*Florida International University*

*School of Computing and Information Sciences*

Software Engineering Focus

Final Deliverable

Project Title: Interactive Paint

**Team Members:** Alain Galvan, Justin Alvarez, Jacob Leschen, Jose Morgan

**Product Owner(s)**: Francisco Ortega

**Mentor(s)**: Francisco Ortega

**Instructor**: Masoud Sadjadi

The MIT License (MIT)

Copyright (c) *2016 Florida International University*

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

**Abstract**

Interactive Paint is a Windows only Desktop application that allows users to paint on a canvas using various input devices such as a Multi-touch Display, Leap Motion, Intel RealSense and Kinect. It functions as a showcase of what can be done with current generation input devices and provides a framework for future iterations of the project to build upon.

**Table of Contents**

[Introduction](#_30j0zll)

[Current System](#_ivuqz3k63rcs)

[Interactive Paint](#_60sa2h4zcygt)

[Purpose of New System](#_ya4g7dblq7sb)

[Interactive Paint](#_gpb3i7muog7s)

[The goal behind our system is to be able to reuse the components that handle input devices. A future iteration of Interactive Paint should integrate Vulkan and virtual reality so we want the developers spending more time on building a fully fleshed out application than trying to make devices like the RealSense and Kinect work properly.](#_1a153jw3836)

[Obsidian](#_no50ks1m6ky9)

[User Stories](#_2et92p0)

[Implemented User Stories](#_tyjcwt)

[Pending User Stories](#_3dy6vkm)

[Project Plan](#_1t3h5sf)

[Hardware and Software Resources](#_4d34og8)

[Sprints Plan](#_2s8eyo1)

[Sprint 1](#_3bw4jq66gqxd)

[User Story #122 - Create Vulkan Shader Example](#_qvybqsg2qgu3)

[Acceptance Criteria:](#_hxy7givaioae)

[User Story #120 - Create a Draft Vulkan Renderer](#_ystvwoyhjzq0)

[Acceptance Criteria:](#_f44aft4x1kvp)

[User Story #134 - Write Vulkan Notes](#_untrssen9jbj)

[Description:](#_ybz5fmj30213)

[Acceptance Criteria:](#_4u5mjvjel1r)

[User Story #136 - Create plf::colony Package](#_io2cys4q48lj)

[Description:](#_z0wxjlid4e49)

[Acceptance Criteria:](#_ngtlc3hlriu8)

[Sprint 2](#_xumyuq5gjsvk)

[User Story #158 - Create Node Descriptors](#_q1f8nke15y9j)

[Description:](#_vi6f1ov6mp28)

[Acceptance Criteria:](#_mi66uqa3vyhw)

[User Story #159 - Create Node Descriptor Renderer](#_jqipx4fch0t4)

[Description:](#_4wdmayftrth2)

[Acceptance Criteria:](#_c9n2czxb318g)

[User Story #160 - Integrate Boost Hana](#_r419y71sjzdg)

[Description:](#_703wzk8vzu3)

[Acceptance Criteria:](#_p14qc6twx62)

[Sprint 3](#_32j7vwo9enjq)

[Acceptance Criteria:](#_lvscsnfu1hmh)

[User Story #160 - Integrate Boost Hana](#_31x9rm3a96e)

[Description:](#_6rjbsp99bh9y)

[Acceptance Criteria:](#_azbmskknkr05)

[Sprint 4](#_p72aluaswz9b)

[Work Log:](#_tqrefiankxht)

[Work Log:](#_ssr6x7thv1e0)

[Work Log:](#_5vz2lf1uzz2w)

[User Story #172 - Create Mesh Node Descriptor](#_51c65kikseau)

[Description:](#_gd46xuun63br)

[Acceptance Criteria:](#_9xnuztkpnz1)

[User Story #175 - Create B+ Node Traversal Algorithm](#_hw38us40nnh4)

[Description:](#_5w0uj63o66ia)

[Acceptance Criteria:](#_67r6h5b252eo)

[User Story #174 - Create View Component](#_4n9ufrnbuo6u)

[As a developer, we need a View component that implements a subset of the CSS3 specification](#_2iyt09jahyk5)

[Acceptance Criteria:](#_u6d7rif1tit8)

[Implement class as according to the CSS3 specification.](#_9oklq5gabed6)

[User Story #173 - Create Shader Component](#_1nc8fa1mxgcb)

[As a developer, I need a shader component in the library to manage the state of a given set of renderable buffers, and propagate their attributes.](#_2iyt09jahyk5)

[Acceptance Criteria:](#_b8zkj65wltpi)

[Implement class for shader.](#_2iyt09jahyk5)

[Sync implementation with Renderer.](#_2iyt09jahyk5)

[Unit test with GTest.](#_95w6jrjc3m2t)

[Sprint 5](#_gsl91jyu03j)

[Work Log:](#_akw4u83tae01)

[User Story #183 - Raw Vulkan Demo](#_sva41x8jrnx7)

[Acceptance Criteria:](#_bkkihyae261m)

[Related Tasks:](#_cbal4mhfaw4)

[User Story #176 - Create Canvas Component for painting on](#_lr3bjhjfjui)

[Description:](#_xct264crweb4)

[Acceptance Criteria:](#_6enuhc2w66ux)

[Sprint 6](#_oji2b41x4vaw)

[Acceptance Criteria:](#_cvtprup4c53b)

[Work Log:](#_5iova8ni4s9h)

[#179 Port Renderer from Imperative to Declarative Components](#_rpaycm5k7xe)

[Acceptance Criteria:](#_fdpmj8ql0c31)

[Related Tasks:](#_esak1uqnsbb6)

[#178 Create Window Component Adapter Layer](#_cszuy0ci7wt0)

[Acceptance Criteria:](#_tg24jpkkoyod)

[System Design](#_3rdcrjn)

[Architectural Patterns](#_26in1rg)

[Obsidian](#_kfzrkkbzalch)

[System and Subsystem Decomposition](#_lnxbz9)

[LunarG Vulkan SDK](#_3jka9ftgqfgq)

[Instances](#_gb7lnhmkp0ac)

[Physical Devices](#_yicgq5asjcn6)

[Logical Devices](#_y9owbxyul35p)

[Queue](#_gnqeqccs3cj0)

[Window Surface Interface](#_3h4i9d6pw1um)

[Win32 Surfaces](#_wdm30w4t9t9o)

[Color Formats](#_mc2zj4lvur44)

[Render Pass](#_qofc25swl4k4)

[Swapchain](#_fo3r851ciifq)

[View Structures](#_hmzqfy7rb798)

[Frame Buffers](#_8yayhdn7m08i)

[Synchronization](#_led2o22n4wts)

[Command Pool](#_xq3i0la5pnry)

[Descriptor Pool](#_ngfu8aofkem3)

[Vertex Buffers](#_5bk1gf5eq8t3)

[Descriptor Sets](#_matmj7devrfk)

[Pipeline Layouts](#_bpgu2828shu1)

[Pipeline State Objects](#_dph6oq64hfho)

[Graphics Pipeline](#_7b2w3230ug1d)

[Pipeline Cache](#_vet74z6ibag)

[Dynamic State Objects](#_w3u1fkrjar9z)

[Shaders](#_i5e671memxrc)

[Command Buffer](#_ws6tzzn39dx9)

[Deployment](#_35nkun2)

[Design Patterns](#_1ksv4uv)

[System Validation](#_44sinio)

[Glossary](#_2jxsxqh)

[Appendix](#_z337ya)

[Appendix A - UML Diagrams](#_3j2qqm3)

[Appendix B - User Interface Design](#_2xcytpi)

[Appendix C - Sprint Review Reports](#_r53iyzxnhx6w)

[Sprint 1](#_sjmb0xhxiapr)

[Sprint 2](#_ls4nyn4y3lqj)

[Sprint 3](#_mlaisq4kw7i7)

[Sprint 4](#_1yzgkif65dtf)

[Sprint 5](#_r1umx9lob44b)

[Sprint 6](#_23yvj2jna89v)

[Appendix D - User Manuals, Installation/Maintenance Document, Shortcomings/Wishlist Document and other documents](#_6w97znsp1i6t)

[Obsidian Installation](#_iha9783tw6dd)

[References](#_2bn6wsx)

# 

# 

# 

# 

# Introduction

This document shows the design and architecture of Interactive Paint. This will cover our current system design and the purpose behind it. This also includes the user stories that we had to guide us throughout the implementation of this application. We cover the hardware and software that was used to make this application. Finally we show the test cases we used to validate our system.

## 

## Current System

### Interactive Paint

Interactive Paint’s system is a similar to the previous iteration but with a focus on decoupling the components that handle interactions with input devices.

In addition, in an attempt to take advantage of the new Vulkan API, the current iteration of the project features Obsidian, a new Vulkan library for C++.

## Purpose of New System

The final purpose of our new system was to separate into individualized groups focusing on one of 3 main aspects: Painting Software, Vulkan, and Virtual Reality, at the request of our product owner. As such, many of our functions are not fully integrated, but have individualized components with complete systems and demos.

### Interactive Paint

The goal behind our system is to be able to reuse the components that handle input devices. A future iteration of Interactive Paint should integrate Vulkan and virtual reality so we want the developers spending more time on building a fully fleshed out application than trying to make devices like the RealSense and Kinect work properly.

### Obsidian

Obsidian is a C++ Vulkan powered component based rendering library intended for use in future Interactive Paint applications, inspired by React Native. It uses functional composition to build a scene graph and updates it using the Flux architecture.

# User Stories

The following section provides the detailed user stories that were implemented in this iteration of the Interactive Paint project. These user stories were approved by Dr. Francisco Ortega and served as the basis for the implementation of the project’s features.

## Implemented User Stories

**User Story #128 - Learning / Reviewing STL**

As a developer, I would like to learn about STL so I can make efficient user applications.

**User Story #135 - Learning Nano GUI**

As a developer, I would like to learn about nanogui, so that I can make a good looking application.

**User Story #137 - Learning and Reviewing C++**

Not having worked with C++ in a while I want to review it as well as learn about newer things in C++ like the conan package manager and functional programming in C++.

**User Story #138 - Research WinAPI**

Since we are going to be implementing touch as a form of input I want to see how touch is handled using Window’s API, I also want to see what other types of interactions it can handle.

**User Story #139 - Research OpenGL**

Since the app will mostly be built with OpenGL at the beginning I want to understand it well.

**User Story #146 - Review C++**

As a developer, I want to review C++ in order to make sure I know how to leverage the language in my favor.

**User Story #147 - Learn libCinder**

As a developer, I want to learn libcinder so I can deliver a smooth user interface.

**User Story #149 - Learn NanoGUI - 2**

As a developer, I want to thoroughly understand nanogui so I can make a more expressive and better designed application.

**User Story #150 - Possible conan integration**

As a developer, I want dev configuration to take as little time as possible, so that I can spend more time actually writing code.

Conan.io is a C++ package manager that makes installing packages on any platform a breeze

**User Story #151 - Library integration**

As a developer, I want to get libcinder and nanogui working together so I can leverage their widget systems.

**User Story #154 - Building GUI using NanoGUI**

As a user I need the application to have a GUI in order for me to paint in the application.

**User Story #162 - Finish integrating Cinder and nanogui**

As a developer, I want to use nanogui for controls and Cinder as a GL abstraction so I can be more efficient.

**User Story #166 - Design how the app will handle concurrent users**

As a user I would like to be able to draw using multiple input devices at the same time as well as have other users draw with me at the same time.

**User Story #201 - Make input devices work with application**

As a user I would want the application to work with multiple input devices such as the Kinect and the Leap Motion.

**User Story #202 - Add tools to new Cinder integrated project**

As a user, I want to be able to use some basic tools (New, Save, Line, Rotate, and color select).

**User Story #212 - Add more tools**

As a user, I would like to have a square creation tool, a circle creation tool. I would also like the save tool to actually work (finished from last sprint).

**User Story #225 - Continue integrating input devices**

As a user I would like to be able to use a full array of input devices with this application.

**User Story #229 - Add more icons**

As a user, I want the tools to be easy to distinguish.

**User Story #235 - Implement gesture controls**

As a user I’d like to be able to use gestures supported by the input devices with the application.

**User Story #241 - Integrate our work**

As developers, Jose and I were working on slightly different things for the past sprints, but they need to be unified now so we can be ready for the final presentation.

**User Story #242 - Add more tools (part 2)**

As a user, I would like to have an endpoint select line tool, a text tool, and an open button.

**User Story #133 - Basic SteamVR and Unity Knowledge**

As a developer, I would like to have a good understanding of the Unity environment with the SteamVR plugin, so that I can begin development on the project using HMD devices.

**User Story #117 - Basic HMD display functionality**

As a user, I would like to display using HMD devices (such as HTC Vive), so that they can be used in the project as output.

**User Story #131 - Basic VR world emulation**

As a developer, I would like to have a functioning VR world displayed with the HTC Vive, in order to implement it within the painting program.

**User Story # 108 - Unity Scene 1**As a developer, I would like to have a scene in unity that can be used for testing basic unity functions, as well as HMD support

**User Story #89 - Unity Scene 2**As a developer, I would like a second unity scene, with more functions, and the ability to switch between scenes, so that I can emulate the user experience for the project.

**User Story #107 - HMD controller functions**As a user, I would like to have multiple functions for the button scheme on the HTC Vive controller, so that I can interact with the 3D world

**User Story #230 - Create an HMD game framework**As a user, I would like to have a game using the HMD, so that the features implemented can be put to a functional use.

## Pending User Stories

**User Story #144 - Unity Integration**

As a developer, I want to have the group’s painting project work integrated into unity, so that my own work (HMD devices) can be used as output.

**User Story #119 - Saving and Loading (Unity)**

As a user, I want to save and load files of painting projects, so that I can continue working on an item across multiple sessions

**User Story #96 - Model Integration**

As a developer, I would like to implement the 3D models created by my fellow programmers into the program, so that the user can interact with custom models during their painting session

# Project Plan

This section describes the planning that went into the realization of this project. This project incorporated the agile development techniques and as such required the sprints to be planned. These sprint plannings are detailed in the section. This section also describes the components, both software and hardware, chosen for this project.

## Hardware and Software Resources

The following is a list of all hardware and software resources that were used in this project:

**Hardware:**

Leap Motion - A motion sensor that tracks hand gestures and hand location.

Intel RealSense - A depth camera that tracks facial gestures as well as hand location.

Microsoft Kinect - A camera that tracks a user’s entire body.

Multi-touch Display - A screen that recognizes up to 10 simultaneous touches.

HTC Vive - A Head-Mounted Display emulating 3D visual space and can track the user’s movements with 2 controllers, as well as the headset.

**Software:**

C++ - The application was built using C++17.

Visual Studio - Used as the environment for development.

Cinder - A library that abstracts OpenGL calls.

