Final Project Report

Jonathan Sumner Evans*, Robinson Merillat†, and Sam Sartor‡
Department of Computer Science, Colorado School of Mines
Golden, Colorado
Email: *jonathanevans@mines.edu, †rdmerillat@mines.edu, ‡ssartor@mines.edu

***Abstract*—Virtual Reality (VR) Technology has been developing rapidly over the past decade. Current solutions such as Unity attempt to use old programming languages and even older paradigms to implement VR applications and thus, limit developers' abilities to create new and unique environments. With every cutting edge technology, new paradigms and design patterns must be invented. In this paper, we discuss a deferred immediate mode (DIM) application architecture suitable for implementing large virtual reality applications. We present a UI library which utilizes this architecture and a few case studies of this library in use.**

I. Introduction

The Virtual Reality (VR) market is growing rapidly. The International Data Corporation (IDC) projects that revenues for the combined Augmented Reality (AR) and Virtual Reality markets will grow from $5.2 billion in 2016 to more than $162 billion in 2020 [4]. This flourishing new industry has created an exciting new field of Software Engineering with great potential for revolutionary new design paradigms and program architectures.

Most current frameworks and libraries attempt to apply old design paradigms and program architectures to virtual reality. While these paradigms and architectures are well-suited for 2D user interfaces and rendering 3D environments to a flat screen, they are not designed with VR in mind. Although endeavors to adapt these old patterns to VR have been successful, they are restrictive in both the code architecture and way of thinking. Grace Hopper loved to say "The most damaging phrase in the language is: 'It's always been done that way'" [3]. This quote promotes new ways of thinking. Utilizing outdated architectures hinders the exploration and the potential for new design paradigms and program architectures.

Our goal is to find a system that provides a modern, fast, and practical approach to virtual reality development. Specifically, we require that this system meets the following criteria.

A. Performant

Traditionally, animation is done around 60 frames per second. This is due to the wave of monitors and televisions that ran at 60Hz and had a refresh rate of 60 frames per second (fps). This is fine when looking at a screen from a distance, however it has been found that in order to appear believable and prevent disorientation, a virtual reality program must run with at least 90 fps. In fact many studies have shown that when VR is run with lower frame rates users experience headaches and nausea faster than when at high frame rates [8]. As VR headsets run two mini monitors concurrently, the effective required frame rate is 180fps. Achieving this frame rate is resource intensive and requires highly efficient and optimized code. Additionally, multi-threading is imperative so that long-running processes can occur without blocking the user interface. This is unlike a traditional desktop user interface where blocking the UI process for a second does not have a major affect on the usability of the program. These factors require a VR framework that is highly performant and multi-threadable.

B. Natural

A *natural user interface* is an interface that can be used without the need for a controller [10]. Although current VR systems utilize hand-held controllers, they emulate this goal much better than desktop, mobile, and web applications. For our discussion of these natural user interfaces, we define the following terms.

Definition 1. *A* planar UI *is a user interface where components are organized along a 2D surface.*

Definition 2. *A* spacial UI *is a user interface where components are organized within a 3D space.*

Both planar and spacial user interfaces are effective in virtual reality applications. This cannot be said of applications that use traditional human-interface devices such as mice and keyboards. Because virtual reality environments are inherently 3D, they make spacial UI convenient and practical for the first time. Thus, the ability to create spacial UI rather than merely to planar UI is a high priority.

C. Flexible

Computer software is used to solve a variety of unique problems which in turn require a variety of user interface solutions. Thus, a good user interface toolkit must be general enough to accommodate the goals of the developers writing the software. Most importantly, such a toolkit must not be opinionated about which types of application should be created with it. The desktop, mobile, and web application fields already have general purpose user interface toolkits (e.g. HTML and GTK). We need a UI library for VR which is equally general-purpose.

D. Modular

A direct result of flexibility is modularity—a flexible architecture cannot be a monolith. Many current VR libraries include features such as pathfinding and character rigging which must be included even if they are not used. This is not modular and most non-entertainment software does not require any of these features.

From a software engineering standpoint, a good UI library must allow the programmer to integrate any number of components, but these components should be add-ons, not dependencies. Additionally, these components should be compartmentalized and not interfere with one another. This library design promotes good software engineering practices including the UNIX philosophy ("do one thing and do it well") and the open/closed principle ("software entities should be open for extension but closed to modification").

