Creation and Manipulation of Swept Surfaces in a Virtual Environment

Ji-an Andrew Li
MEng Computer Science
Submission Date: 2nd May 2014
Supervisor: Prof. Anthony Steed

Abstract

The modelling of three dimensional objects is an extremely common process. However, the tools for performing this task have not seen much innovation over the years; the interface and interactions have largely remained the same, with a few minor refreshes and updates. These tools remain very unintuitive and unfriendly, discouraging inexperienced people from learning and requiring aspiring professionals to extensively train.

This project tries to address this by exploring the possibility of using 3D user interfaces and virtual reality to provide a simpler and more intuitive method to creating 3D objects. Swept surfaces form the basics of many modelled objects, so this project focusses on creating a tool for the creation and manipulation of swept surfaces.

A number of requirements and features were identified, partially based on issues plaguing traditional tools, and a tool was designed and implemented using Unity and MiddleVR, intended to be used within a CAVE (Cave Automatic Virtual Environment). A functional tool was produced, meeting most of the identified requirements.

An informal user study was carried out to evaluate the effectiveness of the tool, which was produced generally positive impressions. The user study also identified some flows that could be addressed in the future.

Acknowledgements

I would like to express my deep gratitude to Prof. Anthony Steed, for his constant guidance and critique throughout all stages of the project. Without his suggestions, this project would not have come together.

My thanks also goes to Dr. David Swapp, who assisted me on multiple occasions with issues ranging from getting the CAVE to work and troubleshooting my code, and Dr. William Steptoe, for his crash course to Unity, which was extremely useful.

Finally, I would like to thank my family for their support and encouragement throughout the duration of my studies.

Contents

List of Figures

Chapter 1

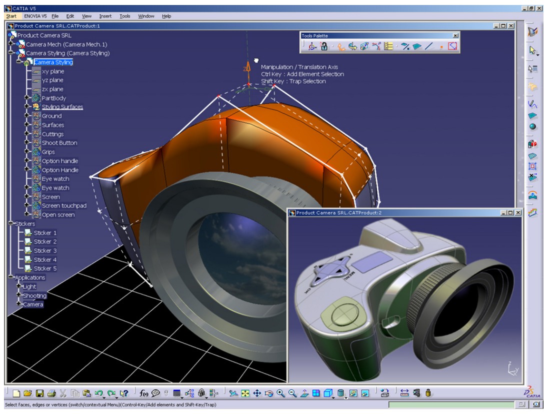
Introduction

1.1 Outline

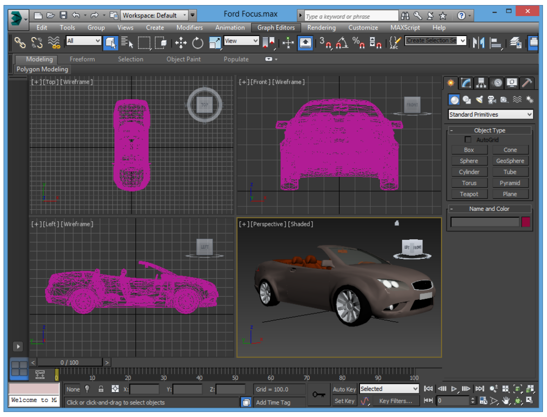
The process of creating three dimensional objects, also known as modelling, is a process that has matured over many decades. Its roots lie in early computer-aided design (CAD) software, and the overall design and concept of these pieces of software has not changed much (see fig. 1.1).

Despite modelling becoming a very common activity used in many different industries, the tools to perform it haven't evolved much. As can be seen, there are large similarities between CATIA V5, released in 1998, and Autodesk 3ds Max 2014, released in 2013. Most of the application screen is dominated by one or more visual projections of the modelled object, with additional tools accessible via toolbars.

However, the process of modelling an object is unintuitive and typically requires significant effort and training for users to learn, as most modelling tools are designed for use by industry professionals. Users are generally presented with a lot of tools to work with, which can be very overwhelming. They also don't allow users to properly visualise how the object looks in a three-dimensional space, meaning that trial and



(a) CATIA V5 (1998)
Source: Screenshot by Encyo [1]



(b) Autodesk 3ds Max 2014 (2013)
Source: Screenshot

Figure 1.1: Comparison of old and modern modelling software

error is usually required to produce desired shape.