Unity - The HMD framework was built using the Unity engine, which has plugin support for the headsets

SteamVR - Application software for most current HMD devices

SteamVR (unity plugin) - plugin for Unity to provide basic HMD functionality

Virtual Reality Toolkit - plugin for Unity to provide streamlined headset and controller functions

C# - The Unity scripts were built using C# 6.0

## Sprints Plan

### Sprint 1

**User Story #128 - Learning / Reviewing STL**

As a developer, I would like to learn about STL so I can make efficient user applications.

Acceptance Criteria:

1. I feel comfortable with STL and (some) design principles
2. I feel comfortable with OO design principles
3. I feel comfortable with the standard C++ containers

**User Story #135 - Learning Nano GUI**

As a developer, I would like to learn about nanogui, so that I can make a good looking application.

**Acceptance Criteria:**

1. I feel comfortable setting up nanogui
2. I can get nanogui compiling
3. I can get the examples working

**User Story #146 - Review C++**

As a developer, I want to review C++ in order to make sure I know how to leverage the language in my favor.

**Acceptance Criteria:**

1. Feel comfortable with C++
2. Feel comfortable with the toolchain (on at least Linux or Windows)
3. Get a better understanding of the conan package manager

**User Story #148 - Design Software**

As a developer, I want the software to be as flexible as possible (mainly so that the rendering backend can be swapped for Vulkan eventually)

**Acceptance Criteria:**

1. Meet with Francisco 2 times for 2 hours each time
2. Think of where to “split” the UI from the rendering “lib”/interface
3. Think of possible models to use to make the abstractions easier to understand

**User Story #137 - Learning and Reviewing C++**

Not having worked with C++ in a while I want to review it as well as learn about newer things in C++ like the conan package manager and functional programming in C++.

Acceptance Criteria:

1. Be able to build and compile C++ applications that leverage all the features of C++.
2. Be comfortable implementing complex algorithms in C++.
3. Get the conan package manager working.

**User Story #138 - Research WinAPI**

Since we are going to be implementing touch as a form of input I want to see how touch is handled using Window’s API, I also want to see what other types of interactions it can handle.

Acceptance Criteria:

1. Know how to implement touch support in an app.
2. Know all the other interactions I can handle using the API.
3. Be able to comfortably leverage the API.

**User Story #139 - Research OpenGL**

Since the app will mostly be built with OpenGL at the beginning I want to understand it well.

Acceptance Criteria:

1. Be able to understand the basics of how OpenGL works.
2. Know what kind of operations are most efficient with OpenGL.

**User Story #133 - Basic SteamVR and Unity Knowledge**

As a developer, I would like to have a good understanding of the Unity environment with the SteamVR plugin, so that I can begin development on the project using HMD devices  
**Acceptance Criteria:**

1. Have Unity and SteamVR plugin installed in local environment
2. Be able to produce demonstrable code in unity
3. Export code to HMD device once it arrives.

# User Story #[122](https://fiu-scis-seniorproject.mingle.thoughtworks.com/projects/multi_modal_interactive_paint/cards/122) - Create Vulkan Shader Example

As a developer, based off work from the Renderer, create a shader example that provides a clear workflow to working with Shaders with IDE support.

**Acceptance Criteria:**

1. Startup Vulkan
2. Create Window
3. Setup Swapchain
4. Create DescriptorSet

**User Story #121 - Create a Vulkan Validation Layer Conan Package**

As a developer, to make development easier, fork your Vulkan SDK package to simply include validation layers.

### Acceptance Criteria:

1. Publish package on Github.
2. Release on Conan.io
3. Download Remotely

# User Story [#120](https://fiu-scis-seniorproject.mingle.thoughtworks.com/projects/multi_modal_interactive_paint/cards/120) - Create a Draft Vulkan Renderer

As a Developer, I want to be able to create a new instance of a Vulkan Renderer, which follows the [actor model](http://bradfrost.com/blog/post/atomic-web-design/) for renderable instances.

This would be a simple OOP renderer, no complex abstractions.

### Acceptance Criteria:

1. Draft design.
2. Implement with Visual Studio
3. Push to Gitlab.

# User Story [#134](https://fiu-scis-seniorproject.mingle.thoughtworks.com/projects/multi_modal_interactive_paint/cards/134) - Write Vulkan Notes

As a developer, I need a thorough understanding of all of Vulkan.

### Acceptance Criteria:

1. Read Vulkan Essentials
2. Read The Vulkan 1.0.24 spec.
3. Write a set of notes.

# User Story [#136](https://fiu-scis-seniorproject.mingle.thoughtworks.com/projects/multi_modal_interactive_paint/cards/136) - Create plf::colony Package

As a developer, I want to use conan to install plf::colony in my project.

### Acceptance Criteria:

1. Create package
2. Publish on Conan.io
3. Download and test

### Sprint 2

**User Story #147 - Learn libCinder**

As a developer, I want to learn libcinder so I can deliver a smooth user interface.

**Acceptance Criteria:**

1. Understand libcinder and its purpose
2. Understand how to use nanogui and cinder together
3. Investigate how the previous group leveraged libcinder

**User Story #149 - Learn nanogui - 2**As a developer, I want to thoroughly understand nanogui so I can make a more expressive and better designed application.

**Acceptance Criteria:**

1. I can understand the more complex examples on their github
2. I can make my own test application that looks like a skeleton of a real application
3. I understand what different kinds of widgets are available

**User Story #150 - Possible conan integration**

As a developer, I want dev configuration to take as little time as possible, so that I can spend more time actually writing code.

Conan.io is a C++ package manager that makes installing packages on any platform a breeze

**Acceptance Criteria:**

1. I understand the strengths and (possible) weaknesses of using conan
2. I understand how difficult it might be to get nanogui and libcinder working with conan

**User Story #151 - Library integration**

As a developer, I want to get libcinder and nanogui working together so I can leverage their widget systems.

**Acceptance Criteria:**

1. Create a project with libcinder and nanogui compiling together
2. Put the project on github (hopefully with libcinder and nanogui as submodules).
3. Use cmake or conan to build the project on Linux/macOS/Windows

**User Story #154 - Building GUI using NanoGUI**

As a user I need the application to have a GUI in order for me to paint in the application.

Acceptance Criteria:

1. Have a fully functional GUI for the app.
2. Document the GUI so that anyone can change it.
3. Build it modular so it can be easily extended.

**User Story #117 - Basic HMD display functionality**

As a user, I would like to display using HMD devices (such as HTC Vive), so that they can be used in the project as output  
**Acceptance Criteria**:

1. HTC Vive display compatibility – including 3D display and head-tracking functionality
2. 3D world emulation on display
3. Integration with project when available

**Tasks:**

1. Documentation
2. Testing
3. HMD device availability

**User Story #108 - Unity Scene 1**

As a developer, I would like to have a scene in unity that can be used for testing basic unity functions, as well as HMD support

**Acceptance Criteria:**

1. Unity scene, complete with usable camera and KB+Mouse controls
2. Objects like light sources and moving models, with scripts for physics
3. Functionality support for HMD devices

**Tasks:**

1. Testing
2. Documentation
3. Unity shapes colors and transformations

As a user, I would like to display simple shapes, colors, and transformations using and HMD device, so that these techniques can be integrated in the multi-modal painting project

**Acceptance Criteria:**

1. Basic shapes can be drawn and seen in Unity world
2. Colors and/or textures can be applied to objects
3. Animation (transformations) can be applied to objects
4. Unity GUI

As a developer, I would like to have a framework GUI in the unity world, in order to implement saving, loading, and other features during the program

**Acceptance Criteria:**

1. Unity UI visible and intractable
2. Buttons for saving, loading, quitting, and changing the scene
3. Buttons for changing the paint color and too

# User Story [#158](https://fiu-scis-seniorproject.mingle.thoughtworks.com/projects/multi_modal_interactive_paint/cards/158) - Create Node Descriptors

As a developer, I want the Vulkan Abstraction to use node descriptors to build vulkan applications from a set of nested components.

### Acceptance Criteria:

1. Create Abstraction Specification
2. Implement in C++
3. Test using Google’s GTest Framework.

# User Story [#159](https://fiu-scis-seniorproject.mingle.thoughtworks.com/projects/multi_modal_interactive_paint/cards/159) - Create Node Descriptor Renderer

As a Developer, create a Node Descriptor Renderer that takes a tree of nodes, based on the OOP Renderer developed last sprint.

### Acceptance Criteria:

1. Review/Design Specification.
2. Implement.
3. GTest the renderer.

# User Story [#160](https://fiu-scis-seniorproject.mingle.thoughtworks.com/projects/multi_modal_interactive_paint/cards/160) - Integrate Boost Hana

As a developer, I want to adopt C++ 14 functional paradigms, so we’ll use boost hana to manage tuples of data.

### Acceptance Criteria:

1. Add boost 1.61 to your dependencies.
2. Create implementation that uses boost::hana
3. Unit Test with GTest.

### Sprint 3

**User Story #166 - Design how the app will handle concurrent users**

As a user I would like to be able to draw using multiple input devices at the same time as well as have other users draw with me at the same time.

Acceptance Criteria:

1. Design must be able to be easily adapted for next semester when the app will switch to Vulkan.
2. Needs to work around the limitation that rendering cannot be made concurrent.
3. Multiple users should be able to draw at the same time.
4. Multiple devices should be usable at the same time.

**User Story #162 - Finish integrating Cinder and nanogui**

As a developer, I want to use nanogui for controls and Cinder as a GL abstraction so I can be more efficient

**Acceptance Criteria:**

1. A project that can be compiled, using nanogui and libcinder
2. A demonstration of nanogui controls and libcinder rendering
3. Works on at least Windows and macOS

**User Story #202 - Add tools to new Cinder integrated project**

As a user, I want to be able to use some basic tools (New, Save, Line, Rotate, and color select).

**Acceptance Criteria:**

1. All tools work
2. All tools can be composed together
3. Any other criteria specified in the sub-tasks are met

**Tasks:**

1. Testing
2. Documentation
3. Color selection tool:

As a user, I would like to be able to select different colors so I can draw colorful pictures.

**Acceptance Criteria:**

1. User can pick any color
2. The interface is intuitive
3. Multiple colors per drawing
4. Rotation tool:

As a user, I would like to be able to press a button to rotate the canvas.

**Acceptance Criteria:**

1. Canvas rotates
2. User can save the rotated canvas back to a file
3. If pressed 4 times, the canvas will return to its original state
4. Line drawing tool:

As a user, I want to draw a line on a canvas

**Acceptance Criteria:**

1. User is able to draw a line on an opened picture file
2. User can save the file back to disk
3. Works with mouse and touch input
4. Saving and loading files:

As a user, I would like to be able to open a picture file (.png, .jpg, etc) in the interactive paint program. (Will use a nanogui control to load the file and display it using cinder)

**Acceptance Criteria:**

1. Works on all OSes that we are supporting
2. It is fast
3. Has an intuitive user interface

**User Story # 131 - Basic VR World Emulation**

As a developer, I would like to have a functioning VR world displayed with the HTC Vive, in order to implement it within the painting program

### Acceptance Criteria:

1. 3D emulated world viewable with HTC vive
2. Head-tracking functionality implemented
3. Player can move around in world with some input device, such as a game controller or keyboard+mouse

**Tasks:**

1. Documentation
2. Testing
3. HMD object display and manipulation:

As a developer, I would like to implement intractable objects within an HMD world, so that they can emulate real-world objects in the final program

**Acceptance Criteria:**

1. 3D objects displayable within HTC Vive
2. Objects have emulated physics and can be manipulated via controller input
3. Objects are placed within emulated 3D world

4. HTC Vive controller manipulation:

As a user, I would like to use the packaged controllers with HTC Vive, in order to implement another mode of drawing into the program.  
**Acceptance Criteria:**

1. Vive controller compatibility as input: includes button inputs and motion-tracking controls.
2. Drawing implementation with controllers.
3. Integration with unity plugin for the project.

# User Story [#160](https://fiu-scis-seniorproject.mingle.thoughtworks.com/projects/multi_modal_interactive_paint/cards/160) - Integrate Boost Hana

As a developer, I want to adopt C++ 14 functional paradigms, so we’ll use boost hana to manage tuples of data.

### Acceptance Criteria:

1. Add boost 1.61 to your dependencies.
2. Create implementation that uses boost::hana
3. Unit Test with GTest.

### Sprint 4

**User Story #201 - Make input devices work with application**

As a user I would want the application to work with multiple input devices such as the Kinect and the Leap Motion.

Acceptance Criteria:

1. Multiple input devices should be working with the application.
2. Implementation should be done in such a way that it can be easily switched over to Vulkan once that is ready.

**User Story #212 - Add more tools**

As a user, I would like to have a square creation tool, a circle creation tool. I would also like the save tool to actually work (finished from last sprint).

**Acceptance Criteria:**

1. All tools work
2. All tools work together
3. Any other criteria specified in the sub-tasks are met

**Tasks:**

1. Testing
2. Documentation
3. Fix saving tool:

Finish the save tool from last sprint.

1. Square tool:

As a user, I would like to be able to draw a square using a simple tool.

**Acceptance Criteria:**

1. There is a live preview for the shape being drawn
2. When the mouse button / touch ends, you can change tools, or drawn another shape
3. Circle tool:

As a user, I would like to be able to draw a circle using a simple tool.

**Acceptance Criteria:**

1. There is a live preview for the shape being drawn
2. When the mouse button / touch ends, you can change tools, or drawn another shape

**User Story #89 - Unity Scene 2**

As a developer, I would like a second unity scene, with more functions, and the ability to switch between scenes, so that I can emulate the user experience for the project.

**Acceptance Criteria:**

1. Objects like light sources, movable models with physics, intractable objects, movement and camera controls
2. Able to view on HMD device and use HMD control scheme
3. Able to switch between scenes in unity using UI

**Tasks:**

1. Testing
2. Documentation
3. Change Scene

Add functionality to change between unity scenes

### Work Log:

* Create a “change scene” button
* If time permits, create a loading zone to swap between scenes, or a map selection scene to change between all scenes later on

1. Lights and Colors

Add light sources and color manipulation in unity scene

### Work Log:

* Add directional, ambient and other light sources to the scene
* Allow user to manipulate light sources
* Let user change the color of objects and light sources

1. Objects and Tools

Create object manipulation, controller support, and tools for HMD device in scene 2

### Work Log:

* Allow HMD controls to pick up objects and use them
* Allow HMD controllers to have weight and physics in world
* Code movement controls for HMD controllers

# User Story [#172](https://fiu-scis-seniorproject.mingle.thoughtworks.com/projects/multi_modal_interactive_paint/cards/172) - Create Mesh Node Descriptor

As a developer, I want to flesh out the mesh node descriptor component to be able to render axiomatic components.