We need a framework that has a minimal feature set baked in while still allowing extensibility via the addition of self-contained modules.

***

Given these criteria, we evaluate current program architectures and UI libraries in Section II. We describe the overarching architecture of these libraries is Section III and discuss the immediate mode architecture in Section IV. We describe how these architectures influenced our deferred immediate mode architecture which we formally present in Section V. In Section VI we describe Flight—an implementation of a VR UI library using the deferred immediate mode architecture. Then we present three case studies of Flight and the deferred immediate mode architecture in use in real applications in Section VII and conclude in Section VIII.

II. Existing Toolkits

We researched many current VR software architectures to find one which suited our needs. In this section we describe a variety of libraries and evaluate their ability to accomplish our goals as described in the previous section.

A. Unity

Unity is a game engine which was designed to allow programmers to easily create games and has many features which make this process effortless. Unity has been used to create many successful, award-winning VR games and applications including SUPERHOT [1] and TiltBrush [9]. The Unity ecosystem is growing rapidly and has become the de facto standard for building VR applications. For example, Google released a TiltBrush toolkit on GitHub under the Apache 2.0 library [2].

However, Unity's UI system was designed for building 2D UIs and has been adapted for making 3D UIs. Although these adaptations have been successful, we wanted to explore systems which have 3D UI elements as first class citizens.

B. A-Frame

A-Frame is a virtual reality engine created by Mozilla for the web. Scenes are built using declarative HTML, and evaluated as an entity component system (ECS). Unlike traditional object-oriented programming, where new object types generally inherit functionality from a single parent, an ECS creates objects through composition. This works very well in a declarative environment, since an entity can be declared into existence by simply

listing a set of components (e.g. color, shape, movement, interactivity). [7]. In addition, building VR apps using HTML has the distinct advantage of leveraging proven web UI frameworks such as Facebook's React library [6].

React and A-Frame are very effective solutions to the problem of VR on the web, however both are still fundamentally reliant on the document object model (DOM). This constrains the performance and generality of virtual reality applications. If the DOM is entirely abstracted away, then there is no reason to build on it in the first place. VR can be attacked at a much lower level using desktop VR frameworks and ported to the web using technologies such as webassembly (WASM).

III. Retained Mode

All of the libraries and frameworks we discussed in Section II use a *retained mode architecture*. Retained mode architectures are declarative. In the context of user interfaces, this means that the entire user interface is defined and stored in memory. For each each frame, the graphics library draws the entire scene from memory. The entire scene is stored in memory between frames and modification of the scene is done by modifying the in-memory representation of the scene [5].

Retained mode presents some problems with synchronization across user interface trees. For example, many applications have a stored state such as a list of items table. However, there may be elements external to the table which modify the list of items. This causes a problem of cross mutation. One common way of fixing this problem in old applications was to add a function call to refresh the table when the button was clicked. This has major scalability problems when the elements which are change by a single button form a massive dependency tree. Additionally, if a programmer misses one refresh call, it can cause the state in part of the application to become stale.

Currently, one of the popular ways of handling this issue is using Flux, the underlying architecture of Redux. This model forces state to be stored in a single global store. Data is sent from this store down the entire UI tree, and messages which modify the state are propagated upwards through the tree. This model ensures that there is one source of truth—the common data store—and that if one component sends a message which modifies that central state, then the entire UI tree can update accordingly.

IV. Immediate Mode

An alternative way of tackling this problem is using *immediate mode*. In this architecture, the UI is defined procedurally for each frame. The graphics library does not store the scene in memory, rather any state that is necessary to create the scene is stored by the application itself [5]. Immediate mode prevents the problem of stale state and cross mutation by recomputing the entire UI scene every frame. In an immediate mode architecture, a programmer defines that a UI element exists on each frame, and if they want to remove a UI element, they simply do not call the function to create an element.

The major drawback of immediate mode is that there is no guarantee that another element will not be created later that will interfere with the current element's state. We call this problem the incomplete information problem. An example of this problem would be an element in a virtual environment whose color depends on whether or not the user is pointing at it. It is easy to detect whether or not the controller is pointed at the element. It is impossible to guarantee that another component will no be added to the scene which will obstruct the controller's view of the first element.

In other words, there are some questions about the state of the system which cannot be answered until all UI elements have "reported" their state. To solve this incomplete information problem, we added a new aspect to immediate mode: deferability described in the next section.