Computer modelling can be likened to modelling in the real world, such as clay modelling, which is a skilled craft that requires training and experience to produce a quality product. However, unlike computer modelling, the act of shaping and modelling clay is a physical interaction and still intuitive to beginners. This brings to question whether the 'intuitivity' of computer modelling could be improved through a different means of interaction.

1.2 Aims and Goals

This goal of this project is to explore the idea of creating swept surfaces using a three-dimensional virtual environment, and ultimately to create a tool that allows users to intuitively create and manipulate swept surfaces in these virtual environments.

Additional aims of the project include learning how to use the Unity game engine to create a tool from scratch, and gaining a deeper understanding about interactions with three-dimensional objects in virtual reality.

1.3 Overview of Report

This report follows a standard structure for reports. Chapter 2 details the background of the project, including a review of all the technology involved or related to the project and the tools to be used. A set of requirements for the project are presented in chapter 3, as well as the design for the project. Chapter 4 describes the implementation process, detailing major milestones and obstacles. The results of the project are addressed in chapter 5, including details of an informal user study. Finally, chapter 6 concludes the report by reflecting upon and evaluating the work done and presenting potential future work to be done.

Chapter 2

Background

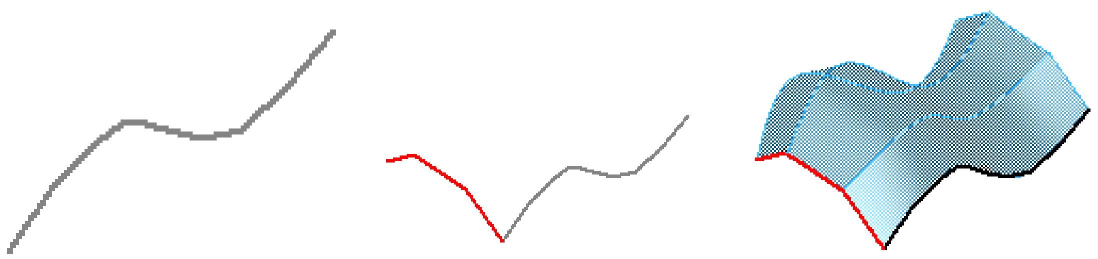
2.1 Project Context

There are a number of specialised technical pieces of terminology relevant to this project and are utilised through the report. This section provides a brief overview of them.

2.1.1 Swept Surfaces

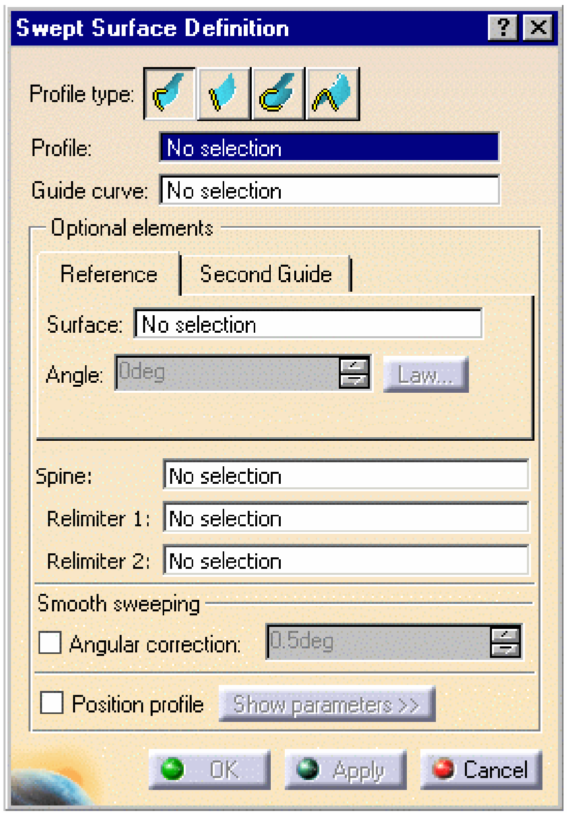
A swept surface typically refers to a type of three-dimensional surface created by extrusion or “sweeping”. To create such a surface, an initial *profile* curve is “swept” along a *path*, as illustrated in fig. 2.1.

