Oregon State University
College of Engineering

CS Capstone Technology Review

November 21, 2017

3D Virtual Reality Painting

Prepared for

EECS

Dr. Kirsten Winters _____ _____
Signature *Date*

Dr. Mike Bailey _____ _____
Signature *Date*

Prepared by

Group 66

PolyVox

Richard Cunard _____ _____
Signature *Date*

CONTENTS

REFERENCES

[1] "Cryengine," http://docs.cryengine.com/display/CEMANUAL/VR+Support, accessed: 2017-11-21.

[2] "Unity," https://docs.unity3d.com/Manual/index.html, accessed: 2017-11-21.

[3] "Unreal," https://docs.unrealengine.com/latest/INT/l, accessed: 2017-11-21.

1 SCRIPTING LANGUAGE

1.1 Overview

The choice of what programming language to use for developing our project will heavily inform the design and functionality of our final product. The goal of the project from a development perspective, is to create an efficient, stable, and flexible program. Any game engine available to the team will have a set of natively compatible scripting languages, which further narrows the choices available to the team.

1.2 Criteria

The primary factors each should be judged on are speed of code execution, robustness of features, and ease of development. Depending on the language, the output of the code can have wildly differing execution times. Given the requirement of maintaining a high frame rate, this is a significant factor in deciding the language. The feature set of each language will significantly affect how the program is constructed, and can help or hinder various aspects of development. Finally, assuming the first two criteria are met, the decision should be based on the how challenging development is with a given language.

1.3 Potential Choices

With that in mind, the most viable options for use in the project ar C++, C#, and JavaScript. Each of these are robust, heavily supported languages, with large selections of publicly available libraries. More importantly, each is used natively in at least one major game engine. Choosing which of the three languages will likely also inform what game engine is chosen for the project, given that certain languages are either poorly supported, or simply incompatible with specific engines.

1.3.1 C#

C#, similar to its predecessor C++, is a powerful object oriented language with extensive functionality and outside support. C# is supported by most modern game engines, including Unreal, Unity and CryEngine, which would present the team with more flexibility for future development choices. As mentioned above, while C# is interpreted, it has several optimization features. These include the ability to query machine specifications, perform runtime code optimizations that would not be possible with a compiled language, and perform heap more efficiently. While the team has very limited experience with C#, this would not seriously affect development, as it is similar in syntax and design to C++, which the entire team has extensive experience working with. C# would be a viable choice for any choice in game engine and design methodology.[1][2]

1.3.2 C++

C++ is a powerful language with extensive outside library support, and a fundamental compatibility with the Unreal Engine, given that it is what Unreal is primarily built with. However, barring use of outside applications, C++ is unavailable for use in the Unity engine, with the only major VR compatible engines that support its use being Unreal and CryEngine. C++ being compiled is situationally a positive or negative factor on performance. While it allows for higher general optimization upon compilation, it lacks the ability to perform runtime querying for hardware specific optimization. Additionally, the primary advantages of C++ in this project are predicated on the use of the Unreal Engine, as C# shares its advantages when using CryEngine, but also includes features that would ease development, such as automated garbage collection.[3]

1.3.3 JavaScript

JavaScript as a whole maintains a similar level of functionality and performance with C#, but lacks several key features. C# gives a finer level of control over process management, while JavaScript obscures this more with its function structure. Where C# has native hardware querying, Javascript requires the use of separate libraries to perform the same action. One aspect that may hinder stability, but improve modularizability within code is that while C# is strongly, statically typed, JavaScript is weekly and dynamically typed, allowing for more flexibility in the code. Additionally, javaScript would limit our choice in game engine to Unity.

1.4 Discussion

When compared to JavaScript, C# is generally equal or superior in performance, has additional functionality, and it is compatible with both CryEngine and Unity. C++, meanwhile, has similar levels of functionality, but lacks automated garbage collection, inferior heap compression, and has no significant difference in performance. However, should the decision be made to build the project using Unreal, C++ becomes the default choice. JavaScript, meanwhile. would be an inferior choice for whatever engine is chosen, given the requirements of the project. By contrast, C# has the required functionality and efficiency, and is natively compatible with both CryEngine and Unity.