V. Deferred Immediate Mode

The Deferred Immediate Mode (DIM) architecture provides all of the advantages of both immediate mode and retained mode. DIM starts with immediate mode as a base and thus inherits the easy procedural definition of UIs from immediate mode. Like immediate mode, DIM also does not have the problem of stale state and cross mutation. DIM solves the incomplete data problem from the immediate mode architecture by deferring the final

state resolution until all all elements have reported their state.

The core pattern of DIM is that UI elements are declared and rendered using the following process

1) Each element is defined and reports their state.
2) Deferred computations are resolved.
3) Each element receives the resolved state and finishes updating.

VI. Flight

Flight is our implementation of a VR UI library and the DIM architecture using the Rust programming language. Flight is designed from the ground up to be performant, general, and modular.

A. Language

We chose to implement Flight using the Rust language for a few reasons.

1) Performance: As mentioned in Section I-A, virtual reality requires a high frame rate. Rust is a very fast compiled language that makes asynchronous and concurrent code safe and easy to write.

2) Safety: Rust's safety guarantees can help to eliminate the time consuming debugging of memory, logic, and concurrency issues.

3) Ecosystem: Despite being very young, the Rust ecosystem already has rich, full-featured tools for graphics, physics, and virtual reality.

4) Functional: Concisely implementing DIM requires first-class functions and functional constructs. C++ does not have this, but Rust does.

B. Dependencies

The Rust dependency manager, Cargo, allows programmers to easily include third-party libraries from https://crates.io. To avoid duplicating work by other programmers, we utilized a few external libraries for Flight. The main dependencies are listed below.

- **`rust-webvr`**: VR hardware API wrapper
- **`nalgebra`**: linear primitives and operations
- **`ncollide`**: geometric operations and queries
- **`nphysics`**: rigid body physics engine
- **`gfx`**: type-safe OpenGL wrapper

C. API Pattern

Most parts of the user interface update process (rendering, point-tests, etc.) can be done online with a pure immediate mode API. Only a few important steps (ray-casting, physics step, etc.) implicitly require DIM. These operations are made available by objects we call "Gurus". Gurus are responsible for aggregating reports and then resolving to a complete output. For example, the API for a simple ray-cast guru might be the following:

```
guru RayCast:
    variable shapes: []
    variable ray: ...

    function ray_cast(shape):
        if ray.hits(shape):
            append shape to shapes
        yield until resolved
        return shapes[0] is shape

    function resolve():
        sort shapes by ray.distance(shape)
```

Any component that depends on a guru must also have a mechanism for deferring:

```
function dim_example(guru):
    data ← query(guru)
    yeild until data
    return data
```

A DIM function in the Rust language usually has this rough form:

```
fn dim_example<'partial>(
    // mutates the self state
    &'partial mut self,
    // depends on some deferred computation
    guru: &mut Guru
)
    // defer self until computation is resolved
    -> impl FnOnce(&GuruReply)
    // output will be available
    -> MyOutput + 'partial
{
    // ask the guru a question
    let data = self.query(guru);
    // defer further work on self
    move |reply| {
        // guru answer is available
        return data(reply)
    }
}
```

Notice that the `dim_example` function returns some type that is `FnOnce`. Specifically, the

last few lines of `dim_example` return a closure (`|reply| ...`) that is responsible for resolving the incomplete computation.

D. Dependencies

The Rust dependency manager, Cargo, allows programmers to easily include third-party libraries from https://crates.io. To avoid duplicating work by other programmers, we utilized a few external libraries for Flight. The main dependencies are listed below.

- **`rust-webvr`**: OpenVR wrapper used by the Servo browser
- **`nalgebra`**: linear primitives, transformations, and operations
- **`ncollide`**: geometric operations and queries
- **`nphysics`**: a simple rigid body physics engine
- **`gfx`**: a powerful, type-safe OpenGL, DX11, and Vulkan wrapper

VII. Case Studies

We started this project with the goal of implementing a graphical shell in virtual reality. The problems we encountered motivated the development of the DIM architecture and `flight-ui`.

A. Yanking, Grabbing, and Pointing

The most important aspect of a virtual reality application is the interface system. Since our interface incorporates many movable elements, we needed intuitive controls for moving objects. We chose to implement yanking (bringing a distant object closer), grabbing (directly manipulating an object at close range), and pointing (triggering an event from a distance). All three were done through a combination of the `InteractGuru` object and a `Moveable` state object.