A common workflow for creating a swept surface using an industrial computer-aided design tool, Autodesk Alias, usually involves first creating a profile curve, and then drawing the sweep path [3]. Once this is set, the surface is then generated. However, drawing a curve in a three-dimensional space using a two-dimensional display is no simple task.



(a) Profile curve (b) Profile curve with path (c) Resultant swept surface

Figure 2.1: Swept surfaces
Source: SolidWorks Knowledge Base [2]



Swept Surface Definition

Profile type:

Profile: No selection

Guide curve: No selection

Optional elements

Reference | Second Guide

Surface: No selection

Angle: 0deg Law...

Spine: No selection

Relimiter 1: No selection

Relimiter 2: No selection

Smooth sweeping

☐ Angular correction: 0.5deg

☐ Position profile Show parameters >>

OK Apply Cancel

Figure 2.2: CATIA's swept surface dialogue box
Source: Screenshot by Prof. Ahmed Kovacevic [4]

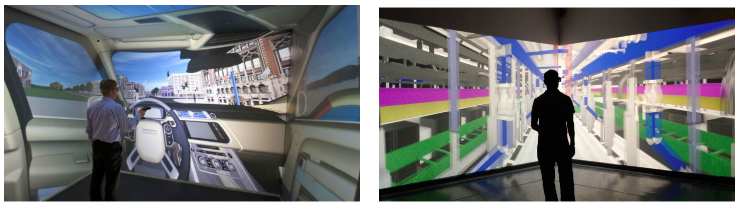
As such, some other programs (e.g. CATIA) opt to display a dialogue box that allows you to define certain properties of the surface (fig. 2.2). While manipulating a dialogue box may be easier than drawing a line in three dimensions, the options available to be changed can be overwhelming for a new user, but also restrictive for an experienced power user. It is also difficult to visualise what the resultant surface could look like if all you can see are the options in a dialogue box.

Extruded shapes and swept surfaces form the basis of a lot of three-dimensional modelling, for example creating a cylindrical object can be achieved by creating a circle profile and extruding it along a straight line. In fact, any shape with the same cross-section throughout the entire shape can be created in this manner. As such, this project could have potential future application with regards to immersive three-dimensional object modelling.

2.1.2 Virtual Reality

Virtual reality (VR), sometimes called artificial reality, describes a computer generated simulation of a three-dimensional environment that simulates physical presence and allows users to interact with it in a seemingly real manner, usually with specialised equipment or peripherals, such as a pair of gloves with embedded sensors. Virtual reality has a long history of use in the automotive, manufacturing and urban design/construction industries, where it has been used to create virtual models of vehicles and buildings as cheaper alternatives to producing physical prototypes. It has also seen use in training (e.g. military, flight simulators), therapy and even games.

There are two main types of immersive virtual reality systems: one is the head-mounted display, the other is the CAVE.



(a) Automotive industry (b) Construction industry

Figure 2.3: Various uses of virtual reality
Source: 2.3a [5], 2.3b [6]

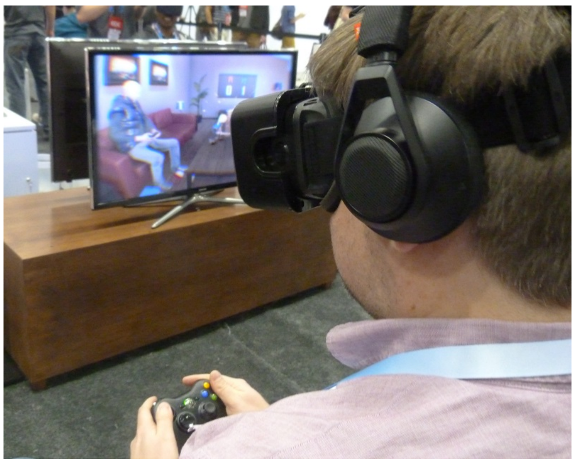


Figure 2.4: Oculus Rift Developer Kit 2
Source: Photo by PCMag [7]

Head-Mounted Display

Head-mounted displays are, as the name implies, flat display panels that are mounted on a user's head in front of his eyes, with specialised lenses to create a wide field of view effect. Computer generated graphics are rendered in real time on the panel(s) perspective-correct to the user, and are updated each frame according to the position of the user's head.

