in CG production. An efficient rendering infrastructure is one that can greatly simplify the shading, lighting and rendering processes while creating accurate and high-quality imagery. In contrast, inefficient rendering infrastructure complicates the process and undermines the capability of generate high-quality image frames precisely.

Digital image frames are typically synthesized through traditional rendering methods such as offline ray trace based rendering. Offline rendering is where image format of frames are rendered, and the frames are displayed as a still image or animation (e.g. 24 frames output to display as 1 second of animation) [30]. Offline renderers produce high-quality images but are time consuming and costly. In CG, ray tracing rendering technique has become a standard due to the high demand of quality shading and lighting effects. Both photo-realistic and non-photo realistic rendering which based upon ray tracing techniques

are critically important for producing Special Visual Effects (VFX). Not only film and TV shows in which high-quality global illumination are required for the purpose of synthesizing photo-realistic imagery, but also in the field of artistic rendering where generating accurate non-photo realistic (NPR) images demands high-quality shading and lighting effects [24]. In both scenarios, either significantly more computing resources or an optimized rendering techniques are in need.

Interactive digital image frames are typically created through various real-time rendering methods such as scanline rendering. Real-time rendering is producing synthetic images and displaying it to the viewer onto the screen fast enough (preferably exceeding 24 frames per second) in a way that the user can pass input to the virtual environment interactively [1]. However, most real-time renderers are not ray trace based so they are unable to produce images that achieving quality so high as ray trace based rendering. To solve this conflict between quality and speed, a more desirable approach and an efficient rendering infrastructure is needed.

In this research, however, the possibility of combing high quality ray trace based rendering and interactive level of speed rendering has been explored. With that been achieved, the creation of shading and lighting effects can be iterated hundreds, if not thousands, times faster interactively. Upon real-time ray trace based rendering, the goal is to develop a graphical user interface (GUI) for 3D content creation artists that is intuitive and easy-to-use. Such that the 3D image rendering result can be easily directed by the artist who may be manipulating the shading or altering the ray tracing operations with direct and simple

interaction through user inputs in real-time. Therefore, this research aims at developing a ray-trace based renderer which is able to generate high-quality image frames efficiently. In the development, an interactive ray-tracer has been implemented which takes advantage of the massively paralleled computing capability of Graphics Processing Unit (GPU). In addition, a GUI has been created which renders and displays the result images, which also allows direct interaction with the virtual scene as well as shader through user inputs. Chethna Kabeerdoss has been successfully created a 3D scene based on a 2D art painting and rendered on top of my real-time ray tracing infrastructure. The shader applied to the 3D scene is derived from generalized Barycentric method and integrated smoothly in her art directed study of NPR with this ray tracer.

Data intercommunication between applications has also been established, this way, artists can create 3D content using existing 3D software packages and render the content of the virtual scene through the GPU based ray-tracer. In summary, this thesis is focused on providing such an interactive ray trace based infrastructure.

2 BACKGROUND AND RELATED WORKS

2.1 Rendering

In the realm of CG, rendering is the image producing process, from virtual objects in what described a scene file collectively, by means of computing programs. Models can be either 2D (2-dimensional), 3D (3-dimensional) or both. Rendering can also be called as the results of such models. Objects in a Strictly defined data structure or language is contained in the scene file. In which geometry object, texture, shader (material information), lighting, etc. would contained by the virtual scene as a description file. Digital images or raster graphics image files are outputted after the data described in the file of the virtual scene passed to and processed by a rendering program. An "rendering from artist" of a scene may be by analogy with the term "rendering" [11]. Though the pipeline of graphics with a rendering device (CPU or GPU) are outlined as the general challenges to overcome in rendering, technical details of synthesizing 2D images from 3D representations stored in scene description files vary substantially. The rendering equation should be solved by the rendering software if a scene under virtual lighting is to look relatively predictable and realistic, as shown in Figure 2.1; or non-photo realistic, as shown in Figure 2.2. Though rendering equation is a general lighting model in which image is computer generated, it does not account for all lighting phenomena happens in nature.

Rendering in practice is always connected to the others and is a computer graphics major sub-topic. As the final major step, in the graphics pipeline, it gives the final appear-

(a) A Studio Photograph (b) A Corresponding Rendering

Figure 2.1: An example demonstrating that it is possible to create image that is very close to studio photograph using state-of-art in rendering techniques, reprinted from [34]

Figure 2.2: Chinese ink-and-brush painting with reflection created using 3D computer graphic, reprinted from [16]

ance to the virtual scene. It has become a more distinct subject since the 1970s due to the increasing sophistication of computer graphics.

Rendering has uses in several different areas including animation production, video games, film or TV special visual effects, and simulations, though different balance of fea-

tures and techniques are employed each. A wide range of rendering software are available as a product. Several are free open-source, some are stand alone and some are integrated as part of larger 3D content creation software packages. Renderer is a program carefully engineered on the inside, which based on a multitude of knowledge related to: mathematics, visual perception, software development and light physics.

Rendering may be done quite slowly, in the area of 3D computer graphics, as in offline rendering (also known as pre-rendering), or in real time (might also be interactive rendering). Typically, offline rendering is used for animation and film or TV visual effects creation due to the process is computationally intensive, while real-time rendering is heavily rely on hardware acceleration from the use of graphics cards that is often done for video games.

2.2 Ray Trace Based Rendering

A method of producing computer generated imagery from 3D virtual environments described as optical ray tracing, which has more advantages than scanline rendering techniques in terms of photo-realism. The color of each pixel is calculated by extending a path from an virtual camera through a 3D screen contains objects, as shown in Figure 2.3.

Typically, all the objects in the scene must go through an intersection test with some subset. The algorithm is then to execute a series of functions to synthesize the resulted color of each pixel with combined information as soon as the nearest object has been found, those steps including gather light incoming at the intersection position, check the shading

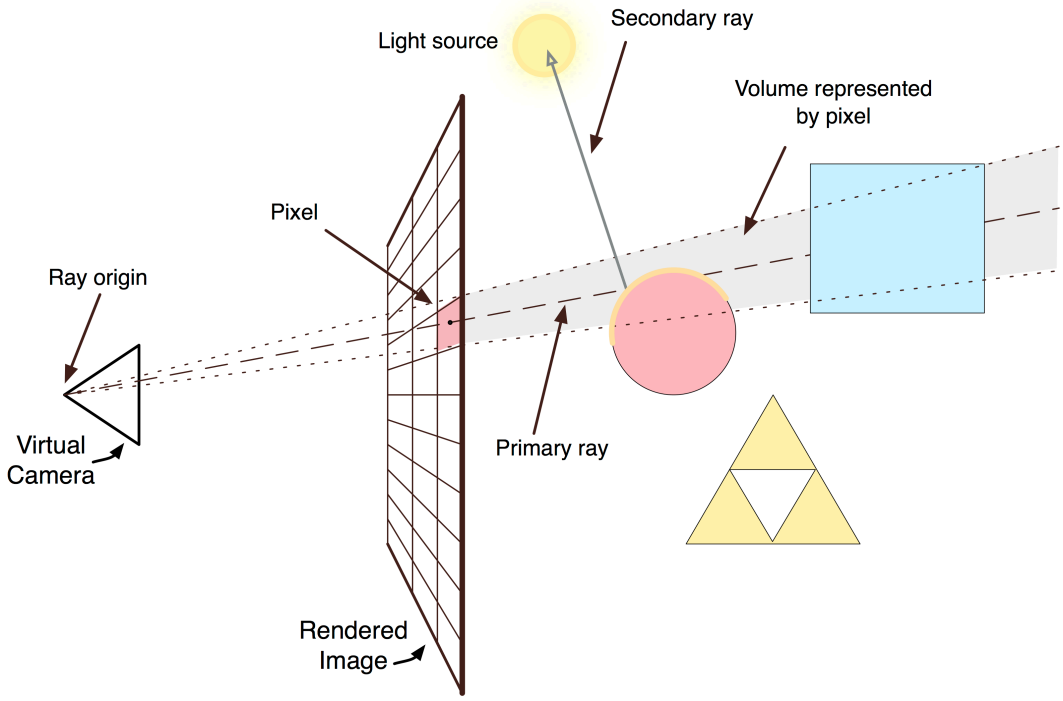


Figure 2.3: Image is built with ray tracing algorithm by sending rays into a scene.

information of the object. Re-cast more rays may be required by certain translucent or reflective materials and certain illumination algorithms.

Rather than send rays into the camera, send rays away from it (the opposite as light does in nature) is many orders of magnitude more efficient, though doing so may first seem counter-intuitive. Since majority of light rays do not pass directly into the eye of viewer from a given light source, meanwhile they are overwhelming, a tremendous amount of computational power could be potentially wasted on light paths that are never recorded in a “forward” light simulation.

Therefore, a presuppose of a given ray intersects the view frame is the shortcut taken in ray-tracing. The ray ceases to travel after either a ray traveling a certain distance and no

intersection is occurred or maximum number of reflections have reached, and then value of the pixel is updated.

2.2.1 What Happens in Reality

In reality, light rays emit by a light source, eventually, travel to a surface that interrupts their progress. Ignoring relativistic effects, these ray will be straight lines in a perfect vacuum. Four things might happen with these light rays regardless of combination: absorption, reflection, refraction and fluorescence [32]. The reflected and/or refracted light may loss intensity resulted by a surface absorbing part of the light ray. Part or all of the light ray might also reflected by the surface in one or more directions. While absorbing some or all of the spectrum a portion of the light might refracts into the object itself in different directions if transparent or translucent is of its properties. In some rare scenario, some portion of the light may be absorbed by a surface and other portion is re-emit in a random direction at a longer wavelength, described as term fluorescence. There are four and only four possible kinds of light interference accounted for all of incoming light. For instance, 30% of an incoming light ray is absorbed, and less or equal to 70% of is reflected, a surface cannot interfere more than 100% of a light ray. From there, the rays that are reflected may strike other surfaces, where again, the progress of the incoming rays affected by their absorptive, refractive, reflective and fluorescent properties. Our eye is then, by chance, hit by some of these rays travel this way and thus contribute to the image we finally see.

2.2.2 Algorithms of Ray Tracing

Arthur Appel presented the first algorithm of ray tracing used for rendering in 1968 [3]. The term "ray casting" has since been used for this algorithm. Detect the closest object in the path of a ray shoot from the eye, one per pixel, is the idea behind ray casting. Each pixel is a square in a screen if think of an images as the screen-door. That pixel the eye sees through is then the object. Arthur Appel's algorithm can dictate the shading of this object using the effect of the lights in the scene and the properties of the material. Light will not be blocked or in shadow and reach a surface if it faces the light made as the simplifying assumption. Traditional 3D computer graphics shading models is used to compute the shading of the surface. The ability to deal with solids and non-planar surfaces easily offered by ray casting was an important advantage over scanline algorithms, for instance, spheres and cones. It can be ray casting rendered if intersection can be made by a ray with a mathematical surface. Solid modeling techniques can be used to create elaborate objects and thus the object can be easily rendered.

Turner Whitted introduced the next important breakthrough of research in 1979. Previous algorithms determine color of the ray without casting more rays recursively in which ray is traced from the camera into the scene until an object is hit. The process continued by Whitted. More rays can be generated (up to three types) when a ray hits a surface, they are: reflection, refraction, and shadow rays [33]. Mirror-reflection direction is where a reflection ray traced, what will be seen in the reflection is the intersection of the closet object it hits. With the addition of entering or exiting a material, refraction rays work sim-

ilarly when traveling through transparent material. Shadow rays are extended toward light sources. If any object is detected between the light source and the surface, the light does not illuminate the surface and it is then in shadow [19]. More realism is added to ray traced images by this recursive ray tracing technique.

2.2.3 Advantages vs. Disadvantages

The popularity of ray tracing becomes prevailing due to its realistic lighting simulation over other imaging synthesize algorithms (i.e. ray casting or scanline rendering). Like it's difficult to simulation using other algorithms, phenomenons like shadows and reflections are direct result come with ray tracing algorithm. Ray tracing is also amenable to parallelization due to its computational independence of each ray [18][6]. There are other advantages come with this independence, such as the ability to improve image rendering speed using spatial bounding volume hierarchy.

However, performance is a serious disadvantage of ray tracing. Data coherence is used by scanline and other algorithms to share computations between pixels, while the process in ray tracing starts afresh, normally, calculating each ray independently.

Although, inter-reflection and optical effects such as refraction does handled by conventional ray tracing accurately, it is not necessarily photo-realistic. The rendering equation is fully implemented or closely approximated when true photo-realism occurs. As every physically based property of light describes by the equation, photo-realism is given by the implementation of the rendering equation. Given the computing resources required, however, this is almost infeasible.

2.3 Rendering Device

Image rendering is a series process of calculations. Those calculations are specified in the computer program which is a compilation of lines of computing language. The computer program will operate functions at runtime to calculate colors of each pixel and carry out instructions to display rendered images. Different programs can run on either CPU (central processing unit) or GPU depending on their purposes, however, their efficiencies are different. In this research project, the program fully takes advantage of computing efficiency from GPU.

2.3.1 Central Processing Unit

Within a computer, a CPU is the electronic circuitry which operates serials of instructions from a program by performing specified basic logical, arithmetic, I/O (input/output) and control commands. The fundamental operation of CPUs remains almost unchanged, although their implementation and design have changed over the course of their history. Modern CPUs have multiple cores, though, each of the cores is still designed for processing complex serial instructions. CPUs with more than one core, ideally, are able to run an application few times faster.