### Acceptance Criteria:

1. Implement class based on specification.
2. Sync implementation with renderer.
3. Unit Test with GTest

# User Story [#175](https://fiu-scis-seniorproject.mingle.thoughtworks.com/projects/multi_modal_interactive_paint/cards/175) - Create B+ Node Traversal Algorithm

As a developer, I need an algorithm that would allow me to traverse the tree of components.

### Acceptance Criteria:

1. Design algorithm spec, look into different implementations.
2. Implement Algorithm.
3. GTests for Algorithm.

# User Story [#174](https://fiu-scis-seniorproject.mingle.thoughtworks.com/projects/multi_modal_interactive_paint/cards/174) - Create View Component

### As a developer, we need a View component that implements a subset of the CSS3 specification

### Acceptance Criteria:

### Implement class as according to the CSS3 specification.

# User Story [#173](https://fiu-scis-seniorproject.mingle.thoughtworks.com/projects/multi_modal_interactive_paint/cards/173) - Create Shader Component

### As a developer, I need a shader component in the library to manage the state of a given set of renderable buffers, and propagate their attributes.

### 

### Acceptance Criteria:

### Implement class for shader.

### Sync implementation with Renderer.

### Unit test with GTest.

### 

### Sprint 5

**User Story #225 - Continue integrating input devices**

As a user I would like to be able to use a full array of input devices with this application.

Acceptance Criteria:

1. All remaining input devices should be working with the application.
2. Input devices should be integrated in such a way that they can be extended when the move to Vulkan happens.

**User Story #229 - Add more icons**

As a user, I want the tools to be easy to distinguish.

**Acceptance Criteria:**

1. All of the old icons work
2. Integrated with NanoGUI in the same way as the existing icons
3. Document how to do this process and contribute it back to the NanoGUI repository

**Tasks:**

1. Testing
2. Documentation

**User Story #241 - Integrate our work**

As developers, Jose and I were working on slightly different things for the past sprints, but they need to be unified now so we can be ready for the final presentation

**Acceptance Criteria:**

1. Program compiles
2. Both of our changes are working
3. All merge conflicts are resolved

**Tasks:**

1. Testing
2. Documentation

**User Story #107 - HMD Controller Functions**

As a user, I would like to have multiple functions for the button scheme on the HTC Vive controller, so that I can interact with the 3D world

**Acceptance Criteria:**

1. All buttons mapped to some function
2. Can pick up and manipulate objects
3. Can equip and use tools with controllers (such as a gun)

**Tasks:**

1. Testing
2. Documentation
3. Teleport User

As a user, I would like to have multiple functions for the button scheme on the HTC Vive controller, so that I can interact with the 3D world

### Work Log:

* Research teleportation methods in unity
* Create teleport function
* Map teleportation controls to button on Vive Controller
* Create visual transition for movement
* Create trajectory model

1. Tools

Create a tool that can be equipped and then used via the controller

**Work Log:**

* Create equip function
* Develop use function
* Code unequip function
* Develop objects that interact with the use function

1. Realistic Grab Physics

Create physics system for picking up and putting down objects that emulate real world physics

* 1. Objects should collide with other physical items at all times
  2. Objects should be able to rotate with the user’s hand motions
  3. Objects that have more weight should be more difficult to pick up

**Work Log:**

* Finish pick up function
* Develop physics model for objects to meet specifications
* Program objects to have appropriate weight
* Code rotation function

# User Story [#183](https://fiu-scis-seniorproject.mingle.thoughtworks.com/projects/multi_modal_interactive_paint/cards/183) - Raw Vulkan Demo

As a user, I want to be able to understand the Obsidian library.

As a product owner, I want a demo.

### Acceptance Criteria:

1. Create a demo using the library
2. Create a Powerpoint for the library
3. Unit test with G-Test

# User Story [#176](https://fiu-scis-seniorproject.mingle.thoughtworks.com/projects/multi_modal_interactive_paint/cards/176) - Create Canvas Component for painting on

As a user, I want a canvas to be able to paint on.

As a developer, I need my canvas to accept WM\_POINTER events to support wacom/touch devices.

### Acceptance Criteria:

1. Hook into the window’s WM\_POINTER events.
2. Abstract events to simplify usage.
3. Unit test with GTest.

### Sprint 6

**User Story #235 - Implement gesture controls**

As a user I’d like to be able to use gestures supported by the input devices with the application.

Acceptance Criteria:

1. Many of the common gestures should be functioning with the application.
2. Gestures should be handled concurrently by different devices.
3. This should be implemented in such a way where it can be extended when it is time to move to Vulkan.

**User Story #242 - Add more tools (2)**

As a user, I would like to have an endpoint select line tool, a text tool, and an open button.

**Acceptance Criteria:**

1. All tools work
2. All tools work together
3. Any other criteria specified in the sub-tasks are met

**Tasks:**

1. Testing
2. Documentation
3. Open tool:

As a user, I would like to be able to open pictures and edit them instead of starting fresh every time.

**Acceptance Criteria:**

1. Opening the picture doesn’t clear the screen
2. At least PNGs should be openable
3. Text tool:

As a user, I would like to be able to write text to the screen.

**Acceptance Criteria:**

1. Input should be handled by NanoGUI and output onto the Cinder context.
2. Any characters in a standard should be writable.
3. Line tool:

As a user, I would like to be able to draw a straight line to the screen (point to point select)

**Acceptance Criteria:**

1. Should work with color select tool
2. Should be live previewable

**User Story #230 - Create an HMD game framework**

As a user, I would like to have a game using the HMD, so that the features implemented can be put to a functional use.

**Acceptance Criteria:**

1. Game uses HMD and controller functions
2. Game has a defined win state
3. Game can be implemented and finished in time for final presentation

**Tasks:**

1. Documentation
2. Testing
3. Brainstorm Game Ideas

Design a rule-set for game that takes advantage of all HMD features

### Acceptance Criteria:

* Game must use teleportation, interactable and usable objects
* Game must be educational in some capacity
* Game should be polished and presentable for final deliverable

1. Maze Game

Create a maze that takes advantage of all features

### Work Log:

* Create a win condition and a lose condition
* Create item that can be used and grabbed that must be taken with the user
* Create gates that are mutually exclusive: User, Key shape 1, and Key shape 2
* Arrange a maze that challenges the user to use all features
* Assure that all game ideas can be implemented with current framework

# [#179](https://fiu-scis-seniorproject.mingle.thoughtworks.com/projects/multi_modal_interactive_paint/cards/179) Port Renderer from Imperative to Declarative Components

As a Product Owner, the current renderer is a rough imperative implementation of Vulkan, make it such that you can interface with the renderer with simple props or dispatch functions.

auto app = (  
 // The Renderer Manages the application loop as well as propigates the vulkan context inner components can use to instantiate their data structures.  
 Renderer({},  
 // A Window is an adapter that manages the title, event listeners of a given window.  
 Window({{"title", "My Obsidian App"}},  
 { /\* ...Shaders, Meshes, Text, Views, etc.\*/ }  
 )  
 )  
);

### Acceptance Criteria:

* Implement a Wrapper Vulkan Component
* Implement fundamental Vulkan Controller logic.
* Unit Test with GTest.

# [#178](https://fiu-scis-seniorproject.mingle.thoughtworks.com/projects/multi_modal_interactive_paint/cards/178) Create Window Component Adapter Layer

As a developer, I want to create a component that serves as an adapter for different operating systems, that depending on the `os` prop of the root application’s props, I will render the Win32Window, or the AndroidSurface, etc. Each OS specific window implementation will have the same interface from which you can manipulate it. (resize, getSize, minimize, etc.)

### Acceptance Criteria:

1. Create stateless component.
2. Implement stateless component
3. Unit test with GTest.

# System Design

This section contains information on the design decisions that went into Interactive Paint. The architecture patterns are outlined and explained. The entire system is shown in a package diagram and the subsystems are explained. Finally, the design patterns used in the project are discussed.

## Architectural Patterns

All aspects of the project have taken advantage of primarily the **Actor Model**. This design allowed for each component to behave independently of each other.

**Package Management** -Conan.io manages all our dependencies.

**Template programming** - For compile-time type safety, zero runtime, and variadic types.

**Factory Functions** - avoid new, allows custom allocation schemes.

### Obsidian

For Obsidian, we designed it to be as close to React semantics as possible in C++ while still keeping with the constraints of the language. A React component decorated with Flow/TypeScript would be written as follows:

import React, {Component} from 'react';  
  
type HelloProps = {};  
type HelloState = {};  
  
class HelloWorld extends Component<HelloProps, HelloState> {  
 render() {  
 return (  
 <Text>  
 Hello World  
 </Text>  
 );  
 }  
}  
  
import {render} from 'react-dom';  
  
render(<HelloWorld/>);

Would look like the following in Obsidian after Babel or TypeScript converts the JSX to factory functions:

#include "obsidian.hpp"  
  
using namespace obs;  
  
struct HelloProps {};  
struct HelloState {};  
  
class HelloWorld : public Component<HelloProps, HelloState>  
{  
 HelloWorld(HelloProps props = {}) : Component(props)  
 {  
  
 }  
  
 render() {  
 return (  
 CreateElement<Text>({}, "Hello World")  
 );  
 }  
};  
  
int main()  
{  
 render(CreateElement<HelloWorld>());  
}

However Obsidian takes advantage of the type safety features of C++, thus it's closer looking to TypeScript/Flow or ReasonML then pure JavaScript.

## System and Subsystem Decomposition

The applications are composed of the following dependencies:

* **Obsidian** - A Declarative Component based rendering library for C++ 11 inspired by React Native and using a Vulkan backend.
  + LunarG Vulkan SDK
  + Conan.io
* **Vive Interactive Paint** - An attempt to port the Interactive paint application to the HTC Vive.
  + Unity 3D 5.5
  + Vive SDK
* **Interactive Paint Core** - Additions to the Interactive Painting application, building on top of the latest version of Cinder and new OpenGL view libraries.
  + LibCinder
  + NanoGUI

### LunarG Vulkan SDK

Shortly after the release of Vulkan, the Khronos Group released a development SDK that provides development layers and a distributable runtime of the Vulkan driver to include in any projects that use the SDK.

*The following is a detailed decomposition of the Vulkan API:*

Vulkan is a new low level API released February 2016 by the Khronos Group that maps directly to the design of modern GPUs. OpenGL was designed in 1992 when GPUs were far more simple, but since then they have become programmable computational units of their own with a focus on throughput over latency.

In any given application we will need to do the following:

1. Create a Vulkan Instance to access inner functions of the Vulkan API.
2. Pick the best Physical Device from every device that supports Vulkan on your machine.
3. Create a Logical Device from your physical device to interface with more Vulkan
4. Create OS Window using OS specific APIs.
5. Create a Surface from your window to serve as the OS interface for Vulkan.
6. Create a primary Render Pass to be used in your swapchain and surface.
7. Create a Swapchain from your logical device.
8. Create a set of FrameBuffers for each image in your swapchain.
9. Create Synchronization primitives like semaphores and fences.
10. Create a Command Pool from your logical device.
11. Create a Vertex Buffer and Index Buffer for your geometry.
12. Compile and load SPIR-V shader binary.
13. Create a Graphics Pipeline to represent the entire state of the Graphics Pipeline for that triangle.
14. Create Commands for each command buffer to set the GPU's state.
15. Use an Update Loop to switch between different frames in your swapchain.

#### Instances

Similar to the OpenGL context, a Vulkan application begins when you create an instance. This instance must be loaded with some information about the program such as its name, engine, and minimum Vulkan version, as well any extensions and layers you want to load.

Extension - Anything that adds extra functionality to Vulkan, such as support for Win32 windows, or enabling drawing onto a target.

Layer - Middleware between existing Vulkan functionality, such as checking for errors. Layers can range from runtime debugging checks to hooks to GPU debugging software like [RenderDoc](https://github.com/baldurk/renderdoc) to even hooks to the Steam renderer so your game can behave better when you Ctrl + Shift to switch to the Steam overlay.

You'll want to begin by determining which extensions/layers you want, compare that with which are available to you by Vulkan. Once that's done, continue creating other Vulkan primitives:

auto installedExtensions = vk::enumerateInstanceExtensionProperties();  
  
std::vector<const char\*> wantedExtensions =  
{  
 VK\_EXT\_DEBUG\_REPORT\_EXTENSION\_NAME,  
 VK\_KHR\_SURFACE\_EXTENSION\_NAME,  
 VK\_KHR\_WIN32\_SURFACE\_EXTENSION\_NAME  
};  
  
auto extensions = std::vector<const char\*>();  
  
for (auto &w : wantedExtensions) {  
 for (auto &i : installedExtensions) {  
 if (std::string(i.extensionName).compare(w) == 0) {  
 extensions.emplace\_back(w);  
 break;  
 }  
 }  
}  
  
auto installedLayers = vk::enumerateInstanceLayerProperties();  
  
std::vector<const char\*> wantedLayers =  
{  
 "VK\_LAYER\_LUNARG\_standard\_validation",  
 "VK\_LAYER\_RENDERDOC\_Capture"  
};  
  
auto layers = std::vector<const char\*>();  
  
for (auto &w : wantedLayers) {  
 for (auto &i : installedLayers) {  
 if (std::string(i.layerName).compare(w) == 0) {  
 layers.emplace\_back(w);  
 break;  
 }  
 }  
}  
  
auto appInfo = vk::ApplicationInfo(  
 "MyApp",  
 VK\_MAKE\_VERSION(1, 0, 0),  
 "MyAppEngine",  
 VK\_MAKE\_VERSION(1, 0, 0),  
 VK\_API\_VERSION\_1\_0  
);  
  
auto instance = vk::createInstance(  
 vk::InstanceCreateInfo(  
 vk::InstanceCreateFlags(),  
 &appInfo,  
 layers.size(),  
 layers.data(),  
 extensions.size(),  
 extensions.data()  
 )  
);

#### Physical Devices

In Vulkan, you have access to all enumerable devices that support it, and can query for information like their name, the number of heaps they support, their manufacturer, etc.

// Initialize Devices  
auto physicalDevices = instance.enumeratePhysicalDevices();  
auto gpu = physicalDevices[0];

Note - Multi-GPU processing isn't supported yet on Vulkan (unless 1.1.x is already out when you read this) so this would be useful for choosing the fastest device to use.

#### Logical Devices

You can then create a logical device from a physical device handle. A logical device can be loaded with its own extensions/layers, can be set to work with graphics, gpgpu computations, handle sparse memory and/or memory transfers by creating queues for that device.