DIM enabled these interactions to be perfectly encapsulated. For example, a huge last-minute refactor of the movement system required virtual no changes to the physics system, global user interface, and applications despite all relying heavily on the old API. Again, countless new edge cases were created by the addition of the yanking operation, but the physics guru handled them easily without any changes.

B. Let's Get Physical and Snowflakes — Physics

We created two applications which rely heavily on physics. *Let's Get Physical* allows the user to swing and throw Mjolnir, Thor's hammer, and hit other objects in a physically realistic manner. *Snowflakes* allows the user to stack snow blocks on top of one another in a physically accurate manner. We created these applications separately and then combined them later. We discuss this combined application here.

The challenge with physics in immediate mode is the fact that objects in the physical world are inherently intertwined. For example, each snow block might influence, or be influenced by, any other snow block or Mjolnir. Without deferred immediate mode, the physics world would need to be the ultimate source of truth regarding which snow blocks exist and where they are in space. The problem with this is that we need to allow the user to spawn and move blocks within the world while still interacting with all of the other physics objects.

With deferred immediate mode, we are able to add each block and Mjolnir to the physics world (using the physics guru) every frame, then we wait for the physics guru to resolve. After the physics guru has resolved, we are able to render the blocks and Mjolnir in the proper location.

Although these applications were developed independently, and even still exist in separate modules of our application, deferred immediate mode with the physics guru allowed us to integrate the two applications with minimal code changes. The end result was being able to hit snow blocks with Mjolnir and vise versa.

C. Global User Interface

For our project we also created a module which allowed the user to turn on and off *Let's Get Physical* and *Snowflakes*. This module rendered two boxes, each of which toggles on and off one of the applications. We stored some minimal state about whether or not each application is enabled. If the application is not enabled, its functions are not called and thus, since we are using immediate mode, the objects are not rendered. When the application is toggled back on, the functions for that application begin to be called again and the objects associated with that application are rendered again.

This example illustrates the power of immediate mode. We did not have to go through the entire scene removing or adding all of the elements associated with the toggled application, we merely had to check a boolean value to determine whether or not to call each applications' functions.

D. State Saving

We also implemented the ability to save the state of the application. Each module of the application individually defines a serialize and deserialize function which converts the module state to a JSON format. When the user exits and reopens the application, the state of the application is restored. By using immediate mode, we are able to minimize the amount of state that we have to store; we only need to store enough state to begin calling the same functions once the application is reopened.

In the example of *Snowflakes* we only need to store the location and rotation of each snow block. Likewise for *Let's Get Physical*; we only need to store the location and rotation of Mjolnir.

VIII. Conclusion

References

[1] Joe Durbin. *Superhot VR Wins Best Game At Unity Vision Summit 2017*. May 1, 2017. URL: https://uploadvr.com/superhot-vr-best-game/.

[2] googlevr. *Tilt Brush Toolkit*. URL: https://github.com/googlevr/tilt-brush-toolkit.

[3] Grace Murray Hopper. *Information Week Interview*. Mar. 9, 1987. URL: https://quoteinvestigator.com/2014/11/27/always-done/.

[4] IDC. *Worldwide Revenues for Augmented and Virtual Reality Forecast to Reach $162 Billion in 2020, According to IDC*. Aug. 15, 2016. URL: https://www.idc.com/getdoc.jsp?containerId=prUS41676216.

[5] Microsoft. *Retained Mode Versus Immediate Mode*. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/ff684178(v=vs.85).aspx.

[6] Kevin Ngo. *aframe-react*. Aug. 31, 2017. URL: https://github.com/ngokevin/aframe-react.

[7] Kevin Ngo. *Build the Virtual Reality Web with A-Frame Mozilla Hacks - the Web developer blog*. Mar. 3, 2016. URL: https://hacks.mozilla.org/2016/03/build-the-virtual-reality-web-with-a-frame/.

[8] *The Importance of Frame Rates*. URL: https://help.irisvr.com/hc/en-us/articles/215884547-The-Importance-of-Frame-Rates.

[9] Unity. *2015 Unity Awards Winners*. URL: https://unityweb.unity3d.com/awards/2015/winners.

[10] Alexandre Wimmers et al. *Natural User Interface and Virtual Reality Integration in Video Games*. Mar. 31, 2015. URL: http://csiflabs.cs.ucdavis.edu/~cs193/cs193_2015/23_design.pdf.