2.3.2 Graphics Processing Unit

A GPU is a electronic circuitry, which is special designed to alter and manipulate memory rapidly to accelerate the process of producing images in frame buffers which are going to be used for output to a display. It is a device built on purpose of being able to

accelerate the process in calculating complex rendering. Modern GPUs are much more efficient than CPUs in terms of processing large blocks of visual data which can be processed parallel due to their highly independent structure. Thus, they are highly efficient at processing computer graphics and image manipulating.

2.3.3 CPU vs. GPU

In order to understand what makes the difference between a CPU and GPU, a comparison between the way they process tasks is needed. There are only few cores consisted in a CPU, which are optimized for sequential serial processing. Whereas, there are thousands of smaller cores consisted in a massively paralleled architecture of a GPU, as shown in Figure 2.4, which are designed significantly more efficient for simultaneously handling multiple tasks.

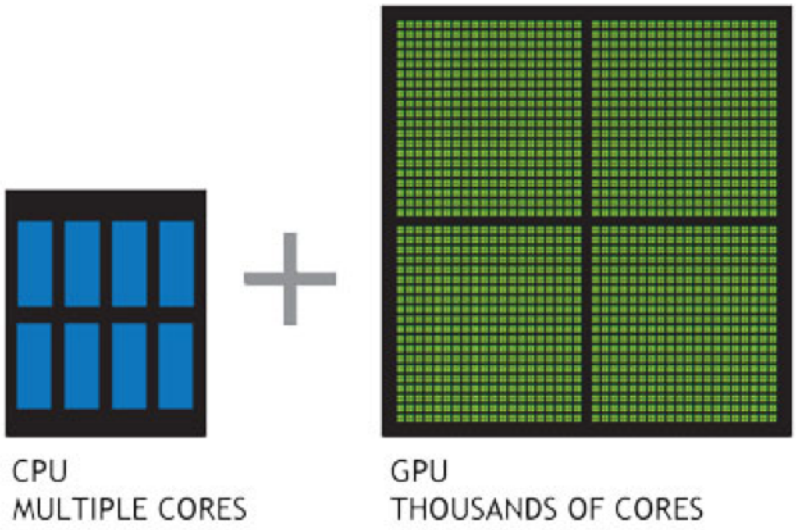


Figure 2.4: This comparison image shows a CPU has few cores whereas a GPU has significantly more cores to process parallel workloads efficiently by NVIDIA, reprinted from [20]

2.3.4 How GPUs Accelerate Applications

The portions of the application that are compute-intensive are offloaded into the GPU by GPU-accelerated computing, while the CPU still process remainder of the code so that it offers unprecedented application performance, as shown in Figure 2.5. Applications simply run many orders of magnitude faster from the perspective of a user.

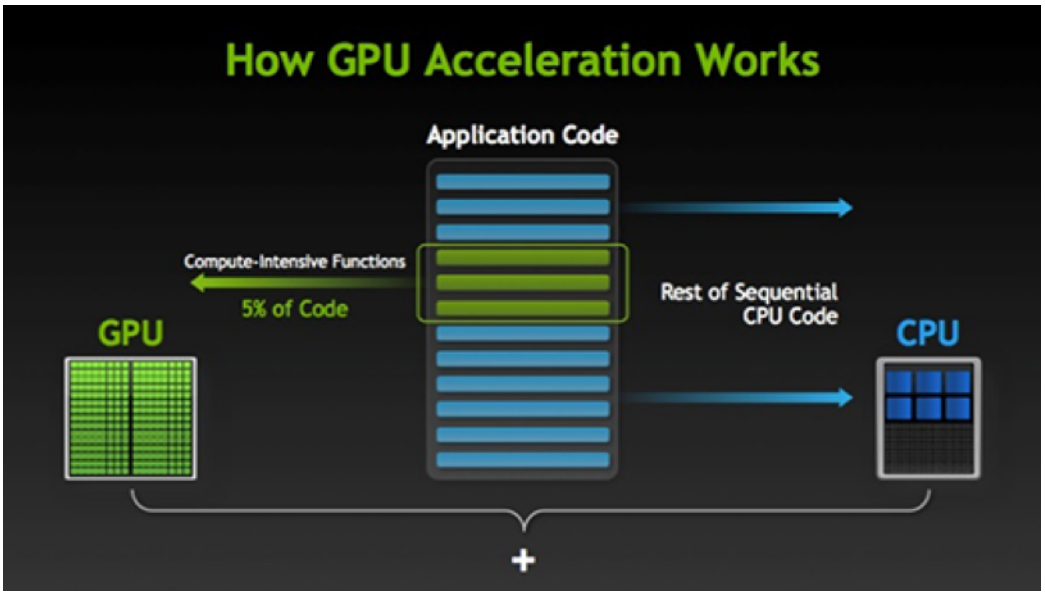


Figure 2.5: Example image shows how GPU acceleration works by NVIDIA, reprinted from [20]

2.4 External GPU

Computers have tens of thousands of variations in terms of hardware configuration. Some have GPU some do not. Most laptops do not have discrete GPU configured for general purpose computing. Although, few of them have discrete GPU, those GPUs are mobile-size that they are only capable of low intense graphics processing. Their comput-

ing power cannot even compare with desktop sized GPUs that are capable of handling highly intensive computation. In order to make the research infrastructure multi-platform compatible, the approach is to have an external GPU rig setup.

2.4.1 Connect External GPU

In the research project, an external GPU is connected to a MacBook which has support of Thunderbolt port. Thunderbolt is an I/O technology; it has been first featured by Apple in 2011. Thunderbolt port is essentially a interface of hardware that allows the external peripherals connect to a computer. Thunderbolt combines DisplayPort and PCIe (Peripheral Component Interconnect Express) into one serial signal. Giving access to PCIe is critically important to leverage the massive parallel computing power from workstation GPU for efficient ray tracing, PCIe used to be a high-speed serial computer expansion bus standard for connecting a computer to one or more peripheral devices, GPU is one of the devices that uses PCIe for data communication [14]. For laptops, normally there is no direct access to PCIe but majority of them provide Thuderbolt port. Therefore, the only way to connect a workstation GPU to a computer is essentially through Thunderbolt port.

Since the ideal way of connecting GPU to a computer is through thunderbolt, a device which provides direct access to thunderbolt and PCIe is desirable. However, based on current market of the industry, such a device might not be a feasible solution due to the licensing requirements of Thunderbolt standard for any companies that intend to offer interfacing with the connection standard. Intel owns Thunderbolt and refuses to give out the proper licensing/certification to companies who wish to produce such PCIe to thunderbolt

products. Therefore, the limitation has kept us from the possibility of direct connection between external GPU and computers through thunderbolt.

With further exploration, a company named Villagetronic (formerly known as Village Instruments) is found which located in Shenzhen, Guandong, China. In this electronic devices company, one of the products they produce called ViDock provides a fair clean solution for external GPU. ViDock is basically an enclosure in which contains PCIe x4 slot and PSU (power supply unit). The enclosure also provides an ExpressCard /34 (ExpressCard 34mm) to PCIe interface. ExpressCard, initially called NEWCARD [25], is an interface to connect peripheral devices to a computer. In ExpressCard standard, the host device supports PCIe through the ExpressCard slot. Numerous devices can connect to a computer through ExpressCards, for example, USB connectors, Ethernet network port, external enclosures for desktop-size graphics cards, solid-state drives, etc. This ViDock enclosure is then used as the housing for the external GPU, therefore, the external GPU is connected directly to a computer which has ExpressCard/34 slot, or to an adapter which has ExpressCard to thunderbolt interface and then connect the adapter to a computer through thunderbolt. Example of external GPU connection rig is shown in Figure 2.6.

2.4.2 Install Driver for the External GPU

In the research project, Yosemite (Mac OS 10.10) is the version of the working operation system. Apple added a new security feature to this OS (operating system) version: a driver signing. Driver that is modified manually is no longer valid. Therefore, a command line is typed in the terminal to enable the loading of unsigned drivers which is necessary

(a) (b)

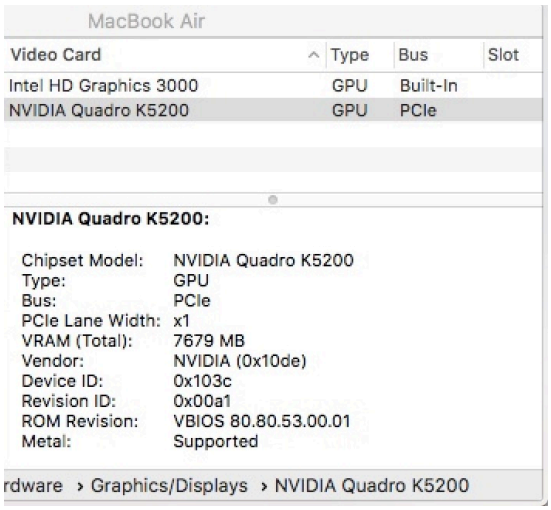
Figure 2.6: A Desktop-size GPU is housed in ViDock enclosure as shown in (a). The enclosure is connected to Sonnet Echo Pro ExpressCard/34 Thunderbolt adapter through ExpressCard, the adapter is then connected to a Macbook through Thunderbolt port as shown in (b).

(details in Appendix A.1). Nvidia drivers need to be installed in order for the GPUs to work properly. The Distribution file in the installation package of the GPU driver is first modified (details in appendix A.2) due to Mac OS System Integrity Protection prior to the installation setup [4]. Secondly, the driver is then installed through the installation package such that the operating system recognizes the GPU device once connected.

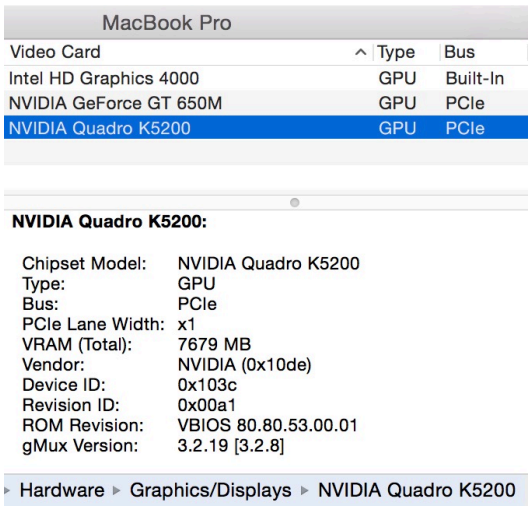
2.4.3 Hijack Thunderbolt-Compatibility

By default, Thunderbolt GPU drivers are not supported by Mac OS (Macintosh operating systems). The reason being, normally, there is such a demand for drivers adapted to hot-plug to functioning on Thunderbolt, but GPU drivers are not. Therefore, modifications are made in the pilots in the system to make the GPU drivers compatible with Thunderbolt,

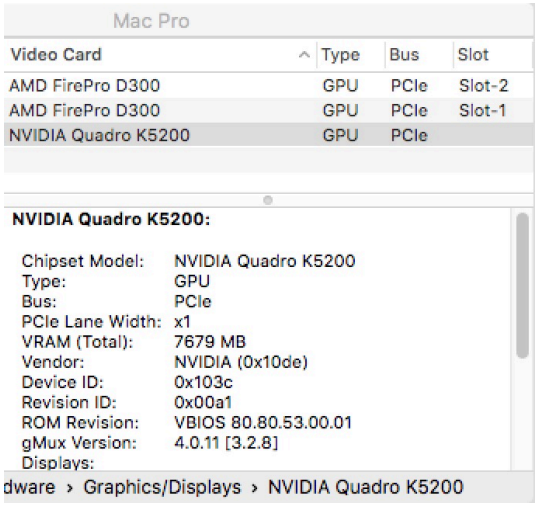
regardless of the hot-plug is not supported. For Nvidia graphic cards, there are three files to edit (details in Appendix A.3) [22]. Once the modifications are done in the three files, reboot the system and the card will be recognized once plugged in and functional. In the end, two different GPUs, GeForce GTX 970 (consumer version) and Quadro K5200, from Nvidia are successfully tested on multiple machines with this setup, as shown in Figure 2.7.



(a) eGPU on MacBook Air



(b) eGPU on MacBook Pro



(c) eGPU on Mac Pro

Figure 2.7: Example of external GPU connecting with different Mac machines.

3 METHODOLOGY

The target system can be divided into five major components, they are: scene initialization, GPU program generation, render and display images, graphical user interface and data intercommunication. This chapter describes the concepts and methods utilized to design and construct these five components. The general relationship between them is illustrated as in Figure 3.1.

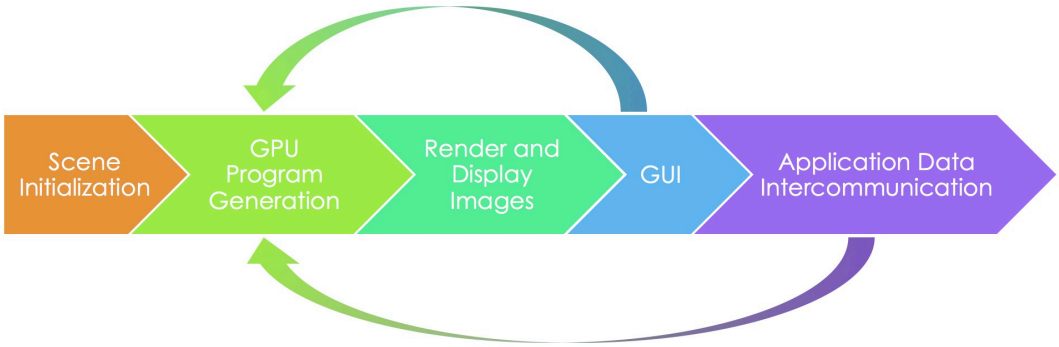


Figure 3.1: Overall relationship between components of target system

First, a 3D scene is initialized to represent the virtual environment to be rendered and displayed onto the screen. Meanwhile, GPU programs, such as shading, are created and attached to each associated object in the virtual scene such that heavy duty calculations are done with the assist of the GPU in a massively paralleled manner. Then, generated image data is offloaded to the image displaying pipeline and rendered onto the screen so viewers can see the 2D representation of the 3D virtual scene. In order to be able to intuitively interact with the virtual environment, a easy-to-use GUI is created which takes user input

as well as gives display update and feedback to user. Finally, data communication is established between the renderer and third-party 3D content creation software so that inputs can be further fed from within various 3D content solutions.