A logical device is your interface to the GPU, and allows you to allocate data and queue up tasks.

auto gpuExtensions = gpu.enumerateDeviceExtensionProperties();  
  
// Init Device Extension/Validation layers  
std::vector<const char\*> wantedDeviceExtensions =  
{  
 VK\_KHR\_SWAPCHAIN\_EXTENSION\_NAME,  
 VK\_EXT\_DEBUG\_MARKER\_EXTENSION\_NAME  
};  
  
auto deviceExtensions = std::vector<const char\*>();  
  
for (auto &w : wantedDeviceExtensions) {  
 for (auto &i : gpuExtensions) {  
 if (std::string(i.extensionName).compare(w) == 0) {  
 deviceExtensions.emplace\_back(w);  
 break;  
 }  
 }  
}  
  
auto gpuLayers = gpu.enumerateDeviceLayerProperties();  
  
std::vector<const char\*> wantedDeviceValidationLayers =  
{  
 "VK\_LAYER\_LUNARG\_standard\_validation",  
 "VK\_LAYER\_RENDERDOC\_Capture"  
};  
  
auto deviceValidationLayers = std::vector<const char\*>();  
  
for (auto &w : wantedLayers) {  
 for (auto &i : installedLayers) {  
 if (std::string(i.layerName).compare(w) == 0) {  
 layers.emplace\_back(w);  
 break;  
 }  
 }  
}  
  
  
auto formatProperties = gpu.getFormatProperties(vk::Format::eR8G8B8A8Unorm);  
auto gpuFeatures = gpu.getFeatures();  
auto gpuQueueProps = gpu.getQueueFamilyProperties();  
  
float priority = 0.0;  
uint32\_t graphicsFamilyIndex = 0;  
auto queueCreateInfos = std::vector<vk::DeviceQueueCreateInfo>();  
  
for (auto& queuefamily : gpuQueueProps)  
{  
 if (queuefamily.queueFlags & vk::QueueFlagBits::eGraphics) {  
 // Create a single graphics queue.  
 queueCreateInfos.push\_back(  
 vk::DeviceQueueCreateInfo(  
 vk::DeviceQueueCreateFlags(),  
 graphicsFamilyIndex,  
 1,  
 &priority  
 )  
 );  
 break;  
 }  
  
 graphicsFamilyIndex++;  
  
}  
  
auto device = gpu.createDevice(  
 vk::DeviceCreateInfo(  
 vk::DeviceCreateFlags(),  
 queueCreateInfos.size(),  
 queueCreateInfos.data(),  
 deviceValidationLayers.size(),  
 deviceValidationLayers.data(),  
 deviceExtensions.size(),  
 deviceExtensions.data(),  
 &gpuFeatures  
 )  
);

#### Queue

Once you have a virtual device, you can access the queues you requested when you created it:

// We only allocated one queue earlier,  
//so there's only one available on index 0.  
auto graphicsQueue = device.getQueue(graphicsFamilyIndex, 0);

#### Window Surface Interface

Each OS has their own specific window generation system. Vulkan 1.0 currently supports Windows, Android, and Linux windows out of the box, with plans for iOS and Mac OS in the future.

A surface is an adapter abstraction to describe an area that will render Vulkan to a window, it's the binding between Vulkan and your OS's windowing system.

|  |  |  |  |
| --- | --- | --- | --- |
| **Extension Name** | **Required Compile Time Symbol** | **Window System Name** | **External Header Files Used** |
| VK\_KHR\_android\_surface | VK\_USE\_PLATFORM\_ANDROID\_KHR | Android Native | <android/native\_window.h> |
| VK\_KHR\_mir\_surface | VK\_USE\_PLATFORM\_MIR\_KHR | Mir | <mir\_toolkit/client\_types.h> |
| VK\_KHR\_wayland\_surface | VK\_USE\_PLATFORM\_WAYLAND\_KHR | Wayland | <wayland-client.h> |
| VK\_KHR\_win32\_surface | VK\_USE\_PLATFORM\_WIN32\_KHR | Microsoft Windows | <windows.h> |
| VK\_KHR\_xcb\_surface | VK\_USE\_PLATFORM\_XCB\_KHR | X Window System Xcb library | <xcb/xcb.h> |
| VK\_KHR\_xlib\_surface | VK\_USE\_PLATFORM\_XLIB\_KHR | X Window System Xlib library | <X11/Xlib.h> |

If you want to support multiple platforms, then you'll need to use OS specific preprocessor definitions, and check if they're defined. Alternatively, you could use something like GLFW if you're only interested in supporting Windows and Linux.

#if defined(\_WIN32)  
  
 // Perform Windows specific logic  
  
#elif defined(\_\_ANDROID\_\_)  
  
 // Perform Android specific logic  
  
#elif defined(\_\_linux\_\_)  
  
 // Perform Linux specific logic  
  
#endif

You'll need to keep in mind things like window size, canvas size (supersampling), DPI and retina support, nested windows, window management and spawning multiple windows.

Call to Action - Make a Vulkan library that makes cross platform Vulkan super easy. You can target problems such as how each OS has their own main function, each OS has their own Windowing abstraction, how to manage each and let users access their OS specific handles if needed. One possibility is to add Vulkan support to [SFML](https://github.com/SFML/SFML).

#### Win32 Surfaces

A Win32 surface is created when you include the VK\_KHR\_win32\_surface extension to Vulkan, declare VK\_USE\_PLATFORM\_WIN32\_KHR, and include windows.h in your project.

Creating windows on [Windows is well documented on MSDN](https://msdn.microsoft.com/en-us/library/windows/desktop/ms632680(v=vs.85).aspx), so refer there for any more questions.

#ifdef \_MSC\_VER  
# pragma comment(linker, "/subsystem:windows /ENTRY:mainCRTStartup")  
#endif  
#define VK\_USE\_PLATFORM\_WIN32\_KHR  
#include "windows.h"  
  
#define VULKAN\_HPP\_TYPESAFE\_CONVERSION  
#define USE\_SWAPCHAIN\_EXTENSIONS  
  
#include "vulkan.hpp"  
  
int main()  
{  
 // Setup Instance  
  
 // Setup Phyisical Devices  
  
 // Setup Logical Devices  
  
 // Setup Window  
  
 std::string title = "MyVulkanApp";  
 std::string name = "MyVulkanApp";  
 uint32\_t width = 1280;  
 uint32\_t height = 720;  
 auto hInstance = GetModuleHandle(0);  
  
 WNDCLASSEX wndClass;  
 wndClass.cbSize = sizeof(WNDCLASSEX);  
 wndClass.style = CS\_HREDRAW | CS\_VREDRAW;  
 wndClass.lpfnWndProc = [](HWND h, UINT m, WPARAM w, LPARAM l)->LRESULT  
 {  
 if (m == WM\_CLOSE)  
 PostQuitMessage(0);  
 else  
 return DefWindowProc(h, m, w, l);  
 return 0;  
 };  
 wndClass.cbClsExtra = 0;  
 wndClass.cbWndExtra = 0;  
 wndClass.hInstance = hInstance;  
 wndClass.hIcon = LoadIcon(NULL, IDI\_APPLICATION);  
 wndClass.hCursor = LoadCursor(NULL, IDC\_ARROW);  
 wndClass.hbrBackground = (HBRUSH)GetStockObject(BLACK\_BRUSH);  
 wndClass.lpszMenuName = NULL;  
 wndClass.lpszClassName = name.c\_str();  
 wndClass.hIconSm = LoadIcon(NULL, IDI\_WINLOGO);  
  
 if (!RegisterClassEx(&wndClass)) {  
 fflush(stdout);  
 exit(1);  
 }  
  
 DWORD dwExStyle;  
 DWORD dwStyle;  
  
 dwExStyle = WS\_EX\_APPWINDOW | WS\_EX\_WINDOWEDGE;  
 dwStyle = WS\_OVERLAPPEDWINDOW | WS\_CLIPSIBLINGS | WS\_CLIPCHILDREN;  
  
 RECT windowRect;  
 windowRect.left = 0L;  
 windowRect.top = 0L;  
 windowRect.right = (long)width;  
 windowRect.bottom = (long)height;  
  
 AdjustWindowRectEx(&windowRect, dwStyle, FALSE, dwExStyle);  
  
 auto window = CreateWindowEx(0,  
 name.c\_str(),  
 title.c\_str(),  
 dwStyle | WS\_CLIPSIBLINGS | WS\_CLIPCHILDREN,  
 0,  
 0,  
 windowRect.right - windowRect.left,  
 windowRect.bottom - windowRect.top,  
 NULL,  
 NULL,  
 hInstance,  
 NULL);  
  
 // Center on screen  
 uint32\_t x = (GetSystemMetrics(SM\_CXSCREEN) - windowRect.right) / 2;  
 uint32\_t y = (GetSystemMetrics(SM\_CYSCREEN) - windowRect.bottom) / 2;  
 SetWindowPos(window, 0, x, y, 0, 0, SWP\_NOZORDER | SWP\_NOSIZE);  
  
 if (!window) {  
 printf("Could not create window!\n");  
 fflush(stdout);  
 exit(1);  
 }  
  
 ShowWindow(window, SW\_SHOW);  
 SetForegroundWindow(window);  
 SetFocus(window);  
}

From there we can create our Win32 surface.

// Screen Size  
auto surfaceSize = vk::Extent2D(width, height);  
auto renderArea = vk::Rect2D(vk::Offset2D(), surfaceSize);  
auto viewport = vk::Viewport(0.0f, 0.0f, width, height, 0, 1.0f);  
  
std::vector<vk::Viewport> viewports =  
{  
 viewport  
};  
  
std::vector<vk::Rect2D> scissors =  
{  
 renderArea  
};  
  
auto surfaceInfo = vk::Win32SurfaceCreateInfoKHR(vk::Win32SurfaceCreateFlagsKHR(), hInstance, window);  
auto vkSurfaceInfo = surfaceInfo.operator const VkWin32SurfaceCreateInfoKHR&();  
  
auto vksurface = VkSurfaceKHR();  
auto createwin32surface = vkCreateWin32SurfaceKHR(instance, &vkSurfaceInfo, NULL, &vksurface);  
assert(createwin32surface == VK\_SUCCESS);  
  
// Get surface information  
auto surface = vk::SurfaceKHR(vksurface);

#### Color Formats

Knowing what Color formats your GPU supports will play a crucial role in determining what you can display and what kind of buffers you can allocate.

// Check to see if we can display rgb colors.  
auto surfaceFormats = gpu.getSurfaceFormatsKHR(surface);  
  
vk::Format surfaceColorFormat;  
vk::ColorSpaceKHR surfaceColorSpace;  
  
if (surfaceFormats.size() == 1 && surfaceFormats[0].format == vk::Format::eUndefined)  
 surfaceColorFormat = vk::Format::eB8G8R8A8Unorm;  
else  
 surfaceColorFormat = surfaceFormats[0].format;  
  
surfaceColorSpace = surfaceFormats[0].colorSpace;  
  
  
auto formatProperties = gpu.getFormatProperties(vk::Format::eR8G8B8A8Unorm);  
  
// Since all depth formats may be optional, we need to find a suitable depth format to use  
// Start with the highest precision packed format  
std::vector<vk::Format> depthFormats = {  
 vk::Format::eD32SfloatS8Uint,  
 vk::Format::eD32Sfloat,  
 vk::Format::eD24UnormS8Uint,  
 vk::Format::eD16UnormS8Uint,  
 vk::Format::eD16Unorm  
};  
  
vk::Format surfaceDepthFormat;  
  
for (auto& format : depthFormats)  
{  
 auto depthFormatProperties = gpu.getFormatProperties(format);  
 // Format must support depth stencil attachment for optimal tiling  
 if (depthFormatProperties.optimalTilingFeatures & vk::FormatFeatureFlagBits::eDepthStencilAttachment)  
 {  
 surfaceDepthFormat = format;  
 break;  
 }  
}

#### Render Pass

For deferred rendering solutions, Vulkan makes render passes first class, letting you describe your whole postprocessing system as a list of SubPasses, groupings of rendered data like a color and depth buffer.

* Attachment Description - a description of the image view that will be attached to the subpass.
* Attachment Reference - an index name to refer to the framebuffer attachment accessed by different subpasses.
* Subpass - A phase of rendering within a render pass, that reads and writes a subset of the attachments.
* Subpass Dependency - an execution or memory dependency between different subpasses. This would be for example, the sampler2D that you would access in a post-processing system that is waterfalled down the chain of effects. This list also determines the order that subpasses are used.

std::vector<vk::AttachmentDescription> attachmentDescriptions =  
{  
 vk::AttachmentDescription(  
 vk::AttachmentDescriptionFlags(),  
 surfaceColorFormat,  
 vk::SampleCountFlagBits::e1,  
 vk::AttachmentLoadOp::eClear,  
 vk::AttachmentStoreOp::eStore,  
 vk::AttachmentLoadOp::eDontCare,  
 vk::AttachmentStoreOp::eDontCare,  
 vk::ImageLayout::eUndefined,  
 vk::ImageLayout::ePresentSrcKHR  
 ),  
 vk::AttachmentDescription(  
 vk::AttachmentDescriptionFlags(),  
 surfaceDepthFormat,  
 vk::SampleCountFlagBits::e1,  
 vk::AttachmentLoadOp::eClear,  
 vk::AttachmentStoreOp::eDontCare,  
 vk::AttachmentLoadOp::eDontCare,  
 vk::AttachmentStoreOp::eDontCare,  
 vk::ImageLayout::eUndefined,  
 vk::ImageLayout::eDepthStencilAttachmentOptimal  
 )  
};  
  
std::vector<vk::AttachmentReference> colorReferences =  
{  
 vk::AttachmentReference(0, vk::ImageLayout::eColorAttachmentOptimal)  
};  
  
std::vector<vk::AttachmentReference> depthReferences = {  
 vk::AttachmentReference(1, vk::ImageLayout::eDepthStencilAttachmentOptimal)  
};  
  
std::vector<vk::SubpassDescription> subpasses =  
{  
 vk::SubpassDescription(  
 vk::SubpassDescriptionFlags(),  
 vk::PipelineBindPoint::eGraphics,  
 0,  
 nullptr,  
 colorReferences.size(),  
 colorReferences.data(),  
 nullptr,  
 depthReferences.data(),  
 0,  
 nullptr  
 )  
};  
  
std::vector<vk::SubpassDependency> dependencies =  
{  
 vk::SubpassDependency(  
 ~0U,  
 0,  
 vk::PipelineStageFlagBits::eBottomOfPipe,  
 vk::PipelineStageFlagBits::eColorAttachmentOutput,  
 vk::AccessFlagBits::eMemoryRead,  
 vk::AccessFlagBits::eColorAttachmentRead | vk::AccessFlagBits::eColorAttachmentWrite,  
 vk::DependencyFlagBits::eByRegion  
 ),  
 vk::SubpassDependency(  
 0,  
 ~0U,  
 vk::PipelineStageFlagBits::eColorAttachmentOutput,  
 vk::PipelineStageFlagBits::eBottomOfPipe,  
 vk::AccessFlagBits::eColorAttachmentRead | vk::AccessFlagBits::eColorAttachmentWrite,  
 vk::AccessFlagBits::eMemoryRead,  
 vk::DependencyFlagBits::eByRegion  
 )  
};  
  
auto renderpass = device.createRenderPass(  
 vk::RenderPassCreateInfo(  
 vk::RenderPassCreateFlags(),  
 attachmentDescriptions.size(),  
 attachmentDescriptions.data(),  
 subpasses.size(),  
 subpasses.data(),  
 dependencies.size(),  
 dependencies.data()  
 )  
);

#### Swapchain

A Swapchain is a structure that manages the allocation of frame buffers to be cycled through by your application. It's here that your application sets up V-Sync via double buffering or triple buffering.