1.5 Decision

While the choice of scripting language must be at least partially informed by the choice of game engine and other interdependent systems, by most measures, C# is the optimal language for the needs of the project. The use of C# would provide flexibility in making design choices, as it is natively compatible with all three of our prospective game engines. The only exception to this would be if the team decides to use the Unreal engine, which would necessitate the use of C++.

2 Graphics API

2.1 Overview

Due to the focus on graphics and system performance, the choice of graphical API is highly significant to the development of the project. In order to maintain a suitable visual fidelity without reducing performance, the ability to modify and construct back-end rendering software is necessary. To accomplish this, the team will use a graphical API to develop graphical back-end features that can exploit both hardware and software to maintain stable performance.

2.2 Criteria

Whatever API employed must be capable of operating on all levels of the graphics pipeline, ranging from the game engine to the GPU. This will allow the team to develop proprietary rendering methods to better optimize the program. Secondly, the API must be capable of implementing high-performance computations on the GPU as directed by the developer, as this will allow for more efficient voxel lookup and ad hoc rendering. Finally, the API must be compatible with a natively VR-capable game engine.

2.3 Potential Choices

The currently available options include DirectX, Khronos API, and Mantle. DirectX contains the Direct3D graphical API and DirectCompute compute shader compatibility. The Khronos API, which includes OpenGL graphical library, the OpenCL compute shader library, and Vulkan, a low-level graphical API used for high performance computing. Mantle is an AMD based API developed to compete with Direct3D and OpenGL.

2.3.1 Khronos API

The Khronos API is in actuality a set of several interoperable graphical libraries. Primarily, the team would use the OpenGL library to handle graphical programming, given that OpenGL is not only compatible with most VR-capable game engines, but able to operate with an array of shader libraries, such as GLSL and OpenCL. This extensive feature list would allow the team to build an extensive back-end for handling not only rendering, but high-performance computations for features such as the previously mentioned voxel lookup. One of the primary differences between OpenGL and Direct3D (the DirectX graphical API) is the way that hardware resources are handled. OpenGL (and the related low-level libraries) requires a proprietary hardware implementation, whereas Direct3D allows this to be handled by the application. This would mean that using OpenGL likely decrease the ease of development, but allow for greater optimization. This would also be offset by the fact that the team is most familiar with OpenGL.

2.3.2 DirectX API

The DirectX API, specifically Direct3D, is a developer-centric proprietary API designed for Windows devices, and supported by most major game engines. One of the primary advantages of Direct3D is its extensive driver support (specifically when compared to OpenGL). While not as varied as the Khronos API, DirectX also has support for the HLSL shader language, as well as DirectCompute, allowing for the development of high performance compute shaders. As mentioned above, Direct3D expects hardware resources to be managed by the application. While this reduces flexibility on hardware resource management, it allows for easy, and often more efficient, resource allocation.

2.3.3 Mantle

Mantle is a low-overhead rendering API developed by AMD as a competitor to Direct3D and OpenGL. Designed as a high performance, low level API, Mantle has been used to create several high performance graphics engines. However, despite its performance capabilities, Mantle has largely been abandoned as a graphical API, with much of its technology and designed having been folded into Vulkan. Effectively, any capabilities of Mantle can be performed by the Khronos or DirectX API.

2.4 Discussion

Ultimately, the choice comes down to DirectX or Khronos. Mantle, while fairly powerful, not only lacks the feature completeness of either of its counterparts, but is effectively overshadowed by Vulkan, which maintains most of Mantle's technical capabilities. The performance of OpenGL and DirectX is relatively comparable, depending primarily on the specific hardware (and more specifically, the drivers used). Both also have compute shader capability. However, OpenGL has a wider breadth of available plug-ins, as well as access to Vulkan. Additionally, the fact that OpenGL necessitates the construction of hardware drivers, while a development challenge, allows the team more latitude in low-level hardware development.

2.5 Decision

While DirectX has solid performance and a strong feature support, the Kronos API has extensive support for lower level development. Specifically, the interoperability with Vulkan, OpenCL and GLSL is a significant advantage, given the requirements of the project. As such, the Khronos API, specifically OpenGL, GLSL, OpenCL and Vulkan will be used for development of the project.