3.1 Scene Initialization

In rendering, the final result is images that generated from a scene description which is a strictly defined in the form of data structure, in which contained information of geometry, material, lights, etc. The data structure is a representation of the 3D virtual scene in terms of computer graphics. Then the data will be processed through rendering pipeline following with ray-geometry intersection test, shading result calculation, shadow casting, etc.

During the initialization of the virtual scene, geometries are created which define characteristic of the virtual objects containing information such as shape, position, orientation, etc. Shaders are then created and applied to the corresponding geometry each to define the material property of the object, for example, reflective or non-reflective surface. In order to have global illumination effects [23], lights and environment map are also created. All of these data is then passed onto ray tracing engine, which calculates the final appearance of the virtual environment pixel by pixel based on information of viewing camera, geometric and shading properties of virtual objects, lighting conditions and environmental setup. Where shading properties dictates whether the rendering is photo-realistic or NPR.

One of the greatest advantage of ray trace based rendering is its computational independence of each ray. This independence gives the ability to optimize data using spatial bounding volume hierarchy, also know as acceleration data structure. Another advantage comes with this independence is its amenable to parallelization. GPU is the perfect device for paralleled computation, we will further discuss it in the next section. Although, scene initialization phase is completed in CPU program. All the data processing, such as constructing acceleration structure, is done on CPU.

3.2 GPU Program Generation

Due to the independence feature of ray tracing, each ray can be generated individually. Hence, in order to utilize the massive parallel computing capability of GPU, ray generation and shader programs are created specifically to be processed by GPU. In other words, the ray tracing engine program is GPU based.

A ray tracing engine is the ”heavey lifter” of the infrastructure, it is responsible for all the heavy-duty calculations such as ray-geometry intersection detection, texture and shading results, lighting effects, global illumination, etc. In order to take advantage of massive parallel computing power of GPU, the engine program of the infrastructure has to have the ability to be executed on GPU which also renders high quality imagery.

Usually, there are numbers of component in a ray tracing engine. Launch of ray tracing engine starts with ”shooting” rays from virtual camera into the virtual scene, following with intersection test, texture look-up, recursive ray-tracing, etc., and ends with

”projecting” captured image onto the screen. In this research project, however, ray tracing engine is divided into two major components due to its GPU based features. They are: ray generation and shader program.

3.2.1 Ray Generation

Rays are generated based on the attributes of the virtual camera. For example, origin and general direction. Number of rays to be generated is defined based on the setting resolution of the final image and anti-aliasing sample. Before the ray tracing engine starts, scene data, such as camera, geometry, shader, light, etc., is processed and constructed by CPU and then offloaded to GPU. A massively paralleled ray generation sets off on GPU at this point. Each ray is generated based on the same set of scene data and carrying payloads, like color and depth. But each ray is targeting at different pixel, which is what makes each ray ”shooting” at different directions individually. The following calculations like ray-geometry intersection test are done in parallel as well.

Artistic rendering effects are developed as early as ray generation stage. In photo-realistic rendering, rays are generated regularly and their directions are slight different from one to the other. The final image is then constructed by those pixel colors one by one which mimics the nature results. However, in NPR rendering, all rays can be altered by an artistic algorithm, whether it is ray origin or ray ”shooting” direction. The alteration can be done completely irregular or random to achieve certain artistic effects. This alteration is called, in this research, artistic camera lens or NPR camera rendering.

3.2.2 Shader Program

Shaders are computer programs that are used to calculate shading effects. Shading means instructions of drawing something. High degree of flexibility is the advantage of shaders that synthesize imaging effects.

The GPU programs provide control over ray-geometry intersection, shading, and other general computation. Control over the shader, which is the shading program, is the main focus over the course of the development for this infrastructure. The interaction with shader is achieved based upon the basic control of the shader program. Resulted image can be modified as render happening by altering attributes such as color, saturation, value of all pixels, vertices, or textures in the shader using algorithms. Textures or variables defined by the program can be used to access those attributes. Infinite effects can be produced by shaders which are also used widely in NPR shading and rendering.

In the end of the ray tracing engine, each pixel colors are calculated and stored as payload in corresponding rays. GPU buffer of the final image is created at this point to receive data regarding pixel colors. Resulted image data is then pass onto CPU program through buffers for rendering and displaying.

3.3 Render and Display Images

The final stage of a non-interactive graphics pipeline is rendering and image displaying. This is not the case in the research project, although, the mechanism for image displaying is the same. Rendered image frames are projected onto the display which is what

the viewer eventually sees. The rendering and displaying are completed within the infrastructure, therefore, the application window is responsible for this task which requires the integration between the two. In this section, the concept and method for image rendering and displaying within the application are discussed.

Image data that is received from GPU is processed and stored in a GPU buffer format. When it is passed to be processed by CPU program, remapping is usually needed. Remapping is done based on pixel locations of displaying. Each pixel data is stored in a unique location of the image buffer. Pixel data of each location is remapped to a displaying buffer by CPU program in a way that the displaying program can read the displaying buffer and display the image correctly.

3.4 Graphical User Interface

Another major key area of the interactive infrastructure is the GUI. With the development of GUI, users can interact with application intuitively through the interface displayed to them. Each component's attribute were provided for changing shader program such that artists can manipulate their image through a intuitive interface. In terms of photo-realistic or NPR, it's up to the choice of the users.

3.5 Applications Data Intercommunication

In this research, the ray tracing infrastructure not only provides various methods of interactivity, but a form of scalability is also developed through the establishment of data communication with other software applications. One possible scenario of the scalable

infrastructure is that an artist creates 3D digital content within other third party software, during the course of content creation, any changes the artist made to the 3D content are rendered and displayed instantly by the ray tracing program. This allows the artist focus on the process of content creation with huge advantage of instant iterations as well as choosing the preferred 3D content creation software.

4 IMPLEMENTATION

4.1 Create the 3D Scene

In rendering, the final result is images that generated from a scene description which is a strictly defined in the form of data structure, in which contained information of geometry, material, lights, etc. In the scope of the programming, each of these aspect is described as object. Although, due to the unique features of each object, the initialization process takes various different steps to complete. In this section, the concept of each programming object and integration of OptiX API (application program interface) are elaborated [21], however, the creation of each OpitX program will only be further discussed in the section 4.2.

4.1.1 Setup Context

An interface for controlling the setup and subsequent launch of the ray tracing engine is provided by an OptiX context. Therefore, context object is first created which encapsulates all OptiX resources such as geometry, shader, lights, buffers, etc. It is last destructed which cleans up all these resources and invalidate any existing handles to them.

An entry point to ray engine computation is served with context launch function which takes an entry point parameter, discussed in section 4.1.1, as well as one, two, or three grid dimension parameters. The dimensions establish a logical computation grid. The context launch function performs any necessary preprocessing and then invokes the ray

generation program associated with the provided entry point index once per computational grid cell. State validation and acceleration structure generation and kernel compilation is included in the launch precomputation. In the end, OptiX buffers is used to pass output back from the launch.

A single context object can leverage multiple hardware devices, therefore, multiple context objects is not needed. By default, the highest compute capable set of compatible OpitX-capable device is used [13].

Supply Entry Points

Multiple computation entry points are set to the context object. A single ray generation program as well as an exception program is associated with each entry point of the context object. A ray generation program must be assigned before each entry point can be used; however, exception program that allows users to specify behavior upon various error conditions is an optional program. Therefore, switching between multiple rendering algorithms as well as efficient implementation of techniques such as multi-pass rendering on a single OptiX context is allowed by such a multiple entry point mechanism [13].

Determine Ray Types

In order to distinguish between rays that are traced for different purposes, OptiX provides useful supports for the notion of ray types. For instance, rays used to compute color values and rays used exclusively for determining visibility of light sources (shadow rays) are distinguished in terms of type and efficiency. Proper separation increases program modularity and the efficiency of operation of renderer. Both the number of different ray

types as well as their behavior is entirely defined by the application.

4.1.2 Create Buffers

Buffer objects are used to pass data between the host and the device, in another words, between CPU and GPU. Prior to invocation of context launch, host creates buffers using buffer creation function. Buffer type as well as optional flags are set by this function. The Direction of data flow between host and device is determined by buffer type. Size, dimensionality and element format is specified before using the buffers.

Access of data stored within a buffer to host is performed with the buffer mapping function. A pointer to a one dimensional array representation of the buffer data is returned by this function. Before context validation, all buffers are unmapped through buffer unmapping function. Device programs access the data stored in a buffer, on the other hand, using a simple array syntax.

4.1.3 Load Textures

Common texture mapping functionality is supported by OptiX textures including texture filtering, wrap modes, and sampling. Texture object is created using texture sampler create function. One or more buffers containing the texture data is associated with each texture object. The buffers can be set with texture sampler set buffer function and they may be in form of different dimensionalities from 1D to 3D.

4.1.4 Assemble Hierarchical Graph Node

A node is given that specifies the root of the graph when a ray is traced from a program using the ray tracing function. This graph object is assembled in the host application by feeding various types of nodes provided by the OptiX API. Hierarchy is the basic structure of the graph, as shown in Figure 4.1, with nodes describing collections of objects at the top, and geometric objects at the bottom. The following sub-sections will describe the individual node types.

A scene graph in the classical sense, however, may not be described by the graph structure in this case. Instead, a way of binding different programs or actions to portions of the scene is served by this graph structure. Different trees or sub-trees may be used due to each invocation of ray trace function specifies a root node. For instance, for performance or for artistic effect, reflective objects may use a different representation then shadowing objects.

Context object is taking as a parameter upon creation of graph nodes through create function calls. However, graph node objects are owned by the context, rather than by their parent node in the graph. Destroy function calls delete variables of objects, but reference counting or automatic freeing of its child nodes remains unprocessed.

Build Geometry

The fundamental node to describe a geometric object is a geometry node. In the rendering program, geometric object is essentially a collection of user-defined primitives against which rays can be intersected.

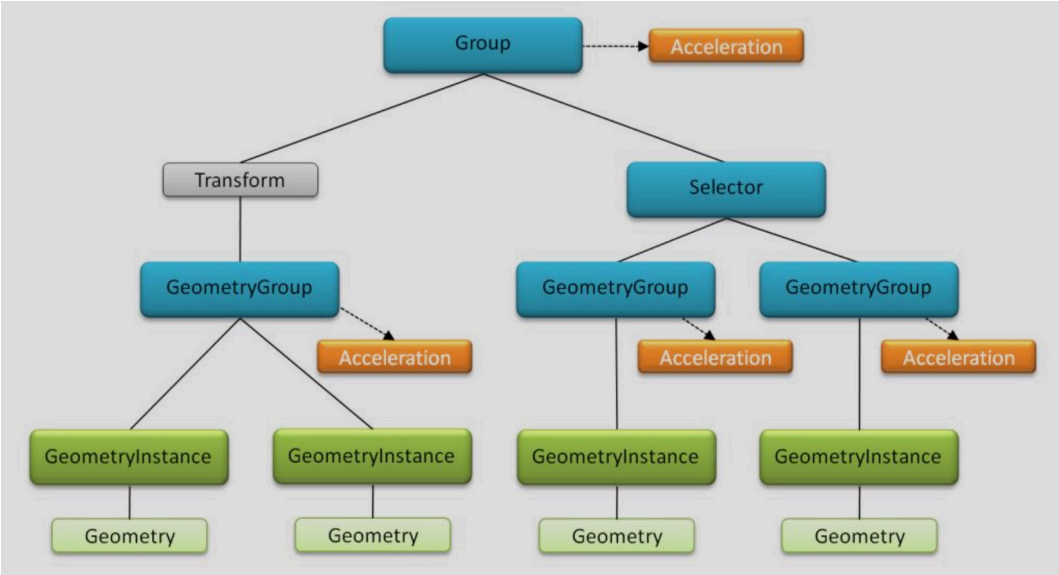


Figure 4.1: Hierarchy example graph.

An intersection program is assigned to the geometry node using set geometry intersection program function in order to define the primitives. A primitive index and a ray are fed as input parameters to an intersection program, and the job of the program is to return the intersection between the two. The necessary mechanisms to define any primitive type that can be intersected against a ray is provided by this program in combination with program variables. For example, the intersection program reads vertex data of a triangle out of a buffer which is passed to the program through a variable, and performs a ray-triangle intersection. Ray-triangle intersection is the mostly commonly used method in arbitrary geometry object cases.

It is necessary to provide a separate bounds program in order to build an acceleration structure over arbitrary geometry. This is completed using set geometry bounding box program function, which simply computes bounding boxes of the requested primitives,

which are then used as the basis for the construction of acceleration structure. The dynamic feature of geometry nodes is achieved by notification to the acceleration structures that containing references to the modified ones, with the function call of mark geometry dirty. This is further discussed in section 4.1.5.