One approach to setting this up is to take in a JSON file at the start of your application, say config.json, which determines if you'll be using V-Sync, your screen resolution, any any other global data you want to configure.

auto surfaceCapabilities = gpu.getSurfaceCapabilitiesKHR(surface);  
auto surfacePresentModes = gpu.getSurfacePresentModesKHR(surface);  
  
// check the surface width/height.  
if (!(surfaceCapabilities.currentExtent.width == -1 || surfaceCapabilities.currentExtent.height == -1)) {  
 surfaceSize = surfaceCapabilities.currentExtent;  
}  
  
auto presentMode = vk::PresentModeKHR::eImmediate;  
  
for (auto& pm : surfacePresentModes) {  
 if (pm == vk::PresentModeKHR::eMailbox) {  
 presentMode = vk::PresentModeKHR::eMailbox;  
 break;  
 }  
}  
  
assert(surfaceCapabilities.maxImageCount >= 3);  
auto swapchainCreateInfo = vk::SwapchainCreateInfoKHR();  
swapchainCreateInfo.surface = surface;  
swapchainCreateInfo.minImageCount = 3;  
swapchainCreateInfo.imageFormat = colorFormat;  
swapchainCreateInfo.imageColorSpace = colorSpace;  
swapchainCreateInfo.imageExtent = surfaceSize;  
swapchainCreateInfo.imageArrayLayers = 1;  
swapchainCreateInfo.imageUsage = vk::ImageUsageFlagBits::eColorAttachment;  
swapchainCreateInfo.imageSharingMode = vk::SharingMode::eExclusive;  
  
std::vector<uint32\_t> queueFamilyIdices;  
queueFamilyIdices.push\_back(graphicsFamilyIndex);  
  
swapchainCreateInfo.queueFamilyIndexCount = queueFamilyIdices.size();  
swapchainCreateInfo.pQueueFamilyIndices = queueFamilyIdices.data();  
swapchainCreateInfo.preTransform = vk::SurfaceTransformFlagBitsKHR::eIdentity;  
swapchainCreateInfo.compositeAlpha = vk::CompositeAlphaFlagBitsKHR::eOpaque;  
swapchainCreateInfo.presentMode = presentMode;  
  
auto swapchain = device.createSwapchainKHR(swapchainCreateInfo);  
auto swapchainImages = device.getSwapchainImagesKHR(swapchain);

#### View Structures

A view in Vulkan is an adapter that lets you interface between GPU data structures.

#### Frame Buffers

A frame buffer in Vulkan is a container of Image Views.

// The swapchain handles allocating frame images.  
auto swapchainImages = device.getSwapchainImagesKHR(swapchain);  
  
// Create Depth Image Data  
auto depthImage = device.createImage(  
 vk::ImageCreateInfo(  
 vk::ImageCreateFlags(),  
 vk::ImageType::e2D,  
 surfaceDepthFormat,  
 vk::Extent3D(surfaceSize.width, surfaceSize.height, 1),  
 1,  
 1,  
 vk::SampleCountFlagBits::e1,  
 vk::ImageTiling::eOptimal,  
 vk::ImageUsageFlagBits::eDepthStencilAttachment | vk::ImageUsageFlagBits::eTransferSrc,  
 vk::SharingMode::eExclusive,  
 queueFamilyIndices.size(),  
 queueFamilyIndices.data(),  
 vk::ImageLayout::eUndefined  
 )  
);  
  
// Search through GPU memory properies to see if this can be device local.  
  
auto depthMemoryReq = device.getImageMemoryRequirements(depthImage);  
uint32\_t typeBits = depthMemoryReq.memoryTypeBits;  
uint32\_t depthMemoryTypeIndex;  
  
for (uint32\_t i = 0; i < gpuMemoryProps.memoryTypeCount; i++)  
{  
 if ((typeBits & 1) == 1)  
 {  
 if ((gpuMemoryProps.memoryTypes[i].propertyFlags & vk::MemoryPropertyFlagBits::eDeviceLocal) == vk::MemoryPropertyFlagBits::eDeviceLocal)  
 {  
 depthMemoryTypeIndex = i;  
 break;  
 }  
 }  
 typeBits >>= 1;  
}  
  
auto depthMemory = device.allocateMemory(  
 vk::MemoryAllocateInfo(depthMemoryReq.size, depthMemoryTypeIndex)  
);  
  
  
device.bindImageMemory(  
 depthImage,  
 depthMemory,  
 0  
);  
  
auto depthImageView = device.createImageView(  
 vk::ImageViewCreateInfo(  
 vk::ImageViewCreateFlags(),  
 depthImage,  
 vk::ImageViewType::e2D,  
 surfaceDepthFormat,  
 vk::ComponentMapping(),  
 vk::ImageSubresourceRange(  
 vk::ImageAspectFlagBits::eDepth | vk::ImageAspectFlagBits::eStencil,  
 0,  
 1,  
 0,  
 1  
 )  
 )  
);  
  
struct SwapChainBuffer {  
 vk::Image image;  
 std::array<vk::ImageView, 2> views;  
 vk::Framebuffer frameBuffer;  
};  
  
std::vector<SwapChainBuffer> swapchainBuffers;  
swapchainBuffers.resize(swapchainImages.size());  
  
for (int i = 0; i < swapchainImages.size(); i++)  
{  
 swapchainBuffers[i].image = swapchainImages[i];  
  
 // Color  
 swapchainBuffers[i].views[0] =  
 device.createImageView(  
 vk::ImageViewCreateInfo(  
 vk::ImageViewCreateFlags(),  
 swapchainImages[i],  
 vk::ImageViewType::e1D,  
 surfaceColorFormat,  
 vk::ComponentMapping(),  
 vk::ImageSubresourceRange(  
 vk::ImageAspectFlagBits::eColor,  
 0,  
 1,  
 0,  
 1  
 )  
 )  
 );  
  
 // Depth  
 swapchainBuffers[i].views[1] = depthImageView;  
  
 swapchainBuffers[i].frameBuffer = device.createFramebuffer(  
 vk::FramebufferCreateInfo(  
 vk::FramebufferCreateFlags(),  
 renderpass,  
 swapchainBuffers[i].views.size(),  
 swapchainBuffers[i].views.data(),  
 surfaceSize.width,  
 surfaceSize.height,  
 1  
 )  
 );  
}

#### Synchronization

Vulkan was designed with conccurency in mind, so you're free to use Mutexes, and built in Vulkan Semaphores and Fences for GPU level Synchronization.

Semaphores coordinate operations within the graphics queue and ensure correct command ordering.

// Semaphore used to ensures that image presentation is complete before starting to submit again  
auto presentCompleteSemaphore = device.createSemaphore(vk::SemaphoreCreateInfo());  
  
// Semaphore used to ensures that all commands submitted have been finished before submitting the image to the queue  
auto renderCompleteSemaphore = device.createSemaphore(vk::SemaphoreCreateInfo());  
  
// Fence for command buffer completion  
std::vector<vk::Fence> waitFences;  
waitFences.resize(swapchainBuffers.size());  
for (int i = 0; i < waitFences.size(); i++)  
{  
 waitFences[i] = device.createFence(vk::FenceCreateInfo(vk::FenceCreateFlagBits::eSignaled));  
}

#### Command Pool

A command pool is a means of allocating command buffers. Any number of command buffers can be made from command pools, with you as the developer responsible for managing when and how they're created and what is loaded in each.

A command pool cannot be used in multiple threads, but you can create one for each thread and manage them on a per thread level.

auto commandPoolInfo = vk::CommandPoolCreateInfo(  
 vk::CommandPoolCreateFlags(vk::CommandPoolCreateFlagBits::eResetCommandBuffer),  
 graphicsFamilyIndex  
);  
auto commandPool = device.createCommandPool(commandPoolInfo);  
  
// Lets allocate 1 command buffer for each swapchain image.  
auto commandBuffers = device.allocateCommandBuffers(  
 vk::CommandBufferAllocateInfo(  
 commandPool,  
 vk::CommandBufferLevel::ePrimary,  
 swapchainBuffers.size()  
 )  
);

You should aim to have the minimum number of command buffers possible in your application.

One possible setup could be taking a flat collection of renderable objects (like a scene), distributing it across as many threads as the computer's CPU allows, allocating a command buffer for each object, creating a pipeline for each object, and finishing by sending a ending buffer to start up the process.

We'll come back to the command buffers we made here later in our app.

#### Descriptor Pool

A descriptor pool is a means of allocating Descriptor Sets, a set of data structures containing implementation-specific descriptions of resources. to make a descriptor pool, you need to describe exactly how many of each type of descriptor you need to allocate.

To do that you need to provide a collection of the size of each descriptor type.

std::vector<vk::DescriptorPoolSize> descriptorPoolSizes =  
{  
 vk::DescriptorPoolSize(  
 vk::DescriptorType::eUniformBuffer,  
 1  
 )  
};  
  
auto descriptorPool = device.createDescriptorPool(  
 vk::DescriptorPoolCreateInfo(  
 vk::DescriptorPoolCreateFlags(),  
 1,  
 descriptorPoolSizes.size(),  
 descriptorPoolSizes.data()  
 )  
);

Like command buffers, we'll come back to descriptor sets later.

#### Vertex Buffers

The fundamental problem of graphics is how to manage large sets of data. A vertex buffer is an array of rows of relevant vertex information, such as its position, normal, color, etc. Unlike OpenGL where it would handle allocation and handling memory for you, in Vulkan, you must:

1. allocate all the memory related to your buffer.
2. Map that data to a host visible handle.
3. Copy that data to your GPU.
4. Bind your buffer to that block of memory.

#### Descriptor Sets

Descriptor Sets store the resources bound to the minding points in a shader. It connects the binding points of a shader with the buffers and images used for those bindings.

In React Fiber there's the idea of a frequently updated view and a not frequently updated view. Unreal Engine 4 shares this with two global uniform families for frequently (called variable parameters) and not frequently (constant parameters) updated uniforms. Descriptor Sets are where you would make this distinction in Vulkan.

// Binding 0: Uniform buffer (Vertex shader)  
std::vector<vk::DescriptorSetLayoutBinding> descriptorSetLayoutBindings =  
{  
 vk::DescriptorSetLayoutBinding(  
 0,  
 vk::DescriptorType::eUniformBuffer,  
 1,  
 vk::ShaderStageFlagBits::eVertex,  
 nullptr  
 )  
};  
  
std::vector<vk::DescriptorSetLayout> descriptorSetLayouts = {  
 device.createDescriptorSetLayout(  
 vk::DescriptorSetLayoutCreateInfo(  
 vk::DescriptorSetLayoutCreateFlags(),  
 descriptorSetLayoutBindings.size(),  
 descriptorSetLayoutBindings.data()  
 )  
 )  
};  
  
auto descriptorSets = device.allocateDescriptorSets(  
 vk::DescriptorSetAllocateInfo(  
 descriptorPool,  
 descriptorSetLayouts.size(),  
 descriptorSetLayouts.data()  
 )  
);

#### Pipeline Layouts

Pipeline layouts are a collection of descriptor sets, the bindings to a shader program. In OpenGL in order to bind a shader to a set of data, you needed to describe how the inputs and outputs are organized in memory (their spacing, size, etc.)

Access to descriptor sets from a pipeline is accomplished through a pipeline layout. Zero or more descriptor set layouts and zero or more push constant ranges are combined to form a pipeline layout object which describes the complete set of resources that can be accessed by a pipeline. The pipeline layout represents a sequence of descriptor sets with each having a specific layout. This sequence of layouts is used to determine the interface between shader stages and shader resources. Each pipeline is created using a pipeline layout.

#### Pipeline State Objects

The rendering pipeline on GPUs is a large complex system that involves a lot of states, a Pipeline encapsulates the entire process.

As much as GPUs are now programmable, they still have some static state that you as a developer need to manage when performing draw calls. These include:

#### Graphics Pipeline

* Color Blending - The function that controls how two objects draw on top of each other.
* Depth Stencil - A extra piece of information that describes depth information.
* Vertex Input - The actual vertex data you'll be using in your shader.
* Shaders - What shaders will be loaded in.

And many more. These can even be cached! These particular draw calls are grouped such that in older graphics APIs, they would trigger shader recompilation.

#### Pipeline Cache

A pipeline cache serves to cache previously created pipelines for reuse later. Since pipelines don't change often, this you can quickly create another for use later.

auto pipelineCache = device.createPipelineCache(vk::PipelineCacheCreateInfo());

#### Dynamic State Objects

Any fast changes of state will happen in the dynamic state objects.

#### Shaders

Shaders must be passed to Vulkan as SPIR-V binary, so any compiler that can make SPIR-V is allowed. Shaders are precompiled, loaded into memory, transfered to a shader module, bundled in a set of pipelineShaderStages, which is then put into a graphics pipeline.

Shaders are compiled using the glslangvalidator bundled with the Vulkan SDK provided by LunarG.

glslangvalidator -V shader.vert -o shader.vert.spv  
glslangvalidator -V shader.frag -o shader.frag.spv

Call to Action: Now that we're sending binary as shaders, we could theoretically compose functions, use metaprogramming, and/or serialize classes to create SPIR-V. This would be a great research topic!