3 Rendering Method

3.1 Overview

A key technical decision for the program will be the rendering method chosen. Different rendering methods can result in drastically different displays to the user. This can affect not only the quality and technical capabilities of the program, but the core aesthetic as well. Different rendering methods have different advantages regarding what they are capable of displaying; Some are better at reflective surfaces and lighting, while others can produce sufficient display effects while maintaining a smaller overhead. Deciding what rendering process the program employs will determine several significant strengths and weaknesses of the program at large.

3.2 Criteria

The criteria regarding the rendering method will mainly be two factors: efficiency, and visual fidelity. Rendering methods can generally be seen as a balance between the two. Ultimately, the system must be capable of supporting scene rendering of whatever method is chosen, but with that given, it is in the interest of the project to use whatever method produces the best output. Features such as dynamic lighting, reflections and textured surfaces are of great value to the product as a whole.

3.3 Potential Choices

The three main options for a rendering method include rasterization, ray tracing, and voxel cone tracing. Rasterization renders the scene ”back to front”, rendering whatever objects in view are furthest away, then moving on to next furthest object. Ray tracing simulates the way light operates, calculating what objects can be seen, and how the light interacts with multiple surfaces. Voxel Cone tracing is a form of ray tracing, but where standard ray tracing treats light rays as beams without thickness, voxel cone tracing generates 'beams' with specific measured thickness (specifically as cones).

3.3.1 Rasterization

Rasterization is a common technique in video game development, and is frequently used for its low computational requirements. Generally seen as the 'simple' method of rendering, rasterization typically produces lower-fidelity images when compared to ray tracing and other, more computationally expensive rendering techniques. As rasterization only performs a single pass on each object in a scene, certain visual effects, such as transparency, reflections, and light distortion are technically impossible to perform, as they are dependent on the interactions and interreflections of light between multiple objects. To compensate, rasterization-based rendering systems often employ techniques to simulate these effects, such as using pre-determined mathematical functions to render an image based on a distortion of objects behind it, or storing certain objects in memory, and mapping them to surfaces, to simulate reflections. Ultimately, the use of rasterization would only be made based on necessity, rather than on its technical merits.

3.3.2 Ray Tracing

Ray tracing is a rendering method that typically results in high-fidelity graphical output when compared to traditional rendering techniques. Ray tracing operates by projecting 'view rays' from the position of the camera view (the user) and tracing their path. When a ray comes in contact with an object, the system determines what color the object in question would return to the viewer. With this method, the system can interpolate visual effects such as light refraction through non-opaque objects, real-time reflections, and other complex lighting mechanisms by tracing the way light bounces from one object to another. Additionally, this allows for calculating interreflective surfaces, something completely impossible for more rudimentary rendering methods. However, ray tracing can be prohibitively computationally expensive. Interreflective calculations in particular, are exeedingly inefficient, as each instance of a light 'bounce' exponentially increases the computational cost of rendering the scene. If the system does not bound these reflections, this can become functionally impossible to model. However, this can be mitigated by setting a system limit on how many 'bounces' can occur for a given ray, limiting the computational complexity at the cost of lost visual fidelity.

3.3.3 Voxel Cone Tracing

Cone tracing is a subtype of ray tracing, with the key difference that unlike standard ray tracing, which generates rays without any defined thickness, it generates discrete rays of a defined volume and thickness. In the case of voxel cone tracing, the rays are measured as discrete volumetric elements. This allows the system to reduce the number of beams necessary to perform a render, effectively creating a tradeoff of visual fidelity for reduced computational requirements. Additionally, cone tracing solves certain issues relating to sampling and aliasing found in standard ray tracing.

3.4 Discussion

As stated above, the core issue is finding the greatest level of visual fidelity available given the technical constraints at hand. Rasterization would result in the most computationally efficient system, but the lowest fidelity, while standard ray tracing would result in the opposite. With that in mind, ray tracing can be further optimized by limiting the scale of light interreflection without too significant a reduction in fidelity (as at a certain point, further modeling of reflections will result in diminishing improvements in lighting effects.) Voxel cone tracing, on the other hand, would provide a higher fidelity output than rasterization, while reducing computational requirements when compared to ray tracing.

3.5 Decision

In order to preserve visual fidelity without unduly increasing computational requirements, cone voxel tracing will be implemented. As stated above, it serves as a median between rasterization and conventional ray tracing. While it would seem plausible to use modern implementations of real-time ray tracing, the requirement of rendering two images simultaneously at a high framerate will likely necessitate the consideration for computational efficiency.