Attach Material Program

A key area in this research, is to provide interactive access to material program. When a ray intersects a primitive associated with a given material, actions are taken which are encapsulated in a material object. Example actions including reflectance color computation, additional ray tracing, intersection ignorance, ray termination. Declaring program variables is used to provide arbitrary parameters. Two types of device programs are assigned to a material they are: any hit and closest hit programs. When and how often they are executed are the major difference of the two types.

Similar to a shader in a classical rendering system, the closest hit program, for the closet intersection of a ray with the scene, is executed at most once per ray. Typical actions performed involve reflectance color computations, texture lookups, light source sampling, recursive ray tracing, etc., and the results are stored in ray payload data structure.

During ray traversal, the any hit program is executed for each intersection found. The program executes all intersections with the scene, may not be ordered along the ray, are eventually enumerated if required by calling ignore intersection function on each of them. Early termination of shadow rays is a typical use of the any hit program.

Geometry Instance and Group

A coupling of a single geometry node with a set of materials is represented by a geometry instance. Set geometry instance function is used to specify the geometry object that the instance object refers to. Set instance material function is used to assign individual materials. Multiple geometry instances are allowed to refer to a single geometry object, enabling instancing of a geometric object with different materials [13]. Likewise, materials can be reused between different geometry instances.

A container for an arbitrary number of geometry instances is used as a geometry group object in the program, instances are assigned using geometry group set child function. An acceleration structure is also assigned to each geometry group with geometry group set acceleration function.

A collection of higher level nodes in the graph is represented by a group object. A group node is eventually passed to tracing function for intersection with a ray, which is used to compile the graph structure. An acceleration structure must also be assigned to every group.

Utilize Transform

A projective transformation of the underlying scene geometry is represented using a transform object. The assignment of the transform must be exactly one child type which involves group, geometry group, transform. That is, the nodes below a transform node may simply be geometry in the form of a geometry group, or a whole new subgraph of the scene [13].

4.1.5 Assign Acceleration Structure

An important tool for speeding up the traversal and intersection for ray tracing is acceleration structures, a key to interactive rendering in this case, especially for large scene databases. Hierarchical decomposition of the scene geometry is the major feature of a successful acceleration structures. Regions of space not intersected by the ray is then quickly culled using this hierarchy.

Many different types of acceleration structures can be used, each with their own advantages and disadvantages. Absolute optimal acceleration structure for all scene does not exist due to one scene can be optimized using one, whereas, another scene can be optimized by one another. Construction speed vs. ray tracing performance is the most common trade-off in respect of acceleration object. For instance, it can take minutes to construct the acceleration structure of a high quality SBVH (spatial split bounding volume hierarchy) builder. Once finished, however, rays can be traced more efficiently than with any other types of acceleration structure, which take less time to build.

Acceleration Objects in the Node Graph

Acceleration object is created with acceleration create function, once created, it is assigned to either a geometry object or a group object. Acceleration object has to be assigned to every geometry group and group in the node graph for ray traversal to intersect those nodes.

When assembling the node graph, the application has a high level of control over how acceleration structures are constructed over the scene geometry by making use of

geometry groups object and group object. There are a number of ways to place several geometry instances, for example, in geometry groups or groups in a scene to fit the use case of the application.

Figure 4.2, as an example, shows geometry instances are placed in a single geometry group. An acceleration structure is constructed over the individual primitives on a geometry group, which is defined by the collection of child geometry instances. Therefore, an acceleration structure is built as efficient as if the geometries of the individual instances had been merged into a single object.

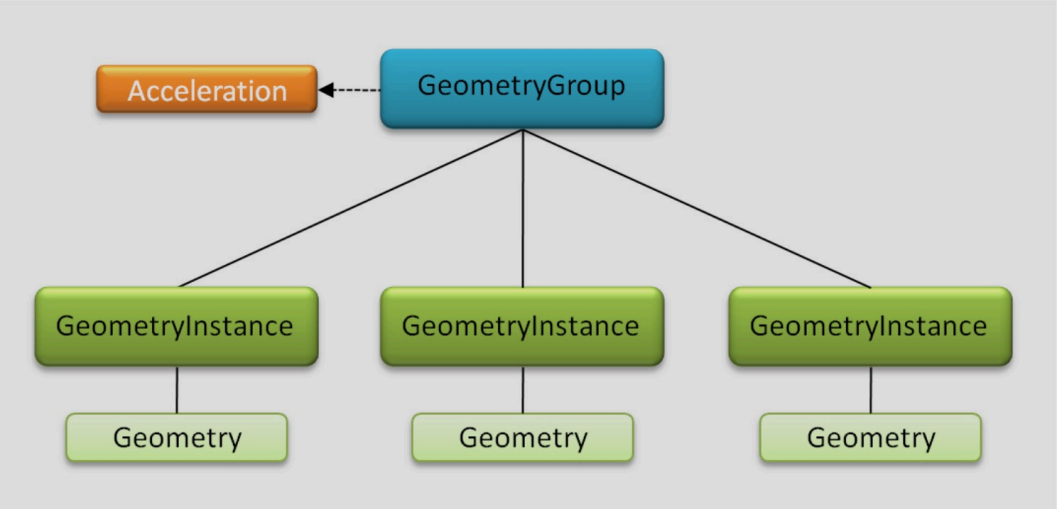


Figure 4.2: Example of geometry instances placed in a single geometry group node.

Figure 4.3 indicates a different approaching in terms of managing multiple geometry instances. Each geometry group placed with its own instance, therefore, each acceleration structures are assigned to each geometry instance. A top level group is then aggregated with the resulting collection of geometry groups, which there is an acceleration structure assigned to itself. Acceleration structures on groups are constructed over the bounding

volumes of the child nodes [13]. High level structures are typically quick to update due to the number of child nodes is relatively low, more often than not. This approach has the advantage that when modifying one of the geometry instances, only its associated acceleration structure needs be rebuilt rather than the acceleration structure of the entire graph object. However, because such approach introduces an additional level of complexity, the ray traversal is likely not as effective as the one presented in previous approach. Therefore, how frequently individual geometry instance is to be modified, in this case, is the main consideration of the trade-off the application needs to balance.

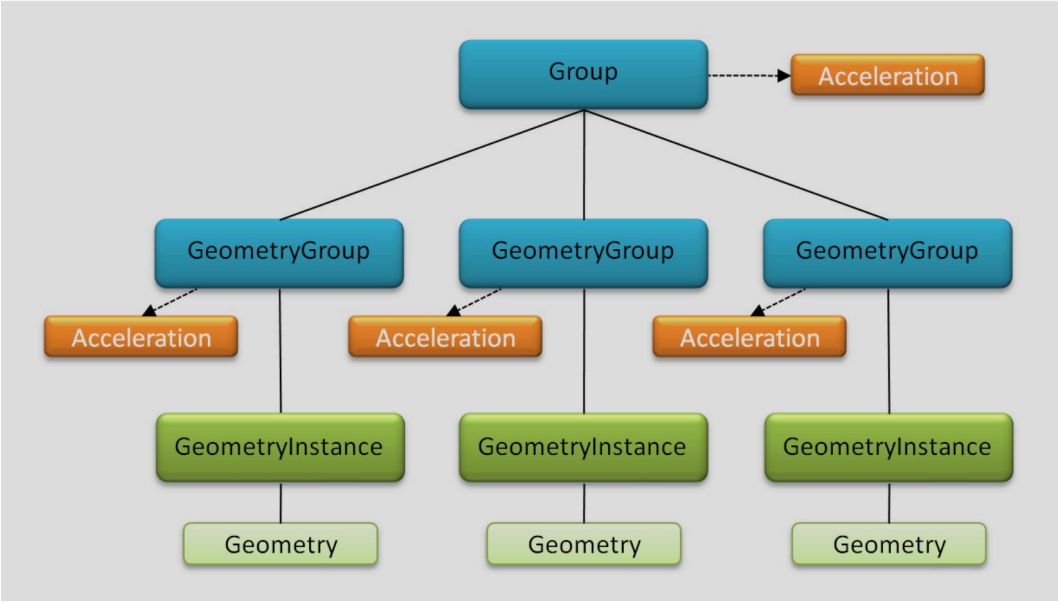


Figure 4.3: Example of multiple geometry groups place with its own geometry instance in each.

Builder and Traverser

A builder and a traverser are consisted within the spectrum of acceleration program. Collecting input geometry which is, in most cases, the bounding boxes created by

the bounding box program of geometry objects, and computing a data structure are jobs that the builder responsible for. Meanwhile, a traverser is to accelerate a ray-scene intersection query within such computing data structure. Both builders and traversers are not application-defined programs [13]. Instead, an appropriate type of each, which are also correspondence of each other, are chosen by the application. The builder is set by acceleration set builder function, and the corresponding traverser, which must be compatible with the builder, is set with acceleration set traverser.

Acceleration Structure Builds

In OptiX framework, acceleration structures, when they need to be rebuilt, are flagged as marked “dirty”. All flagged acceleration structures are built before ray tracing begins during the launch of ray tracing engine. Every newly created acceleration object is initially flagged for construction as well.

Any time during the launch, mark an acceleration structure for rebuild can be done explicitly decided by the application. For instance, an underlying geometry of a geometry group changes, rebuild is required to the acceleration structure attached to the geometry group. Acceleration mark dirty function is used to achieve this. If new child geometry instances, for example, are added to the geometry group this must also be done. The same is true for acceleration structures on groups.

4.2 Create GPU Program

In this section, a GPU based ray tracing framework has presented that support real-time rendering of the interactive infrastructure. This framework provides APIs that give access to the scene content which needs to be processed on the graphics device side. For example, change attributes of a given material program or change transformation of a given geometry instance. A comparison between GPU-based and CPU-based rendering has also been conducted.

The different types of GPU programs and their implementation is described in this section. OptiX programs are categorized as program objects in the application model.

4.2.1 Ray Tracing Engine

OpenGL can be used for developing such an engine, however, it is not the purpose of this research. Therefore, OpenGL framework is only used for image displaying in this research project, whereas a dedicated GPU based ray tracing engine handles the rest heavy duties. Although, a few GPU ray tracer has been developed over the course of study, currently, Nvidia OptiX Ray Tracing Engine provides integrated features for ray trace based rendering development. Given the Optix framework, development of the ray trace based infrastructure has been accelerated by keeping focus on integrating the engine into the research program as oppose to developing a fully functioning ray trace based GPU engine from the beginning. One limitation that of OptiX Ray Tracing Engine featured by Nvidia is, however, supported by Nvidia graphics card only. Therefore, this research program is merely implemented with support of Optix framework and renderings are completed

on Nvidia devices. Well documented programming guide and API references from Nvidia make Optix framework a more compelling choice of the development of the ray trace based infrastructure.

4.2.2 Interactive Infrastructure

Nvidia Optix framework is based on GPU, which is highly effective for massive parallel computation. Therefore, computations such as ray intersection test and shading calculation, traditionally programmed on CPU and processed in a sequential serial approach million to billion times (depend on the configurations of rendering), can be programmed on GPU and executed simultaneously. A result rendering in real-time from this research is shown in Figure 4.4. In addition, Nvidia provides GPU programming support by its own featured platform named CUDA (Compute Unified Device Architecture). CUDA is a general purpose parallel computing platform, API (application programming interface) and programming model that leverages the parallel compute engine in Nvidia GPUs to solve many complex computational problems in a more efficient way than on a CPU [7]. It allows software developers to use a CUDA-enabled GPU for general purpose processing, an approach also known as GPGPU [28]. Therefore, more flexibility are given when it comes to any GPU related programming of the infrastructure.

4.2.3 Create GPU Program Objects

In OpitX framework, GPU program is written in the form of CUDA (Compute Unified Device Architecture) C-style language. The GPU program is then specified to the API through PTX (parallel thread execution) file, which is the virtual assembly language asso-

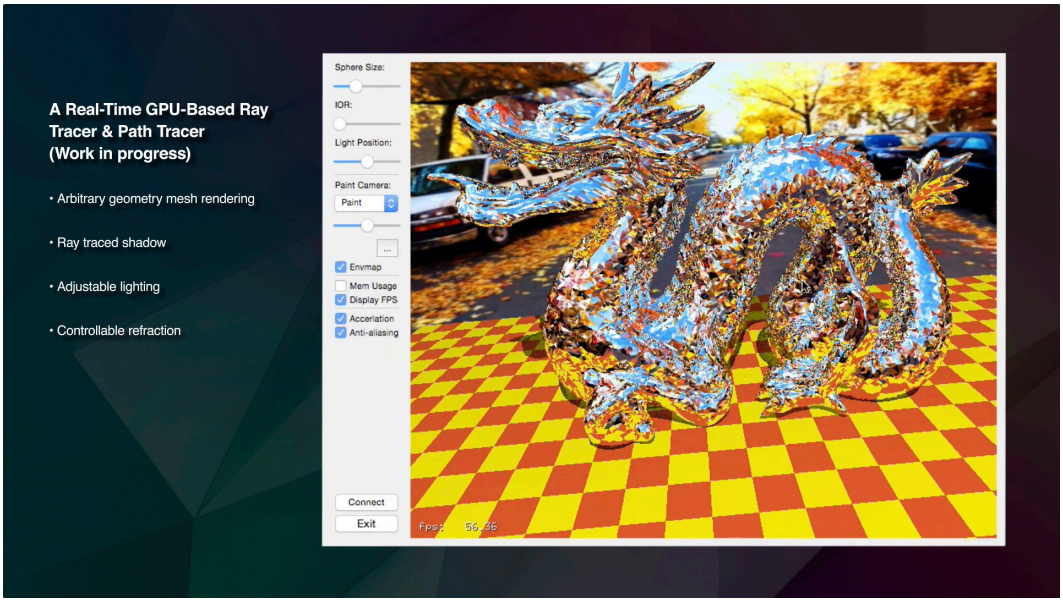


Figure 4.4: A dragon polygon with approximately 100,000 faces are rendered in real-time.

ciated with CUDA. PTX file is compilation of the OptiX GPU programs which is written in CUDA C. through the host API, PTX files are then bound to program objects.