Vulkan's GLSL code is the same as OpenGL 4.5:

// Vertex Shader  
#version 450  
  
#extension GL\_ARB\_separate\_shader\_objects : enable  
#extension GL\_ARB\_shading\_language\_420pack : enable  
  
layout (location = 0) in vec3 inPos;  
layout (location = 1) in vec3 inColor;  
  
layout (binding = 0) uniform UBO  
{  
 mat4 projectionMatrix;  
 mat4 modelMatrix;  
 mat4 viewMatrix;  
} ubo;  
  
layout (location = 0) out vec3 outColor;  
  
out gl\_PerVertex  
{  
 vec4 gl\_Position;  
};  
  
  
void main()  
{  
 outColor = inColor;  
 gl\_Position = ubo.projectionMatrix \* ubo.viewMatrix \* ubo.modelMatrix \* vec4(inPos.xyz, 1.0);  
}

// Fragment Shader  
#version 450  
  
#extension GL\_ARB\_separate\_shader\_objects : enable  
#extension GL\_ARB\_shading\_language\_420pack : enable  
  
layout (location = 0) in vec3 inColor;  
  
layout (location = 0) out vec4 outFragColor;  
  
void main()  
{  
 outFragColor = vec4(inColor, 1.0);  
}

Shaders are loaded into Pipeline Layouts which are then executed by a command buffer.

auto vertModule = device.createShaderModule(  
 vk::ShaderModuleCreateInfo(  
 vk::ShaderModuleCreateFlags(),  
 vertexShader.size(),  
 vertexShader.data()  
 )  
);  
  
auto fragModule = device.createShaderModule(  
 vk::ShaderModuleCreateInfo(  
 vk::ShaderModuleCreateFlags(),  
 fragShader.size(),  
 fragShader.data()  
 )  
);

#### Command Buffer

A command buffer is a container of GPU commands, this is where you would see commands similar to OpenGL's state commands:

* setViewport
* setSissor
* blitImage
* bindPipeline

A common pattern for building a command buffer is:

1. Start Render Pass
2. Bind Resources
   1. Descriptor Sets
   2. Vertex and Index Buffers
   3. Pipeline State
3. Modify Dynamic State
4. Draw
5. Repeat 2 Through 4 as Needed
6. End Render Pass

Different command buffer pools allow muti cpu command buffer recording, thus you could allocate a thread for each core on the CPU, and split rendering tasks across each core. This could be used to distribute rendering individual objects, differed rendering passes, physics calculations with compute buffers, etc.

auto renderArea = vk::Rect2D(vk::Offset2D(), vk::Extent2D(width, height));  
std::vector<vk::ClearValue> clearValues =  
{  
 vk::ClearColorValue(  
 std::array<float,4>{0.0f, 0.0f, 0.2f, 1.0f}  
 )  
};  
  
// From here we can do common GL commands  
// Lets add commands to each command buffer.  
for (int32\_t i = 0; i < commandBuffers.size(); ++i)  
{  
 commandBuffers[i].begin(vk::CommandBufferBeginInfo());  
 commandBuffers[i].beginRenderPass(  
 vk::RenderPassBeginInfo(  
 renderpass,  
 swapchainBuffers[i].frameBuffer,  
 renderArea,  
 clearValues.size(),  
 clearValues.data()  
 ),  
 vk::SubpassContents::eInline  
 );  
  
  
 std::vector<vk::Viewport> viewports =  
 {  
 vk::Viewport(0, 0, width, height, 0, 1.0f)  
 };  
  
 commandBuffers[i].setViewport(0, viewports);  
  
 std::vector<vk::Rect2D> scissors =  
 {  
 renderArea  
 };  
  
 commandBuffers[i].setScissor(0, scissors);  
  
 // Bind Descriptor Sets, these are attribute/uniform "descriptions"  
 commandBuffers[i].bindPipeline(vk::PipelineBindPoint::eGraphics, graphicsPipeline);  
  
 commandBuffers[i].bindDescriptorSets(  
 vk::PipelineBindPoint::eGraphics,  
 pipelineLayout,  
 0,  
 descriptorSets,  
 nullptr  
 );  
  
 commandBuffers[i].bindVertexBuffers(0, vertexBuffers, offsets);  
 commandBuffers[i].bindIndexBuffer(indices.buffer, 0, vk::IndexType::eUint32);  
 commandBuffers[i].drawIndexed(indices.count, 1, 0, 0, 1);  
 commandBuffers[i].endRenderPass();  
 commandBuffers[i].end();  
}

## Deployment

Currently each product deploys with a Visual Studio Solution compilation step. Either the repo includes a Visual Studio solution, or it generates it using Conan.

By default, Conan is already configured to compile to visual studio, refer to the installation guidelines section for more details.

Unity uses its own generator plugin you’ll have to install if you want to use Visual Studio, but is not necessary as unity includes its official editor MonoDevelop is for making changes.

## Design Patterns

**Actor Model** - Components are independent actors in an application, and all aspects of the interactive paint project have taken advantage of this pattern.

**Object Composition** - In an attempt to avoid complex classical hierarchies while programming in a more functional ML manner, we’ve opted to use Object Composition as a mechanism for reuse.   
  
In Unity, GameObject prefabs serve as independent entities providing interactive logic to the application. They are composed of component primitives such as mesh renderers, transformation matrixes, and controller scripts to further add functionality.

In Obsidian, the Actor Model is taken a step further by taking advantage of what Brad Frost coined as **Atomic Design**, a means of composing views as individual nested actors that are built from *atoms*, *molecules*, and *pages*.

Atomic design is a derivative of Component driven design, which Obsidian tries to extend outside of the boundaries of the web and JavaScript language into C++.

# System Validation

This section shows the unit tests and integration tests that were done throughout the course of developing Interactive Paint.

**Unit Tests**

**Sunny Day Tests**

LeapMotion update method

Left and Right finger locations were forced to x = 0.5 y = 0.5

Input: screen height = 720, width = 1280

Expected Result: left and right finger locations should become x = 640 y = 360

Actual Result: left and right finger locations were x = 640 y = 360

RealSense update method

Cursor location was forced to x = 0.5 y = 0.5

Input: screen height = 720, width = 1280

Expected Result: cursor location should become x = 640 y = 360

Actual Result: cursor location became x = 640 y = 360

DrawQueue execute method

DrawQueue vector filled with {(20,20), (30,30), (40,40), (50,50)}

Expected Result: On the screen there should be white dots at positions 20, 20; 30, 30; 40, 40; and 50, 50.

Actual Result: On the screen there were white dots at positions 20, 20; 30, 30; 40, 40; and 50, 50.

LeapMotion isCircleGestureClockWise method

Input: Clockwise circle gesture

Expected Result: True

Actual Result: True

Test case: Draw circle

Purpose: Test to see if a circle is drawn properly

Preconditions: Color is set to a value c

Action: with circle tool active, press an initial position, O, and drag out a distance, r, from the center and release mouse

Expected result: Value of pixel at point r away from O should be the preset color c

Test case: Draw square

Purpose: Test to see if a square is drawn properly

Preconditions: Color is set to a value c

Action: with circle tool active, press an initial position, O, and drag out a distance, r, from the center and release mouse

Expected result: Value of pixel at point r away from O should be the preset color c

Test case: Pick up object

Purpose: Test to see if object framework can pick up object and track the controller

Preconditions: user has no other objects picked up

Action: User moves controller over the object’s model, presses the pickup button, then moves the controller

Expected result: Object should start tracking the controller’s movements: velocity, rotation, trajectory when the button is released

Actual result: same as expected

Test case: Change scene

Purpose: Test to see if user can swap between scenes

Preconditions: User has no object picked up

Action: user presses the teleport button

Expected result: User should load the other unity scene (there are only two at this point)

Actual Result: same as expected

Test case: Move user

Purpose: Test to see if user can move to intended locations

Preconditions: User has no object picked up; user has object picked up

Action: user presses the teleport button, points to a location, then releases it if the color is green

Expected result: User should move to the pointed to location if the color of the beam is green, does not if the color is red

Actual result - Same as expected

Test case: Pick up object

Purpose: Test to see if object picks up when touched

Preconditions: User has no object in hand

Action: User moves controller over the object, then uses the pickup button

Expected result: Object should be highlighted when the models intersect, then the object should snap into place with the controller and track the movement

Actual result - same as expected

Test case: Use object

Purpose: Test to see if usable objects work properly

Preconditions: User has usable object picked up

Action: With usable object in hand, the user presses the use button.

Expected result: Usable object (gun) should shoot a projectile

Actual result - same as expected

Test case: Win Condition

Purpose: Test to see if win animation plays when key triggers the collider

Preconditions: Win condition has no other objects touching it

Action: User places the key item on the win condition box

Expected result: Yellow light should activate in an animation that increases its intensity

Actual Result: same as expected

Test case: Key gates

Purpose: Assure that gates can only fit the associated key item; user gates can only let user pass

Preconditions: User has the key item in hand

Action: user presses the teleport button, points to a location, then releases it if the color is green; User attempts to fit the key item into the hole

Expected result: User should teleport to the location. If passing through a transparent purple slot, the key item should be dropped; Key item should fit only into the associated shaped hole

Actual result: same as expected

**Rainy Day Tests**

LeapMotion isCircleGestureClockWise method

Input: Swipe gesture

Expected Result: False

Actual Result: False

**Integration Tests**

The painting application built by Jose and Justin was tested manually by allowing multiple users to use the application for several minutes at a time. Each trying out all the different features that the application has.

As most of the work done was split into individualized groups, there is little room for integration, or integration tests. At the request of our product owner, the concept of integration was to be reserved for future iterations of the project.

# 

# 

# 

# Glossary

**NanoGUI:** A C++ library that we used for making the GUI

**Cinder:** An OpenGL abstraction/framework that we used instead of using raw OpenGL to make writing code more efficient

**LeapMotion:** A motion sensor that can track hand movement as well as hand gestures.

**Intel RealSense:** A depth camera that can track gestures and hands.

**Microsoft Kinect:** A camera used to track body movement.

**HTC Vive:** A head-mounted display (HMD) that emulates a full 3D view and can track user motions. It has two hand-held controllers with a button layout, which are also tracked in 3 dimensions.

**Unity Engine:** A 3D physics engine aimed towards making games. It has several plugins and support options for common Virtual Reality (VR) hardware.

**Vulkan:** a new low level API released February 2016 by the Khronos Group that maps directly to the design of modern GPUs.

**OpenGL:** OpenGL stands for Open Graphics Library. It is a specification of an API for rendering graphics, usually in 3D.

# Appendix

## Appendix A - UML Diagrams



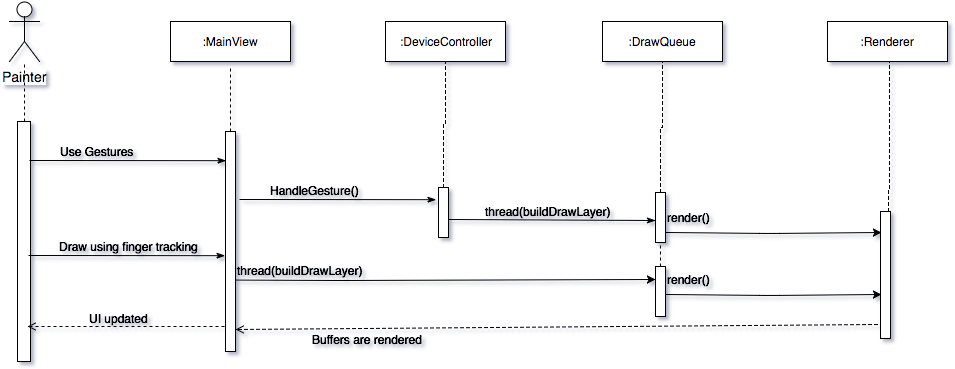
usecase.png

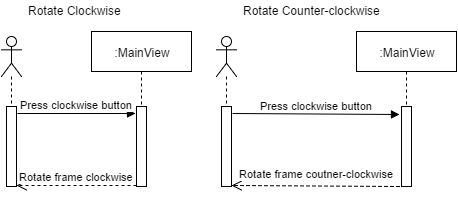
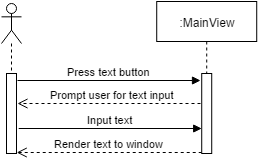
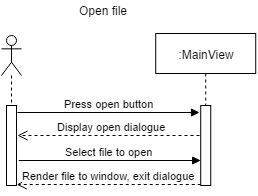
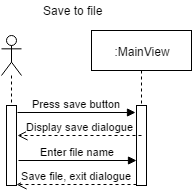
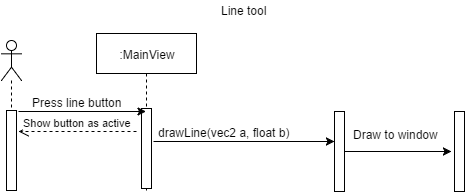
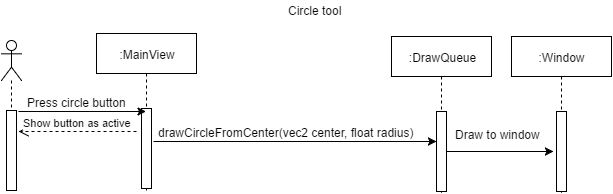
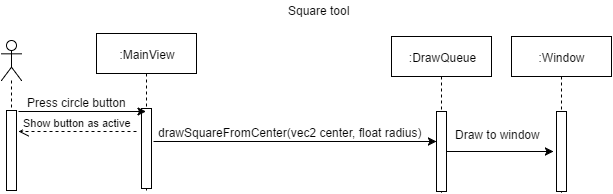
Sprint3UseCase.png