Communications between device program objects and the host program are achieved through OptiX program variables which are declared using OptiX declare variable function. The program variables are available to both device program code through C-style language, and to the host program through the OptiX variable object API. Family of the variable get function and variable set function may be used by the host program to read and write variables declared in this way, however, a buffer should be used instead if communication is from the device program to the host.

4.2.4 Associate Ray Generation and Exception Programs

The first point of entry upon a call to context launch function is a ray generation program. The ray generation program then spawns subsequent computation performed

by the kernel, from casting rays to reading and writing from buffers. Unlike a serial C program, ray generation program is executed, in a paralleled manner, once for each thread. Each thread is assigned a unique launch index. Therefore, the value of variable is used to distinguish neighbors, in this case, for the purpose of writing result to a unique location in the output buffer with avoidance of race conditions between buffer writes.

A desired ray generation program must be specified using entry point index in order to configure a ray tracing kernel launch. The ray generation program is then associate with each entry point index created in this manner.

When errors are encountered, ray tracing kernels invoke an exception program which meant to provide communication to the host program when something has gone wrong during the launch. Exception program is also associated with entry point using set exception program function.

4.2.5 Associate Closest and Any Hit and Miss Programs

For each material, there is a closet hit program associated with it. The key of achieving interactive shader manipulation is utilizing the feature of closest hit program. Closest hit program is executed every time when its associated material is queried, which means the shading of this material will be eventually visible to the viewer. The approach in this research is therefore, a variable contains the ID (identification) of this material from the host program is passed to the material program. The ID of the material is then returned to the host program from device program through buffers at each execution of its associated closest hit program, in which the material IDs of every execution are stored in unique

locations of the buffers. Finally, material can be queried through material IDs by specifying location in the buffers and, therefore, changes can be made to the material by the host program.

Like closest hit programs, for each material, there is also an any hit program associated with it. The any hit program can increase performance when an application only needs to determine whether any intersection occurs by eliminating redundant traversal computations, in this case, the identification of the nearest intersection is irrelevant. This technique is used to implement, for instance, shadow ray casting, which is often a binary true or false computation.

Miss program is invoked when a ray intersects no primitive traced by the trace function. Variables may be accessed by miss programs the same way as closest hit and any hit programs. In the research application, the miss program implemented an environment mapping program through texture reader.

4.2.6 Associate Intersection and Bounding Box Programs

Geometry is represented with intersection and bounding box programs by implementing ray-primitive intersection and bounding box algorithms. These programs are associated with geometry objects using geometry functions of set and get, respectively, intersection and bounding box program.

Reporting Intersections

An intersection program is invoked, during ray traversal, when the current ray encounters one of primitives of a geometry object. The intersection program is responsible

for computing and reporting parametric value of the intersection as well as other details, such as surface normal vectors, through variables. Once the intersection has determined to be the closest intersection, the material attached to the primitive is queried, subsequently, the closest hit program is executed and the calculated results are passed to host through buffers such as color and shader ID.

Specifying Bounding Boxes

Spatial extent of scene primitives is bounded to acceleration structures using bounding boxes to accelerate the performance of ray traversal. The responsibility of a bounding box program is to describe the minimal three dimensional axis-aligned bounding box that contains the primitives, which is always specified in object space. They are built as tight as possible for best performance.

4.3 Render and Display Images

The final stage of a graphics pipeline is rendering and image displaying. Rendered image frames are projected onto the display which is what the viewers eventually see. The rendering and displaying is completed within the infrastructure, therefore, the application window is responsible for this task which requires the integration between the two. In this section, the concept and method of implementation for image rendering and displaying within the application are discussed respectively.

4.3.1 Camera "Captures" Image of Virtual Scene

Before the initial launch the of application, a virtual camera of the scene that is going to be rendered is initialized. The virtual camera acts like a real camera that the image "captured" from the virtual scene is what the viewer going to see as if they look at the scene through the lens of a real camera which has similar properties of this virtual camera. The virtual camera is initialized at the origin of the virtual scene with its "front" facing the negative direction of z-axis and its up-vector is pointing to the direction of y-axis for the purpose of convenience. Data of the virtual camera is passed into the program context, therefore, the ray generation program generates rays based upon the settings of the virtual camera passed through context program variables. For example, the position of the camera represents the ray generation origin and the look-at direction of the camera indicates a general direction of each shooting ray.

The calculations of the process where the virtual camera "captures" the virtual scene is completed on GPU. The heavy duty of rendering of the scene is then handled in device program with massive parallel computation enabled from GPU. The result color of each rendered position on the final image is stored in buffers and passed back to host program for displaying.

4.3.2 Display Images with OpenGL

OpenGL, also known as Open Graphics Library, is an application program interface (API) to graphics hardware such as GPU [12]. There is a set of several hundreds functions consisted in the interface that allow programs to specify operations involved in producing

high-quality graphical color images [29]. The image buffer data carried out by OptiX API has various different types that is define by users, while OpenGL provides the most flexibility with its functions and procedures set.

The Data of rendered image is carried out in OptiX buffer object from device program. OpenGL buffer object is then used to map out the OptiX buffer and retrieve the color of each pixel from the image data. Finally, the rendered image is pushed through the graphics pipeline of OpenGL and displayed onto the screen to the viewers.

4.3.3 Display Image Frames in Qt Widget

Qt is a C++ based framework which enables interactive and cross-platform application development [27]. It is mainly used for the development of applications with GUI; though, programs do not have a GUI can be developed with Qt framework as well. The interface of the research application is developed based upon Qt frameworks for the purpose of creating an interactive GUI thereof, which means rendered image frames are displayed in the application window made with Qt carried by a Qt widget, the GUI and Qt is further discussed in section 5.4. Qt widget has built-in OpenGL functionalities, therefore, the display of image is migrated to Qt Widget OpenGL from conventional OpenGL frameworks with proper adjustments and, eventually, image frames are displayed in the graphical window the application.

4.4 GUI

Qt Creator®provides well established UI framework [8], and Qt Designer gives a GUI-approach of generating UI for applications. The Qt framework is utilized by a large number of production tools in the animation industry, including Autodesk Maya®. Therefore, Qt Creator is used as the IDE (Integrated Development Environment) for the development of the interactive ray trace based infrastructure. In addition, Qt Designer is a subsystem of Qt Creator, which is a powerful cross-platform GUI layout and forms builder. Therefore, Qt Designer is naturally used as means of creating GUI for the target application.

4.4.1 Create GUI

As briefly discussed in section 4.3.3, Qt provides a framework such that interactive application can be implemented through a set of very convenient methods. The interaction of a Qt application is described as three major different approaches: one is through mouse event handle functions, the other one is to use signals and slots of Qt widgets, and the combination of the two can provide even more flexibility in certain cases. In an OpenGL window, mouse event handle functions can monitor mouse state and track mouse motions, whereas, signals and slots methods are used for interactions between Qt widgets such as push buttons and drag sliders, in which specific actions taken emit signals and the corresponding slots are responsible for receiving the signals and execute preset operations upon receipt of the signal. In the research program, all of these approaches are implemented.

The back bone of the image display is still OpenGL. Therefore, the most intuitive

method of interacting with the scene, such as toggle camera, is to dictate directly within the virtual scene. While other interactions are implemented with Qt widgets through signal and slots, or the combination of both depending on various scenarios. Methods and concepts of each implementation are described in the following sections

4.4.2 Direct Interaction with Scene

In the virtual scene, as an example, move camera position is achieved through middle mouse drag; change the orientation of the camera is achieved through left mouse drag, and zoom in/out is achieved through right mouse drag, in which all mouse drags are completed directly on the displayed images in the OpenGL window of the application. Changes made in the application display window are passed to OpenGL program object as user inputs, the associated OpenGL functions is then executed to update program object variables, in this case camera properties, and notify the ray tracing engine. The engine updates its context object upon receipt of change notifications. Therefore, new rays with new origins and directions are generated, subsequently, new image frames of the virtual scene are rendered. New image data is then passed to OpenGL program for displaying through buffers.

4.4.3 Signals and Slots

Some interactive features which cannot be implemented with intuitive direct scene interaction by user inputs are achieved through signals and slots of objects provided with Qt framework. Where specific actions taken emits signals and the corresponding slots is responsible for receiving the signals and execute preset operations upon receipt of the signal.

For example, load in an environment map for the OptiX miss program, in which ray intersects no primitive so the ray tracing program renders a texture from the environment map instead of a solid background color, such as sky. The procedures, in this case, may not be implemented with mouse drag for scene interaction. Whereas, a check box on the GUI is created, when checked, a signal of the box has checked is emitted and a slot of the application receives the signal. Thereof, the application executes another method of widget object which reads a preset image file of the environment map into the application as texture. The data of the texture is then offloaded to the program, rendered, buffered, and displayed as the environment image of the virtual scene through various procedures.

4.4.4 Interactive Shader

In this research, the infrastructure provides access to shader which gives artists ability to manipulate shading effects. For example, change diffuse color, change index of refraction, change reflectance ratio, etc.

At first, shader selection is enabled by mouse-clicking directly on the geometry instance which has the associated material program within the displayed scene image. This way, users can intuitively select a specific shader. A signal of the mouse click is emitted, although, the only data from this emission may be passed to the program is the position of the mouse when the click event occurs within the OpenGL coordination, which is monitored by the mouse event functionalities.

In order to retrieve the correct shader that its associated geometry instance is clicked by users, secondly, a combination approaches of interacting with GUI are implemented. At

the initialization stage of the material program, a shader ID which represents the material program is passed to itself through a device program variable. After the launch of the program, each execution of the closest hit program within the material program writes the shader ID to a device buffer at a unique location based upon the launch index. Which means the pixel color that is eventually going to be visible to the viewers is stored in a unique location of a buffer, upon each execution of the closest hit. At the same time, the shader ID of the material program which is used to calculate the shading of the geometry and resulted the pixel color is stored in the same unique location of another buffer. Therefore, query on the same locations in each buffer receives a pair of a pixel color and a shader ID, while the pair represents data in the same location of the two buffers. In turn, the position of each mouse click within the virtual scene may be used to query the shader ID of the pixel clicked by users. In other words, this shader ID indicates the material program which is associated with the geometry selected by users.

Manipulate Shader through User Input

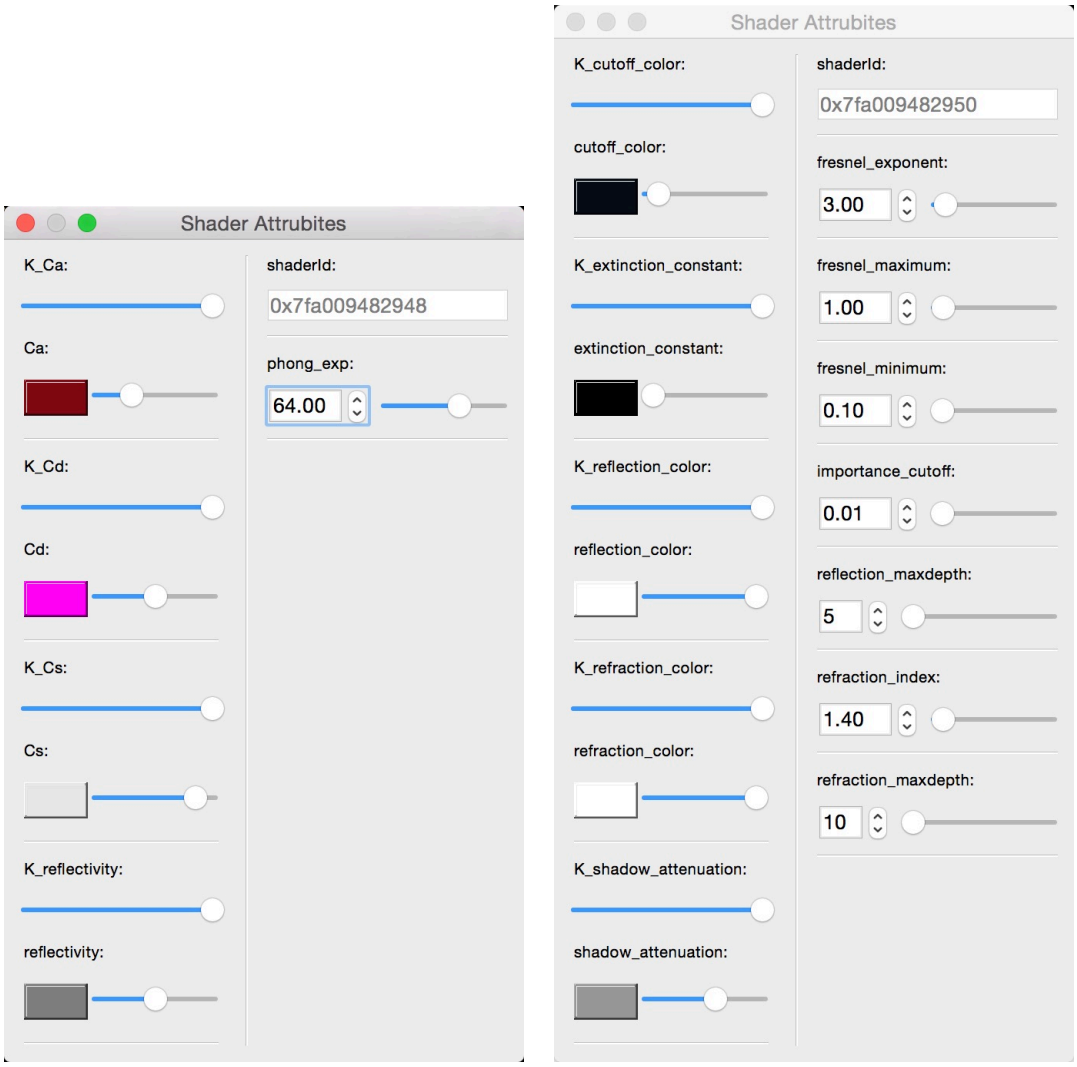
Given the shader ID, interact with a targeted material program may be achieved. Although, another GUI is created with the implementation of signals and slots – shader attribute GUI. A shader contains various attributes stored in the form of material program variables. Each attribute dictates the shading behavior of the property of a certain material texture, such as specular color of a Phong shader. Meanwhile, two different shaders may have different types of attributes. For example, a shader of glass material contains index of refraction and other attributes to describe the shading behavior as oppose to an opaque

material where no such attributes are needed. Therefore, upon the receipt of a mouse click event within the OpenGL coordination, a condition is determined whether the click occurs on a geometry object. Thereof, if the condition is true, another condition is determined whether the geometry instance object has material associated with it. If true again, a window of shader attributes is created, in which controls and displays of each different type of attributes from the selected shader are populated (Figure 4.5). Connections between signals and slots of each control of attribute are established.

As an example, move the slider position of the diffuse color emits a signal which is received by the slot of push button of diffuse color. The push button changes the brightness of its color, concurrently, the brightness of the diffuse color of the shader also changes. Updated shading effects is then rendered and displayed in the application window which reflects the change made through the slider move. Click direct on the attribute push button emits another signal, upon receipt, a dialog window of color selection is created which allows user select specific color value for the clicked attribute (Figure 4.6). Selection of color also emits signal to the corresponding slot and recovers which executes a serial of procedures, in the end, renders image with updated shading effects and updated shader attribute GUI (Figure 4.7).

4.5 Applications Data Intercommunication

Autodesk Maya 2015®[5] is used, in this research, as the 3D content creation software. Maya is an industry standard 3D content creation package which is widely used in



(a) (b)

Figure 4.5: Attributes of different shaders are populated in the shader attribute GUI; different types of attribute are also categorized separately by barycentric and classic.

animation, film VFX and game production studios. Maya provides flexible programmability through its integrated UI commands and scripting languages: MEL (Maya Embedded Language) and Python [10]. Therefore, the establishment of data communication between Maya and the ray tracing application is achieved through a message publish and subscribe

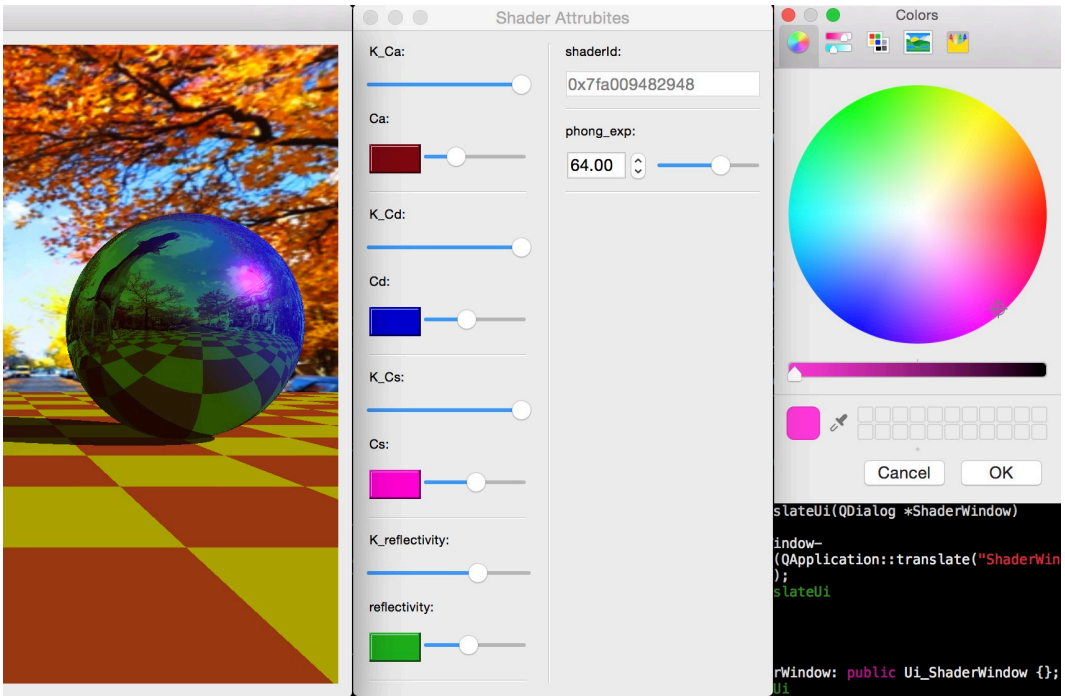


Figure 4.6: Example of color selection dialog.

system provided by PubNub®Python SDK (software development kit). PubNub is a global Data Stream Network that utilizes a publish/subscribe model for real-time data streaming and device signaling (Figure 4.8). Due to its internet based feature, more advantages and flexibility in developing the scalability of the interactive infrastructure may be accomplished. For example, 3D content is created in Maya running on a laptop, meanwhile, the ray tracer is running on a dedicated workstation for an optimized interactive ray tracing performance. Both machines have internet access. During the creation process, changes of the 3D content are published to PubNub network which are then subscribed by the ray tracer. Final results are displayed on the screen of the dedicated workstation interactively. The concept and method for integrating PubNub SDK to both Maya and the ray tracing application is described in the following sections:

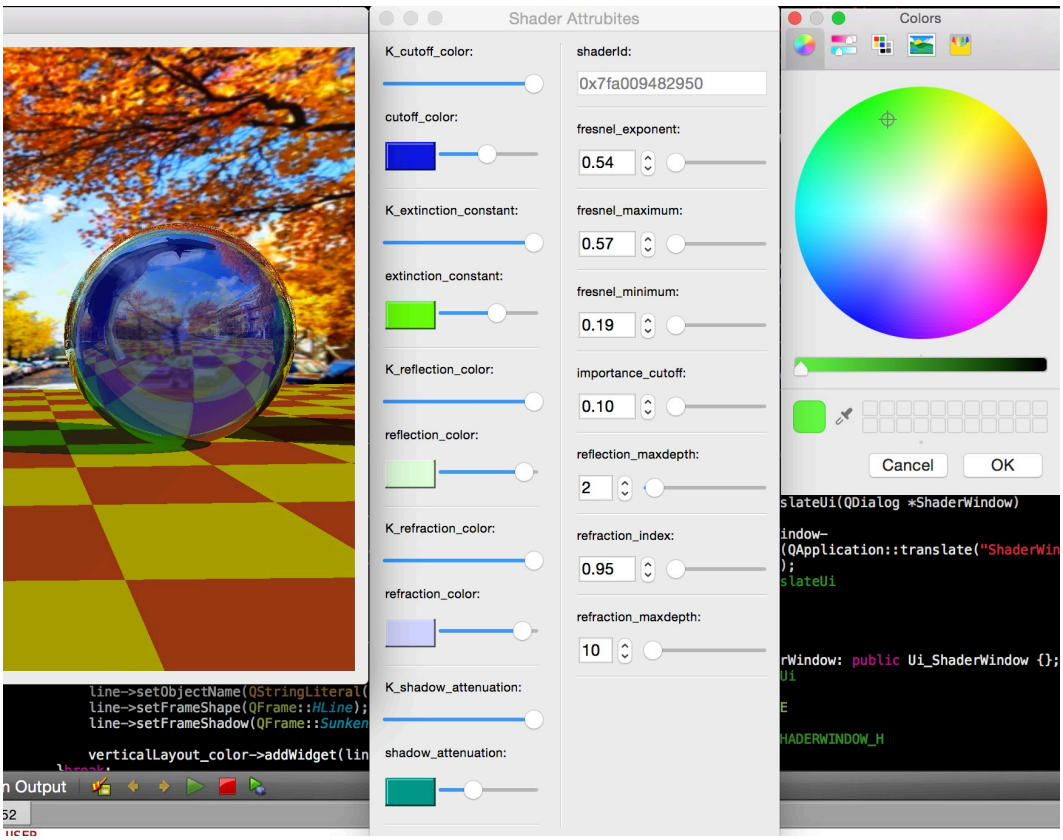


Figure 4.7: Example of color selection dialog (updated).

4.5.1 Maya Data Publish

The primary utilization of Maya, in this project, is to create digital content. Changes that are made to the 3D content are published by Maya to the Data Stream Network featured by PubNub so the receiver, ray tracer in this case, can reflect the changes.

A Maya UI command has first implemented in itself using the integrated Python scripting language. The implementation of this UI command initializes a signaling mechanism which consistently monitoring the operation of the Maya session. Specific changes dictated by user inputs triggers the signaling mechanism, such as transform geometry. The message broadcast system, which is part of PubNub SDK, is then naturally integrated into

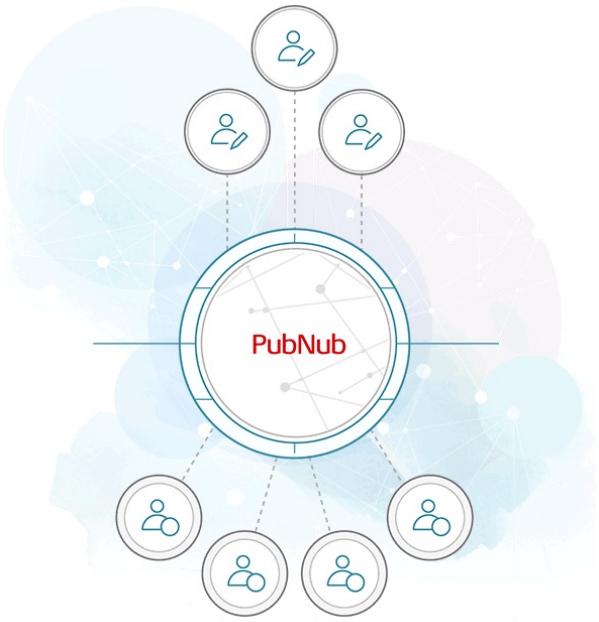


Figure 4.8: Data Stream Network by PubNub, reprinted from [26]

the signaling mechanism as it is written in python.

Each valid signaling executes a message broadcast which contains data of changes, such as transformation matrix, in a form of Python data structure are published to the Data Stream Network. Meanwhile, a publish key is assigned to the message which is used by the message receiver to subscribe designated publish event.

4.5.2 Ray-Tracer Data Subscribe

First, the application is programmed in C++ on the host side, whereas, PubNub SDK is written Python. Therefore, rather than natural integration, the PubNub Python SDK is implemented in a C++ Python embedding method. As part of the SDK, message subscription is then integrated to the application which is executed in a form of C++ object. A subscription key is provided as message ID such that the application receives the targeted

message from the Data Stream Network which is published by designated Maya session.

The message subscription, however, is in a form of looping mechanism which means the application needs to consistently execute the instruction rather than a single serialized instruction to receive the interactive data from the Data Stream Network. Therefore, an additional CPU thread is programmed in the application which provides the ability of running the interactive message receiving program parallel with the main application program.

Subscribed data is in the form of Python data structure from Maya in the first place. The Python data, mostly Python list and tuple, is then reassembled and converted to C++ float array which is eventually turned into matrix using C++ data structure. The matrix data is then passed to the main interactive host program running on anther CPU thread, upon receipt of change notification. The attributes of corresponding object are updated and passed to the rendering engine, in the end, new image frames rendered and displayed onto the application window, which reflect the message received, which also reflect the changes are made in Maya interactively (Figure 4.9).

4.6 Integration and Measurement

4.6.1 Painting Camera and Cubist Rendering

Altering the property of the virtual camera “lens” where camera and its lens are merely a concept rather than real objects. They are invisible to the users even in the displayed virtual scene. Therefore, again, intuitive direct scene interaction through the change of mouse state and position may not be implemented with such purposes.