Usecase4.png

Untitled Diagram (4).png

Untitled Diagram (6).png





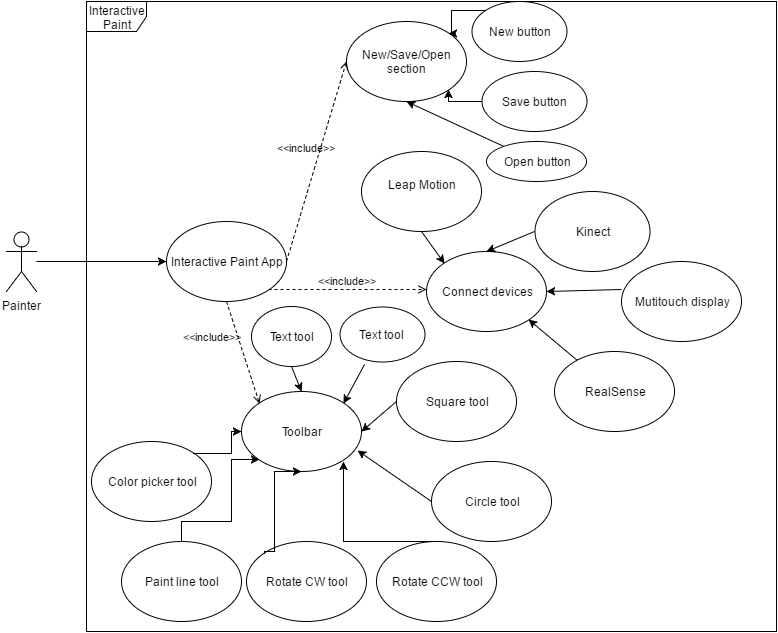
Untitled Diagram (2).png

Sprint3SequenceDiagram.png

Untitled Diagram (8).png

sequence4.png

Untitled Diagram (5).png

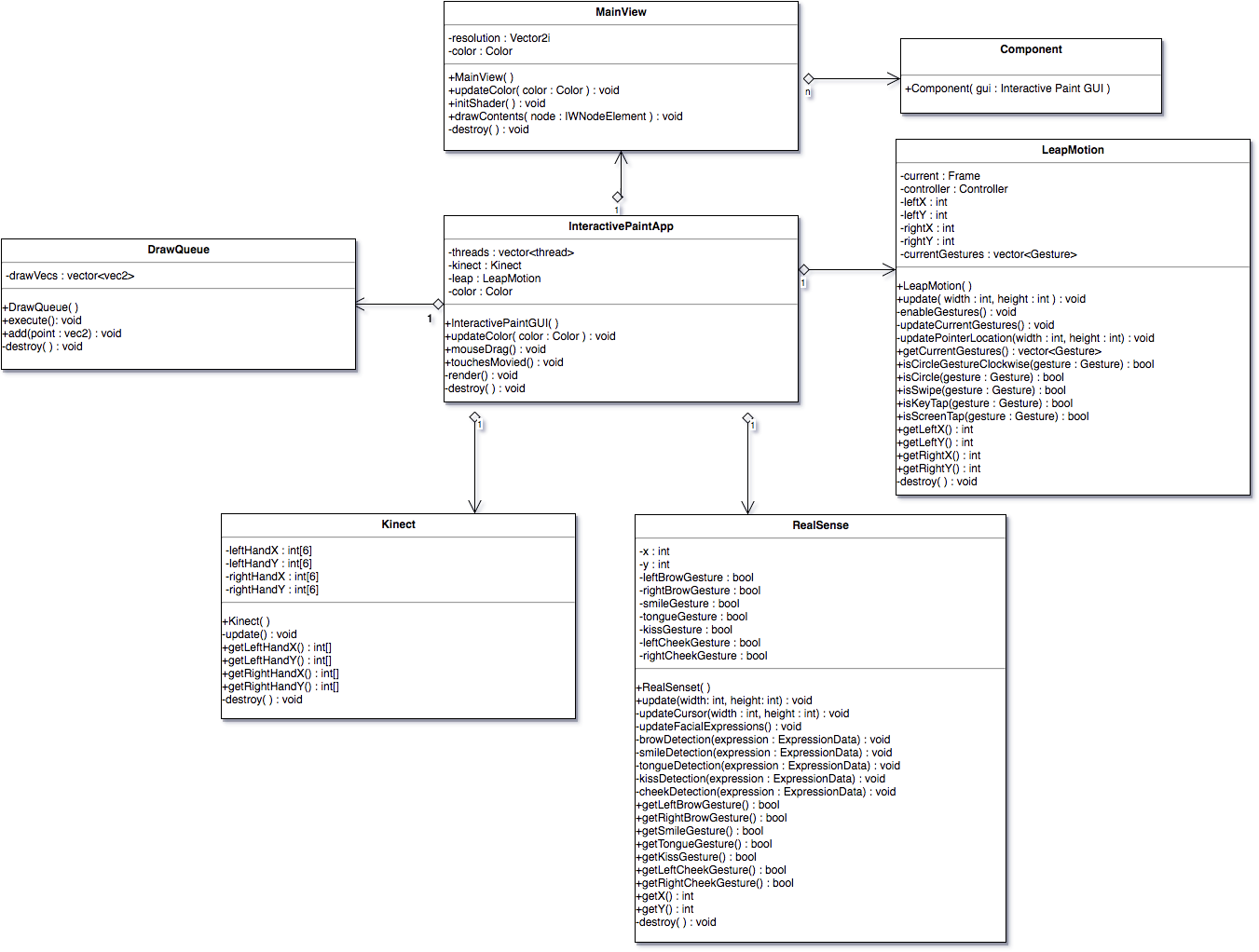


Sprint3ClassDiagram.png

Untitled Diagram (3).png

class4.png

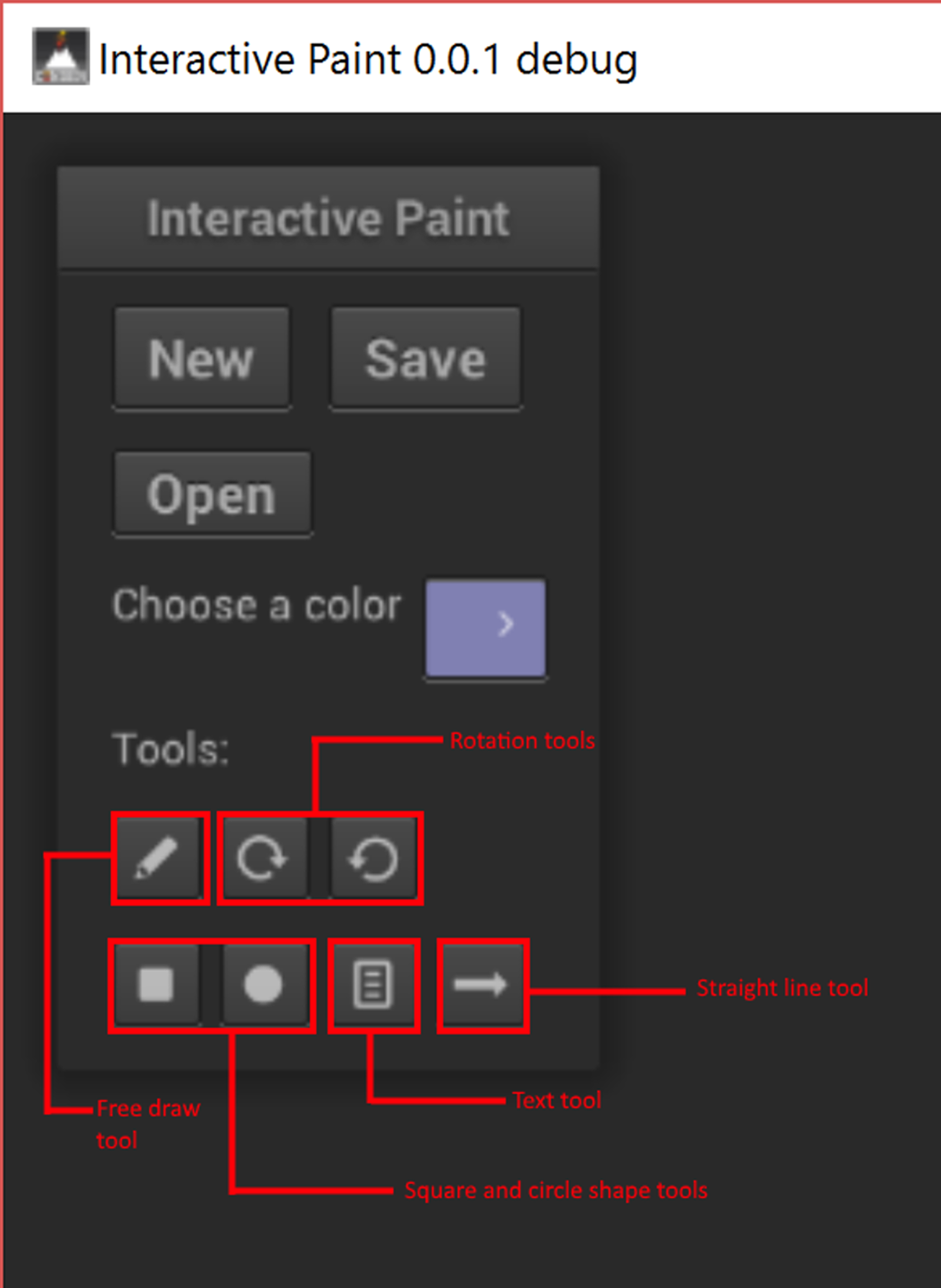
Untitled Diagram (7).png

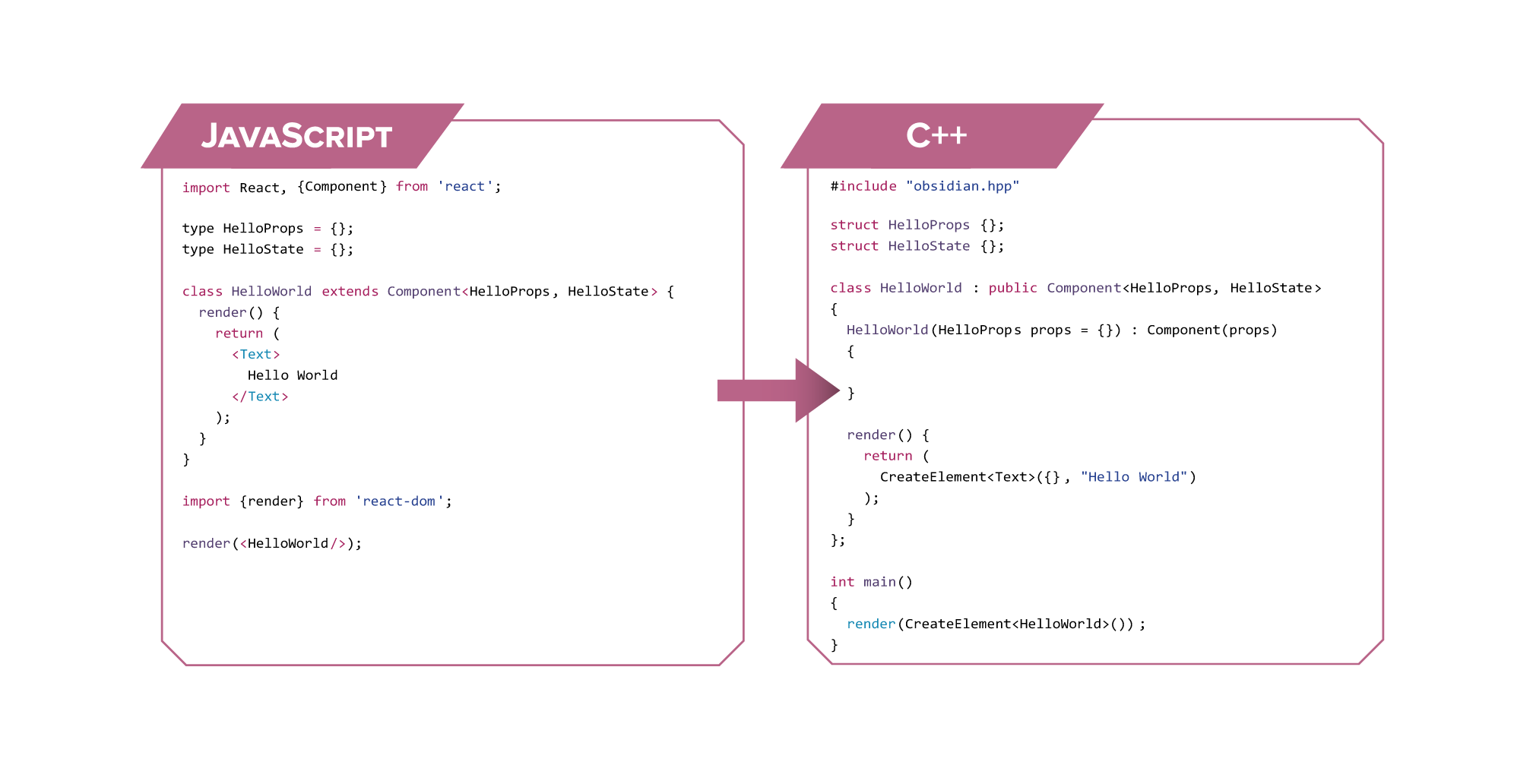
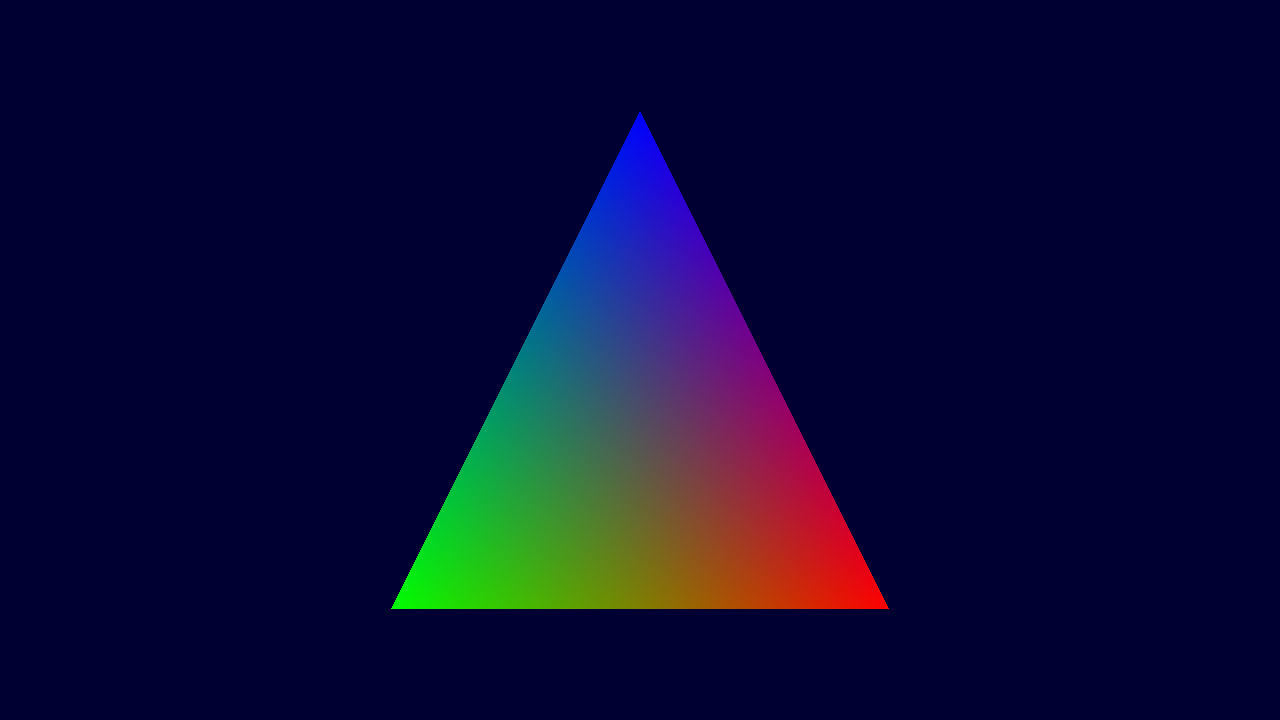
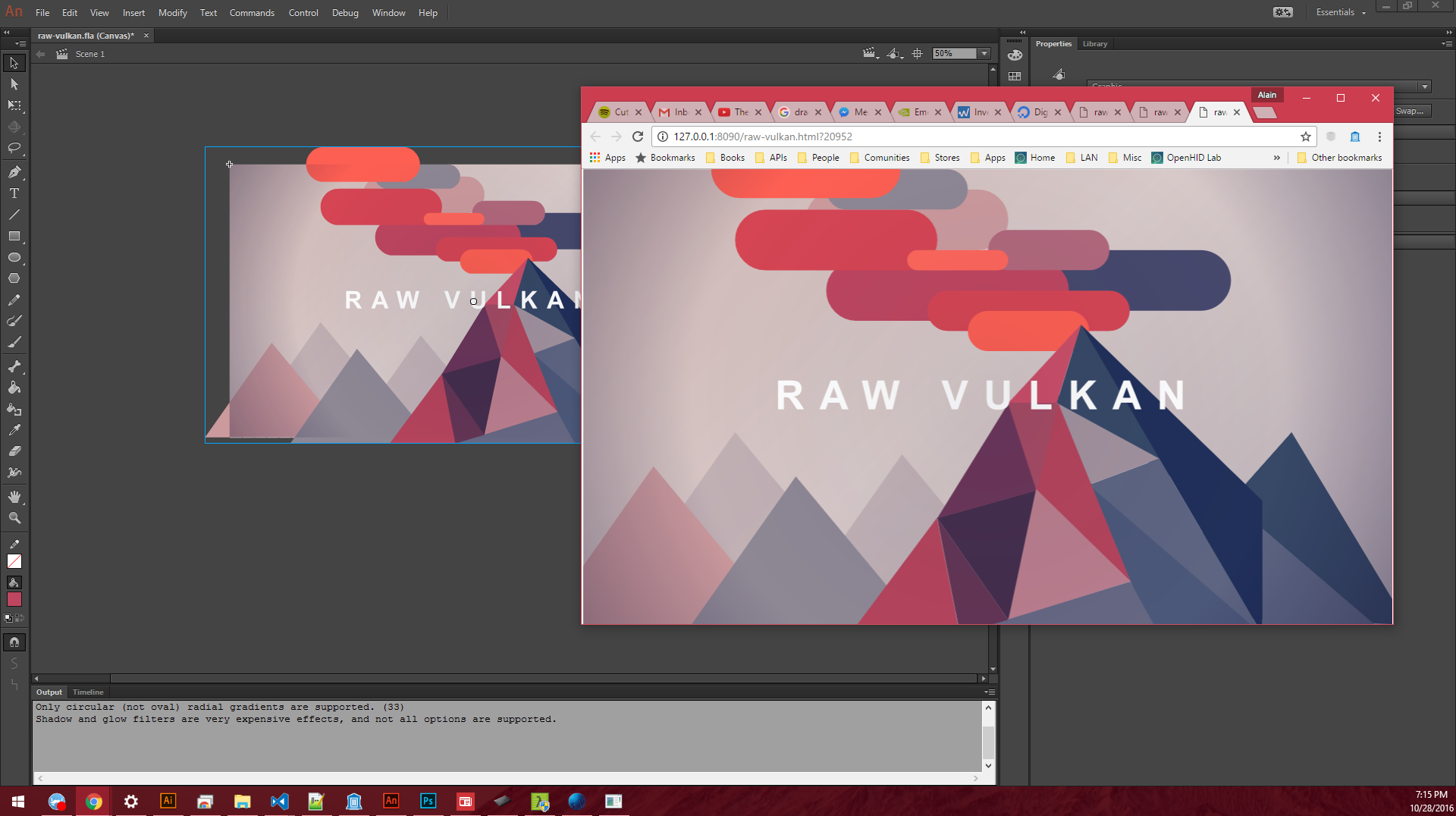
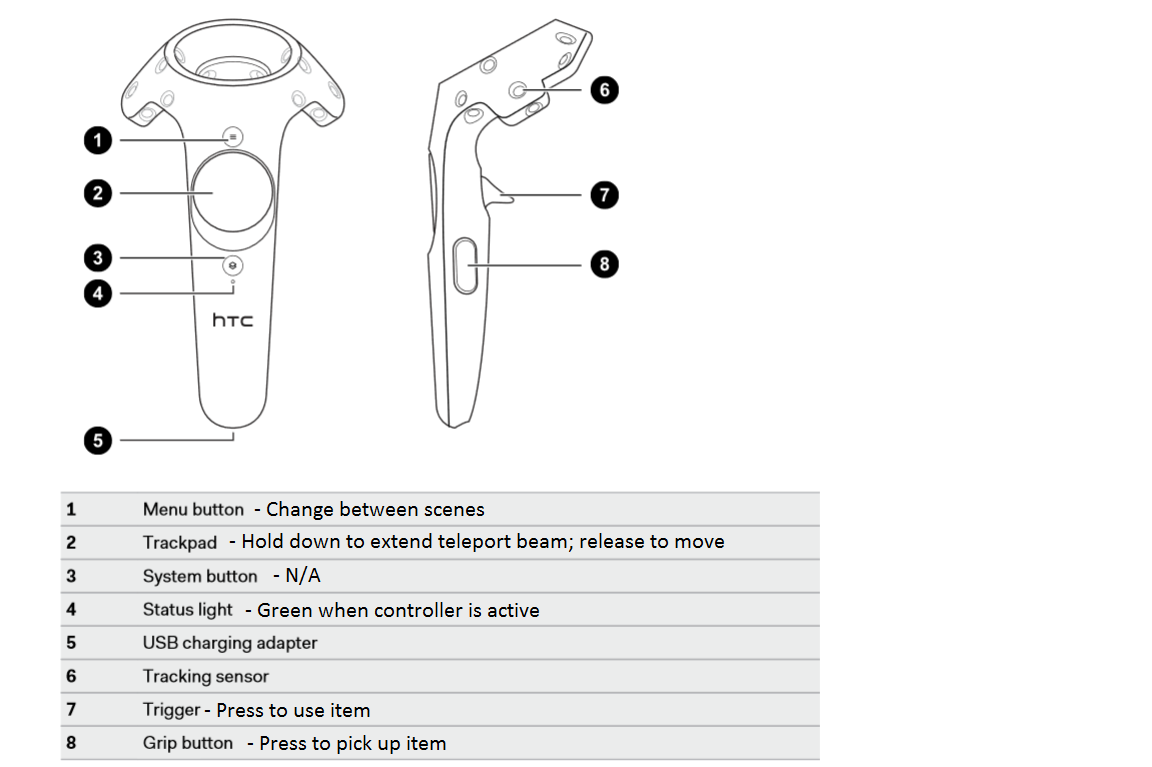