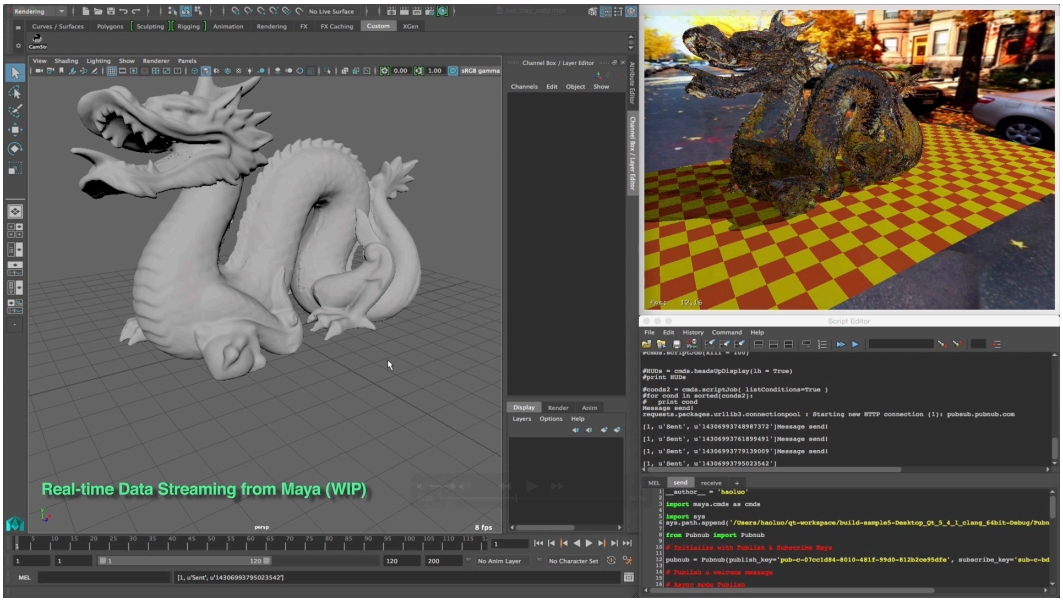


Figure 4.9: Example shows Maya camera toggling and geometry transforming data is streamed to the receiver (currently the renderer). The rendered image frames have reflected changes that are made in Maya.

As an example, ray generation program can be treated as if the camera is associated with a shader program. The special shader alters the direction, origin, or both direction and origin of each ray in the process of ray generation based upon user inputs, in this case, a texture map. A tool button on the GUI is created, when clicked, a signal of the button has clicked is emitted and a slot of the application receives the signal. Thereof, the application executes another method of widget object which pops up a window of the file browser upon receipt of the signal. The file browser window allows user point to the directory contains the image file of the camera lens filer map and read it into the application as texture. The color data of each pixel from this texture is translated into vector. This vector map is then offloaded to ray generation program, in the end, resulted in Painting Camera (Figure 4.10) and Cubist Rendering (Figure 4.11) through different ray altering algorithms. The virtual

scene is rendered as a refracted world through camera, in which the refracted lens is created for the rendering by implementing Painting Camera Technique [17]. An image is read in as the refracted lens image, resulted in such an illusory and unreal world like in someone's dream. Similar to the Painting Camera, cubist rendering style is created by loading in a cubist image that has the look of targeted style. Cubist camera is created for the rendering by implementing Cubist Rendering technique [31].

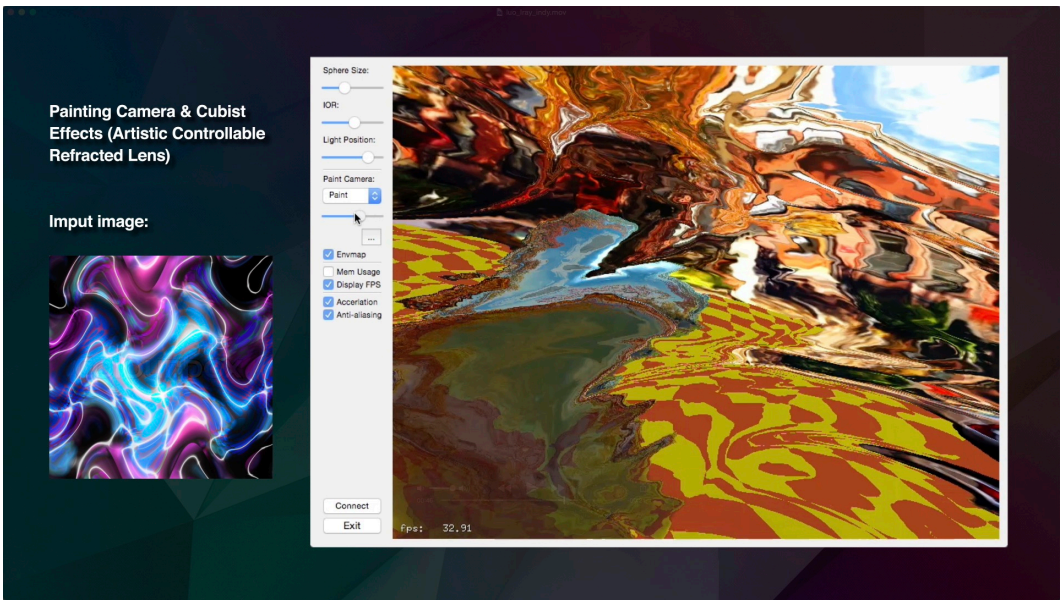


Figure 4.10: Painting camera frame rendered from infrastructure.

4.6.2 Interactive 3D Painting

Using this infrastructure, Chethna Kabeerdoss has successfully created a 3D dynamic scene with a Barycentric shader that emulates a 2D art painting [15]. She integrated this scene into my ray tracing infrastructure to reconstruct and dynamically render the 3D scene which closely represent the 2D art painting shown in Figure 4.13.

That process is illustrated as in Figure 4.12. A 3D scene is created as in the given

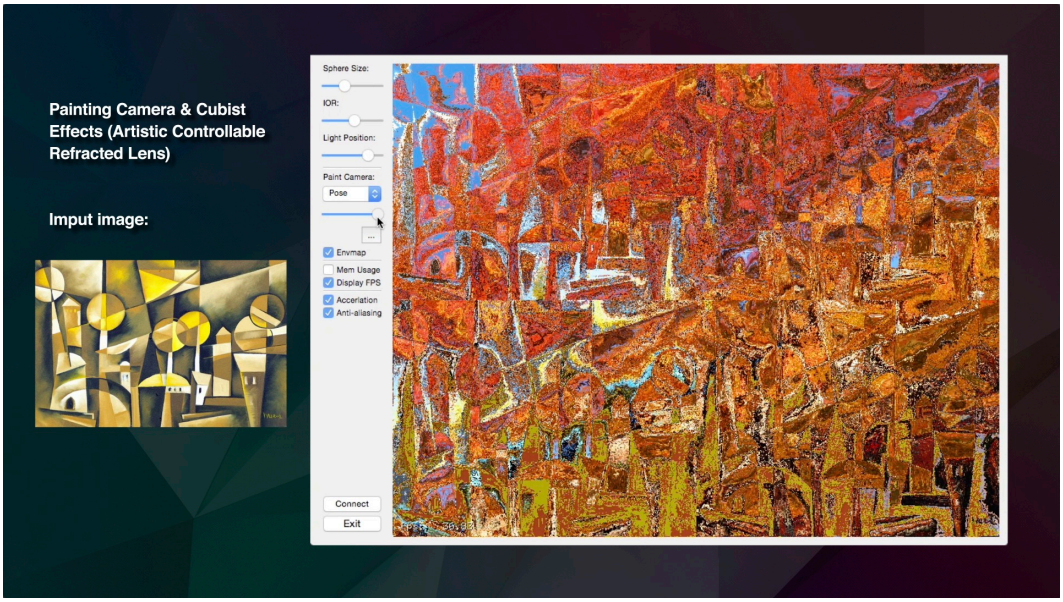


Figure 4.11: Cubist Rendering example in real-time.

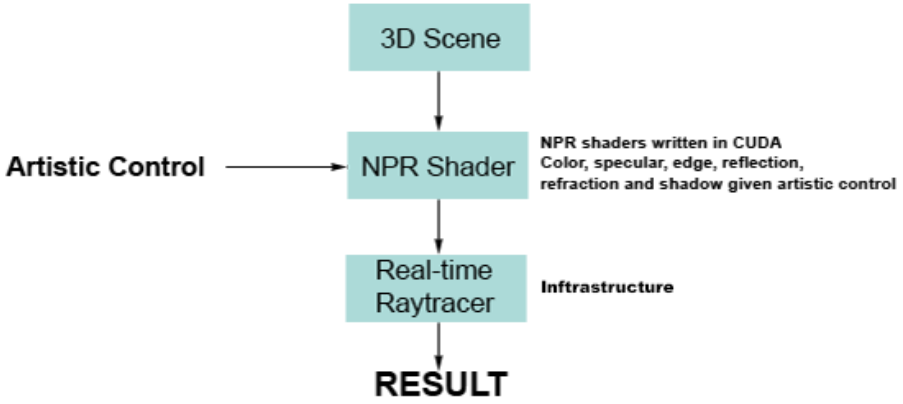


Figure 4.12: A flowchart showing the process for creating the interactive 3D painting artwork. CUDA is used for developing the Barycentric shader for and applied to 3D object. The user has real-time artistic controls to adjust the effect parameters through GUI. In the research Autodesk Maya is used again for modeling, CUDA for developing shader and the real-time ray tracer is used for rendering. Texture maps which were used in the Barycentric shader are painted and edited in other third party software.

Figure 4.13: ‘Decido (decisions)’ - Reference digital painting by Rachel Cunningham, reprinted from [9]

A generic Barycentric shader for non-photorealistic real-time rendering is developed. Real-time reflections, refractions and shadows is handled by this shader using my ray tracing infrastructure. This shader is largely based on the principle of the Barycentric Shaders developed by Akleman et al [2]

To match the reference painting, Kabeerdoss modeled the characters in the 3D scene and designed the layout of the scene to ensure similarity between the position of the characters and the reference painting. Finally in Autodesk Maya, She created the virtual camera to match point of view in the reference. Figure 4.14 shows an image capture of the 3D

scene.

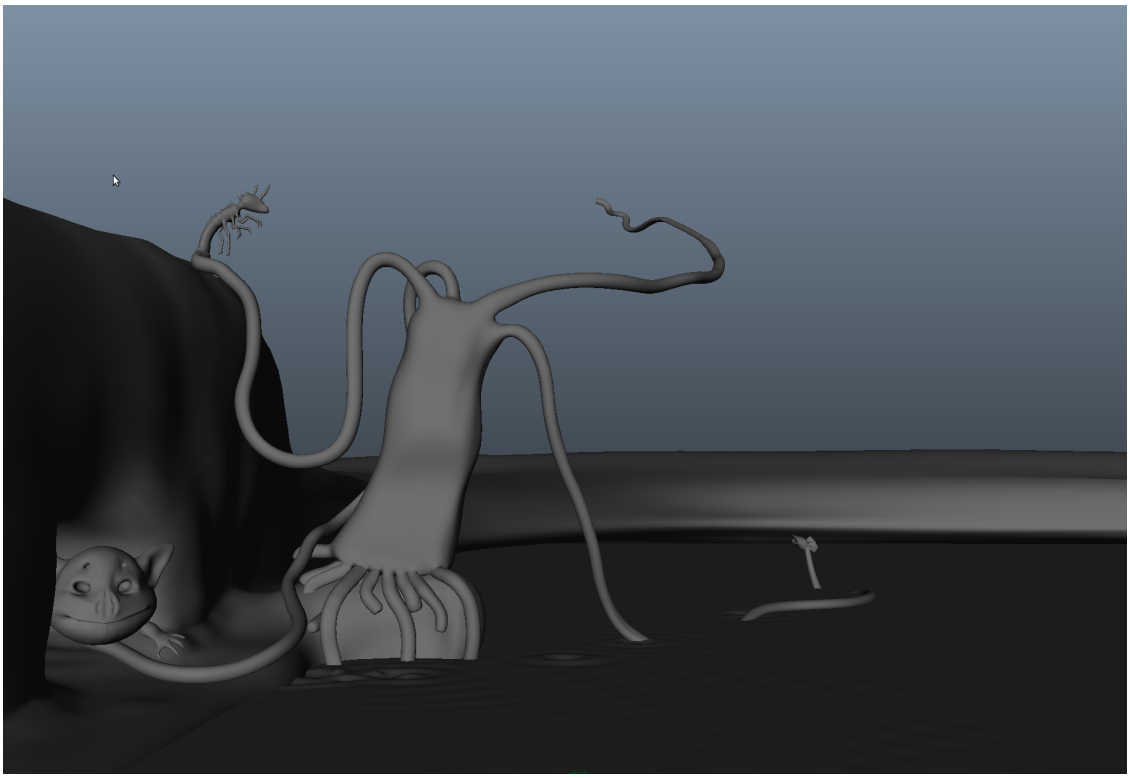


Figure 4.14: 3D scene modeled in Autodesk Maya, reprinted from [15]

Rendering Scene Using Mental Ray

Kabeerdoss first rendered the geometry using Mental Ray renderer out of Autodesk Maya to evaluate the volume of the reconstructed scene and the visual aesthetics. In order to maintain the same look and feel of the visual reference, they also hand-painted the textures. Scene is carefully illuminated with virtual lighting imitating the original lighting in the referenced artwork. Shaders in Mental Ray did not offer enough control for artistic input though. Beauty (contains diffuse, specular, reflection, refraction and shadow color information), and normal passes are rendered as output, therefore, more artistic control of

the final image in post production is maintained.

Figure 4.15: Composited image of 3D scene by Chethna Kabeerdoss, reprinted from [15]

Digital Compositing/Post Processing

The output passes are transferred into The Foundry's Nuke for post process. Built-in nodes in Nuke which execute the basic 2D image manipulations are used to synthesize the final image. Kabeerdoss used the blur, grade, and edge detect nodes to assemble the passes for each of the objects in the scene. These are then combined using the merge node. Finally, synthesized Figure 4.15 which shows the resulted image. The scene was rendered with a moving virtual camera. Several passes were needed in order to have artistic control during post production. Each pass has a render time that ranges from a few seconds up to 2 minutes, in Autodesk Maya. The final sequence is rendered out of Nuke taking a minimum of 2 minutes per frame.

4.6.3 Create Generic Barycentric Shader for NPR Real-time Rendering

Another part in research of Kabeerdoss is to develop Barycentric shader for real-time rendering with GI which is also art-directable [23]. The development is aimed at replicating the results, that were achieved in post production, in real-time. She made us of CUDA to build the generic art-directable Barycentric shader for NPR and rendered images in my real-time ray tracing infrastructure.

Based on the visual analysis in the research, a procedural shader that computes diffuse reflection, specular highlight, silhouette, transparency and shadow is developed. This customised barycentric shader is then used for the objects in the 3D scene, by enabling the appropriate effect parameter, the parameters provided to the user for art-directable control. Kabeerdoss achieved this final result of the interactive 3D scene as shown in Figure 4.16. The scene renders at 5 frames per second.

4.6.4 Comparison Between Rendering Means

The renderings from resulted infrastructure has also been measured against CPU-based rendering. Targeted rendering speed has been achieved at interactive level, meaning around 24 fps, as oppose to CPU-based rendering which usually takes minutes if not hours to render only 1 frame at same quality. In Kabeerdoss's case, her reconstructed 3D scene with the integrated shaders has a rendering rate at 5 frames per second on the external GPU Nvidia Quadro K5200, whereas, the same 3D scene consumes 20 second to 2 minutes to render only one frame in Maya Mental Ray, as shown in Table 4.1. Therefore, rendering is approximately 100 to 600 times more efficient on the GPU-based infrastructure than on

Figure 4.16: Frame rendered by the ray-tracing infrastructure with art directed shader integrated.

the commercial CPU-based renderer.

CPU-based vs GPU-based Rendering		
Property	Maya Built-in Renderer	GPU-based Renderer
Render time per frame	20s - 2 mins	0.2s
Real time interaction	No	Yes
Artistic Control	No	Yes
Color space control	No	Yes

Table 4.1: Comparison between CPU-based commercial renderer and GPU-based ray tracing infrastructure.

5 CONCLUSION AND FUTURE WORK

The goal of this research project of merging two major technologies is achieved: produce both photo-realistic and NPR imagery with high visual realism through ray trace based rendering, meanwhile, possess the capability of rendering complex scene in real-time using parallel computing techniques. Based upon interactive ray tracing, this research infrastructure is developed successful in a way such that artist can intuitively interact with the virtual environment through GUI of the application, such as iterating shading and lighting effects. A creative approach of establishing data communication between the main application program and other 3D software has also presented in this research. Users may interact with the infrastructure through third party software Maya by the Data Stream Network supported by PubNub. In addition, such interactions are not limited to be on single computer, whereas, users can interact with the main ray tracing application from one machine to another. The infrastructure is object-oriented programmed in a way such that it remains high level of readability, usability and scalability.

The project is tested on two GPUs of different categories from Nvidia: GTX 970 and Quadro K5200. An approach of connecting workstation-size GPU to a computer through Thunderbolt port and/or ExpressCard slot dedicated to the program has presented. This external GPU approach provides more flexibility, adaptability, compatibility, and portability for the research project.

This project is benefit for artists whose work is directly related to shading, lighting,

and rendering in a variety of production pipelines once integrated. In CG production, huge amount of time would be saved on rendering. For artists whose work is shading and lighting related, faster rendering means they have earlier access to the final results of shading and lighting, in turn, results in quicker iteration. Therefore, quality and/or creativity of the art work may be improved.

In conclusion, the development of the interactive ray tracing infrastructure opens a door for exploring the implementation of such technology into the graphics pipeline such that it may be beneficial to the process of CG production whether it is photo-realistic or NPR.

5.1 Integrate Virtual Reality Devices

The ability of rendering images in real-time makes it possible for integrating the render application to Virtual Reality (VR) Devices. In VR application program, rendered images are essentially projected onto the VR display and distorted in a way that it imitates how human eyes perceive images in reality when viewer look at those images right in front of eyes. VR display is basically a screen or screens, similar to PC monitor, specifically made to fit the view range of human eye's vision and wearable like eyeglasses. Therefore, with proper algorithms of image distortion, frames rendered from the research program can be projected onto the VR display. The transform data from VR device can be send back to rendering program to update camera, thereof, renders updated images and displays in VR headset. A new way of interacting with ray traced based virtual environment can be

accomplished with such graphics pipeline.

5.2 Simpler and Faster GPU Setup

With the introduction of Thunderbolt 3, a simpler and faster external GPU setup can be implemented. Intel has finally granted permission for companies who produce enclosures that provides support for the Thunderbolt 3 connection standard after its debut. Thunderbolt 3 has higher data transfer speed and it also compatible with USB (universal serial bus) Type-C. Therefore, GPU can be housed in a Thunderbolt 3-enabled enclosure and directly connect to any Thunderbolt 3 compatible computer through Thunderbolt 3 or USB Type-C port, meanwhile, beneficial from the 40Gbps data transfer rate compared to a 500Mbps through ExpressCard. Therefore, higher display rate can be achieved.

REFERENCES

[1] T. Akenine-Müller, E. Haines, and N. Hoffman, Real-Time Rendering. Natick, Massachusetts, USA: A K Peters, Ltd., 2008.

[2] E. Akleman, D. H. House, and S. Liu, "Barycentric shaders: Art directed shading using control images," Proceedings of Expressive'2016: Computational Aestetics Conference, pp. 39–49, 2016.

[3] A. Appel, "Some techniques for shading machine renderings of solids," in Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference. ACM, 1968, pp. 37–45.

[4] Apple, "About system integrity protection on your mac," Website: https://support.apple.com/en-us/HT204899, Apple Inc., 2016. [Online]. Available: https://support.apple.com/en-us/HT204899

[5] Autodesk, "Autodesk maya," Website: https://www.autodesk.com/products/maya/overview, Autodesk Inc., 1990. [Online]. Available: https://www.autodesk.com/products/maya/overview

[6] T. D. Chalmers and E. Reinhard, "Practical parallel rendering," ISBN 1-56881-179-9. AK Peters, Ltd., 2002.

[7] N. Corporation, “Multi-process service,” Website: https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf, NVIDIA Cooperation Inc., 2019. [Online]. Available: https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf

[8] Q. Creator, “Qt creator whitepaper,” Website: https://wiki.qt.io/QtCreatorWhitepaper, Qt Project, 2007. [Online]. Available: https://wiki.qt.io/QtCreatorWhitepaper

[9] R. Cunningham, “Decido (decisions),” Digital Painting, Website: http://www.rachelcunninghamwang.com/digital-painting.html, 2012. [Online]. Available: http://www.rachelcunninghamwang.com/digital-painting.html

[10] P. S. Foundation, “About python,” Website: https://www.python.org/about/, Python Software Foundation, 1990. [Online]. Available: https://www.python.org/about/

[11] A. Glassner, An Introduction to Ray Tracing. Cambridge, Massachusetts, USA: Academic Press, 1989.

[12] S. Graphics, “Sgi - opengl overview,” Website: https://web.archive.org/web/20041031094901/http://www.sgi.com/products/software/opengl/overview.html, Silicon Graphics, 1992. [Online]. Available: https://web.archive.org/web/20041031094901/http://www.sgi.com/products/software/opengl/overview.html

[13] N. O. Group, "Optix programing guide," Website: https://raytracing-docs.nvidia.com/optix6/guide_6_5/index.html, NVIDIA Cooperation Inc., 2017. [Online]. Available: https://raytracing-docs.nvidia.com/optix6/guide_6_5/index.html

[14] HowStuffWorks, "How stuff works pcie," Website: https://computer.howstuffworks.com/pci-express.htm, 2009. [Online]. Available: https://computer.howstuffworks.com/pci-express.htm

[15] C. Kabeerdoss, "Art directed shader for real time rendering - interactive 3d painting," Master's thesis, Texas A&M University, College Station, TX, 2016. [Online]. Available: http://oaktrust.library.tamu.edu/handle/1969.1/165773

[16] S. Liu and E. Akleman, "Chinese ink and brush painting with reflections," in SIGGRAPH 2015: Studio, ser. SIGGRAPH '15. New York, NY, USA: ACM, 2015, pp. 8:1–8:1. [Online]. Available: http://doi.acm.org/10.1145/2785585.2792525

[17] S. Meadows and E. Akleman, "Abstract digital paintings created with painting camera technique," Proc. D'ART 2000/Information Visualization 2000, 2000.

[18] J.-C. Nebel, "A new parallel algorithm provided by a computation time model," in Proceedings of Eurographics Workshop on Parallel Graphics and Visualisation'1998. Eurographics, 1998, pp. 65–72.

[19] Z. Noh and M. S. Sunar, "A review of shadow techniques in augmented reality," in in Machine Vision. Los Alamitos, CA, USA: International Conference on, 2009, p. 320–324.

[20] NVIDIA, “What is gpu-accelerated computing,” Website: http://www.nvidia.com/object/what-is-gpu-computing.html, NVIDIA Cooperation Inc., 2007. [Online]. Available: http://www.nvidia.com/object/what-is-gpu-computing.html

[21] N. Optix, “Nvidia® optix™ ray tracing engine,” Website: https://developer.nvidia.com/optix, 2016. [Online]. Available: https://developer.nvidia.com/optix

[22] Pan, “External gpu on mac,” Website: http://www.journaldulapin.com/2013/08/24/a-thunderbolt-gpu-on-a-mac-how-to, Pierre Dandumont, 2013. [Online]. Available: http://www.journaldulapin.com/2013/08/24/a-thunderbolt-gpu-on-a-mac-how-to

[23] P. B. P.Dutré and K. Bala, Advanced Global Illumination. Natick, Massachusetts, USA: AK Peters, 2006.

[24] F. Pfenning, “NPR,” Website: http://www.cs.cmu.edu/~fp/courses/graphics/pdf-2up/21-npr.pdf, 2003, pp. 3–, accessed 17 April 2003. [Online]. Available: http://www.cs.cmu.edu/~fp/courses/graphics/pdf-2up/21-npr.pdf

[25] H. Pixel, “Howling pixel adat lightpipe expresscard,” Website: https://howlingpixel.com/i-en/ADAT_Lightpipe, 2017. [Online]. Available: https://howlingpixel.com/i-en/ADAT_Lightpipe

[26] PubNub, “Pivotal web services documentation,” Website: http://docs.run.pivotal.io/marketplace/services/pubnub.html, PubNub., 2016. [Online]. Available: http://docs.run.pivotal.io/marketplace/services/pubnub.html

[27] Qt, "Qt: About us," Website: https://web.archive.org/web/20170222172844/https://www.qt.io/about-us/, Qt Project, 1995. [Online]. Available: https://web.archive.org/web/20170222172844/https://www.qt.io/about-us/

[28] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. mei W. Hwu, "Optimization principles and application performance evaluation of a multithreaded gpu using cuda," in Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, ser. PPoPP '08. New York, NY, USA: ACM, 2008, pp. 73–82. [Online]. Available: https://dl.acm.org/citation.cfm?id=1345220

[29] M. Segal, K. Akeley, C. Frazier, J. Leech, and P. Brown, "The opengl graphics® system: A specification," Website: https://www.khronos.org/registry/OpenGL/specs/gl/glspec40.core.pdf, Khronos Group, 2006. [Online]. Available: https://www.khronos.org/registry/OpenGL/specs/gl/glspec40.core.pdf

[30] P. Shirley and S. Marschner, Fundamentals of Computer Graphics. Natick, Massachusetts, USA: A K Peters, Ltd., 2009.

[31] J. Smith, E. Akleman, R. Davison, J. Keyser et al., "Multicam: A system for interactive rendering of abstract digital images," in Bridges: Mathematical Connections in Art, Music, and Science. Bridges Conference, 2004, pp. 265–272.

[32] K. Suffern, Ray Tracing from the Ground Up. Natick, Massachusetts, USA: A K Peters, Ltd., 2007.

[33] T. Whitted, “An improved illumination model for shaded display,” in ACM Siggraph 2005 Courses. ACM, 2005, p. 4.

[34] L. Wilson, “Cinema4d - photorealistic render challenge,” Website: www.behance.net/gallery/2819697/Photorealistic-Render-Challenge, Cinema4D, 2016. [Online]. Available: www.behance.net/gallery/2819697/Photorealistic-Render-Challenge

APPENDIX A

CONNECT EXTERNAL GPU

A.1 Terminal Commands

sudo nvram boot-args=”kext-dev-mode=1”

A.2 Modify the Distribution File in Installation Package

Download the drivers (here the latest for Yosemite 10.10.1).

Then unzip the drivers.

pkgutil –expand WebDriver-343.01.02b02.pkg WebDriver

Modify the Distribution file with a text editor (such as TextWrangler).

Replace the line var found_hardware = 0; by var found_hardware = 1;

Zip the drivers.

pkgutil –flatten WebDriver WebDriver-334.01.02b02_mod.pkg

Install the drivers and reboot.

A.3 Modify Kernel Extension Files

The first two are related to the display, the third one is dedicated to the support of the sound through HDMI or DisplayPort.

`/System/Library/Extensions/NVDAStartup.kext`

`/System/Library/Extensions/IONDRVSupport.kext`

`/System/Library/Extensions/AppleHDA.kext/Contents/PlugIns/AppleHDAController.kext`

You have to get each file, open the package (secondary click -> Show Package Contents) and open the file Info.plist with a text editor that supports editing system file.

In this file, look for sections beginning with $< key > CFBundleIdentifier < /key >$ and add just before $< /dict >$ the two following lines :

$< key > IOPCITunnelCompatible < /key >$

$< true/ >$

Beware, there are several occurrences in IONDRVSupport.kext

Note: the operation has to be repeated at each minor system update (e.g the move to 10.10.5).

Warning: if the computer and GPU cable is disconnected while on, the kernel panic is to be foreseen.