Class5.png



## Appendix B - User Interface Design





## Appendix C - Sprint Review Reports

### Sprint 1

Attendees: Jacob Leschen, Justin Alvarez, Alain Galvan, Jose Morgan

Start time: 4:30 PM

End time: 5:00 PM

What went wrong?

* Did we do a good job estimating our team's velocity?
  + **No**, most of us ended with an incorrect number of hours that had to be corrected over the weekend.
* Did we do a good job estimating the points (time required) for each user story?
  + **Yes**, but a bit more generous, it’s hard to be exact.
* Did each team member work as scheduled?
  + **Yes**.

What went right?

* Seeing as how this was more of half a sprint (since the first week was spent organizing), there wasn’t a lot that could have gone wrong, **Nothing of note.**

How to address the issues in the next sprint?

* How to improve the process?
  + Be more exact on sprint meeting times.
* How to improve the product?
  + Study more of the APIs we’re relying on.

### Sprint 2

Attendees: Jacob Leschen, Justin Alvarez, Jose Morgan, Alain Galvan

Start time: 2:00 PM

End time: 2:45 PM

What went wrong?

* Did we do a good job estimating our team's velocity?
  + Yes, each member predicted an accurate amount of work to do for the 2 week period
* Did we do a good job estimating the points (time required) for each user story?
  + Mostly yes, some user stories were done quicker than expected, others are not completed yet, we will use this knowledge for our sprint 3 planning
* Did each team member work as scheduled?
  + Mostly yes, some members worked overtime, but everyone put in at least 40 hours of work during the sprint

What went right?

* We were very aware of our limitations, and worked only on items that could be immediately tackled with our current hardware, knowledge, and progress with the project.
* We were accurate in projecting the amount of work to be done

How to address the issues in the next sprint?

* How to improve the process?
  + Try to create user stories with dedicated tasks, which can more accurately break down how many hours a story will take
* How to improve the product?
  + Use the skills we learned during the sprint, and obtain the hardware we need to proceed with important goals of the project.

### Sprint 3

Attendees: Jacob Leschen, Justin Alvarez, Jose Morgan, Alain Galvan

Start time: 2:00 PM

End time: 2:45 PM

What went wrong?

* Did we do a good job estimating our team's velocity?
  + Yes, each member predicted an accurate amount of work to do for the 2 week period
* Did we do a good job estimating the points (time required) for each user story?
  + Mostly yes, some user stories were done quicker than expected, others are not completed yet, we will use this knowledge for our sprint 3 planning
* Did each team member work as scheduled?
  + Mostly yes, some members worked overtime, but everyone put in at least 40 hours of work during the sprint

What went right?

* We were very aware of our limitations, and worked only on items that could be immediately tackled with our current hardware, knowledge, and progress with the project.
* We were accurate in projecting the amount of work to be done

How to address the issues in the next sprint?

* How to improve the process?
  + Try to create user stories with dedicated tasks, which can more accurately break down how many hours a story will take
* How to improve the product?
  + Use the skills we learned during the sprint, and obtain the hardware we need to proceed with important goals of the project.

### Sprint 4

Attendees: Jacob Leschen, Justin Alvarez, Jose Morgan, Alain Galvan

Start time: 3:00 PM

End time: 3:15 PM

What went wrong?

* Did we do a good job estimating our team's velocity?
  + No, so far we’re actually slowing down.
* Did we do a good job estimating the points (time required) for each user story?
  + Yes.
* Did each team member work as scheduled?
  + Yes.

What went right?

* We each contributed to our work individually.

How to address the issues in the next sprint?

* How to improve the process?
  + We need to work in a more unified manner.
* How to improve the product?
  + Really design the MVP.

### Sprint 5

Attendees: Jacob Leschen, Justin Alvarez, Jose Morgan, Alain Galvan

Start time: 3:00 PM

End time: 3:15 PM

What went wrong?

* Did we do a good job estimating our team's velocity?
  + Some slowdowns are occurring due to losing sight on the “big picture”. Lack of ideas for future features.
* Did we do a good job estimating the points (time required) for each user story?
  + Yes.
* Did each team member work as scheduled?
  + Yes.

What went right?

* We each contributed to our work individually.
* We are mostly prepared to begin our final demos

How to address the issues in the next sprint?

* How to improve the process?
  + We need to get more feedback from the owner to get better guidance on the next sprint.
* How to improve the product?
  + Really design the MVP.

### Sprint 6

Attendees: Jacob Leschen, Alain Galvan, Jose Morgan, Justin Alvarez

Start time: 3:00 pm

End time: 3:30 pm

What went wrong?

* Did we do a good job estimating our team's velocity?
  + Mostly, there were a few stories that needed to leak over into our half sprint 7, but this was anticipated from the beginning, so this was okay
* Did we do a good job estimating the points (time required) for each user story?
  + Yes, each team member gave an accurate estimation for the time requirements
* Did each team member work as scheduled?
  + Yes

What went right?

* Everything that we have planned for our final deliverable and presentation is ready to go

How to address the issues in the next sprint?

* (This is the final sprint) N/A

## Appendix D - User Manuals, Installation/Maintenance Document, Shortcomings/Wishlist Document and other documents

One of the main shortcomings/major design changes is the shift to more individualized work halfway towards the semester. The project split into 3 individualized teams - Vulkan API, Interactive Paint, and VR Framework. Because of this, much of this team’s work is more inclined towards future iterations, but there is little to no integration of the projects. Each team had their own goals to work towards, which is why it may seem like many of the results of our work doesn’t add up together. This was all at the instruction of our mentor, Francisco Ortega, who wished to have this more foundational work ethic. Hopefully, future iterations of the project will be able to integrate our work more fluidly than we have.

### Interactive Paint Installation

To install interactive paint you first need to install the SDKs for the Kinect, RealSense and Leap Motion, then install Cinder. See references for links to those items. Once those things are installed clone the project from github and open the solution in Visual Studio. Due to SDKs for the Kinect and RealSense being Windows only this project will only run on Windows 7, 8 and 10. The Kinect will only work on Windows 10 as of now but the application will still run it just won’t recognize input from the Kinect.

### Obsidian Installation

To install obsidian one simply needs to have the Conan C++ package manager installed, and enter the following in the command line.

conan install obsidian/1.0.0@alaingalvan/stable

**Example:**

Here's a simple counter example, where you press the button to adjust the counter's value.

#include "obsidian.hpp"  
  
using namespace obs;  
  
struct CounterProps  
{  
 CounterProps(uint32\_t color = 0xffffffff) : color(color);  
 uint32\_t color;  
}  
  
struct CounterState  
{  
 CounterState(uint32\_t count = 0) : count(count);  
 uint32\_t count;  
}  
  
class Counter : Component<CounterProps, CounterState>  
{  
 render()  
 {  
 return (  
 View({},  
 Text({props.color}, "Hello World!"),  
 Button({[&](auto e) { state.count += 1;}, "Click Here"}})  
 );  
 )  
 }  
};  
  
int main()  
{  
 // An easy to use  
 auto node =  
 Provider({},  
 Window({},  
 Counter({0xffae21})  
 )  
 );  
  
 obs::render(node);  
}

**Rendering with Vulkan**

Obsidian comes with standard primitives for rendering views, as well as primitives for manipulating a 3D space.

|  |  |
| --- | --- |
| **UI Components** | **Description** |
| Text | A component that displays text. |
| View | A wrapper component that can be used to compose other components. Implements a subset of CSS. |
| Image | A viewable image built from a a bitmap. |

|  |  |
| --- | --- |
| **OS Components** | **Description** |
| Provider | A Vulkan provider that provides the instance and devices to components within it. |
| Window | An abstraction for all windows the library supports. |

|  |  |
| --- | --- |
| **3D Components** | **Description** |
| Material | A description of shading data. |
| Mesh | A description of vertex data. |
| Surface | A renderable component that can be used to render to a window, post-processing pipeline, or texture. |
| Scene | A collection of actors, manages multi-threaded rendering and updates of actors. |
| Camera | A renderable surface target |

|  |  |
| --- | --- |
| **Primitives** | **Description** |
| Plane | A subdivisible plane. |
| Cube | A rectangular prism. |
| UV Sphere | An adjustable UV Sphere. |

# 

# 

# 

# 

# References

1. C++ reference <http://en.cppreference.com/w/>
2. Libcinder <https://libcinder.org/docs/>
3. Microsoft Kinect <https://msdn.microsoft.com/en-us/library/hh855354.aspx>
4. Leap Motion <https://developer.leapmotion.com/documentation/cpp/index.html>
5. Intel RealSense <https://software.intel.com/sites/landingpage/realsense/camera-sdk/v1.1/documentation/html/index.html?doc_devguide_introduction.html>
6. NanoGUI  
   <https://github.com/wjakob/nanogui>
7. HTC Vive  
   <http://www.vive.com/us/>
8. SteamVR  
   <http://store.steampowered.com/steamvr>
9. SHIRAEF, J. 2016. An Exploratory Study of High Performance Graphics Application Programming Interfaces. University of Tennessee.
10. FISHER, B. 2015. Flux: A Unidirectional Data Flow Architecture for React Apps. Applicative 2015.