However, because the panels are directly in front of the user's eyes, it effectively blocks out the real world. This includes things like the user's body which, in a real life situation, should be visible. This is useful for some types of media, for example, games or movies, but it is not suitable for this project, as the project is concerned with the user's interactions.

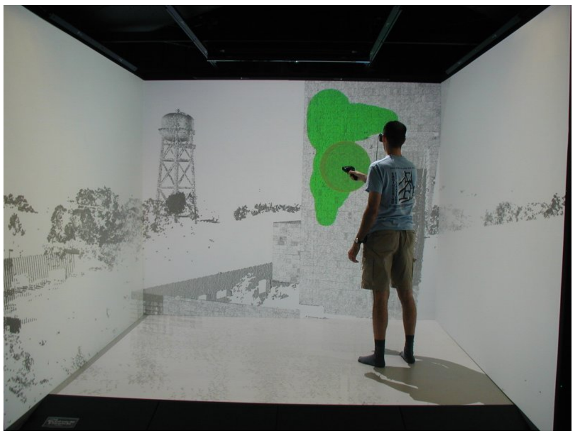


Figure 2.5: Cave Automatic Virtual Environment
Source: Photo by Oliver Kreylos [8]

CAVE

CAVE stands for “Cave Automatic Virtual Environment”, and essentially consists of three to six walls arranged in a cube shape where each of the sides is a screen. This allows for the virtual environment to be projected all around the user, meaning that the user can rely on intuitive actions like turning his head to see things to the side. A CAVE system typically will have a number of trackers as part of the system that allow the position of the user, and possibly also a body part (usually an arm) to allow interaction with the scene.

While using a CAVE system, the user’s entire body is in the system, so the user is able to rely on familiar physical interactions to interact with the environment, as opposed to using some form of controller with a head-mounted display. This means that interactions in a CAVE are going to be more intuitive and feel more natural to a user, as they are simply regular movements.

2.1.3 3D User Interfaces

The term “user interface” is commonly associated with software user interfaces, like the WIMP (**W**indows, **Icons**, **M**enus, **P**ointer) design paradigm for modern graphical user interfaces. However, the term actually encompasses all the means through which users interact with machines, including physical input devices like mice, keyboards, controllers, and even voice input.

For a two-dimensional device, input devices that move along a plane (e.g. mice) or are a plane themselves (e.g. touch screens) are sufficient. But once an extra dimension is added, these devices are no longer capable of interfacing properly, as we can already see through 3D modelling programs, which rely on projecting the 3D space into two dimensions. However, in a real physical three dimensional space, a projection into two dimensions simply isn't possible.

In order to interact with a physical 3D space, users need to be able to input spatial and positional data in a natural way. Some of the basic interactions that have to be facilitated include navigation, selection and manipulation, all within a 3D space. These types of systems typically involve a form of positional tracking [9], through a combination of sensors and sometimes external cameras or emitters.

While research on 3D user interfaces has been taking place for many years [10], it remains a constant challenge to provide a completely natural and intuitive way to interact with a 3D space. Many solutions rely on combinations of one- or two-dimensional interactions with buttons[11] to perform navigation within three-dimensions, such as the the FakeSpace Cubic mouse (see fig. 2.6), though some more recent breakthroughs combine these interactions using an analogue input, such as the 3Dconnexion SpaceMouse Pro.

3D user interfaces have become widely deployed after the video game industry took interest in it, and are available in the forms of the Nintendo Wii Remote, Sony PlayStation® Move and the Microsoft Kinect, and the recent boom in interest in virtual reality thanks to the Oculus Rift [13][14][15][16] has prompted many other developments like the Razer Hydra and the Sixense STEM System that powers it.

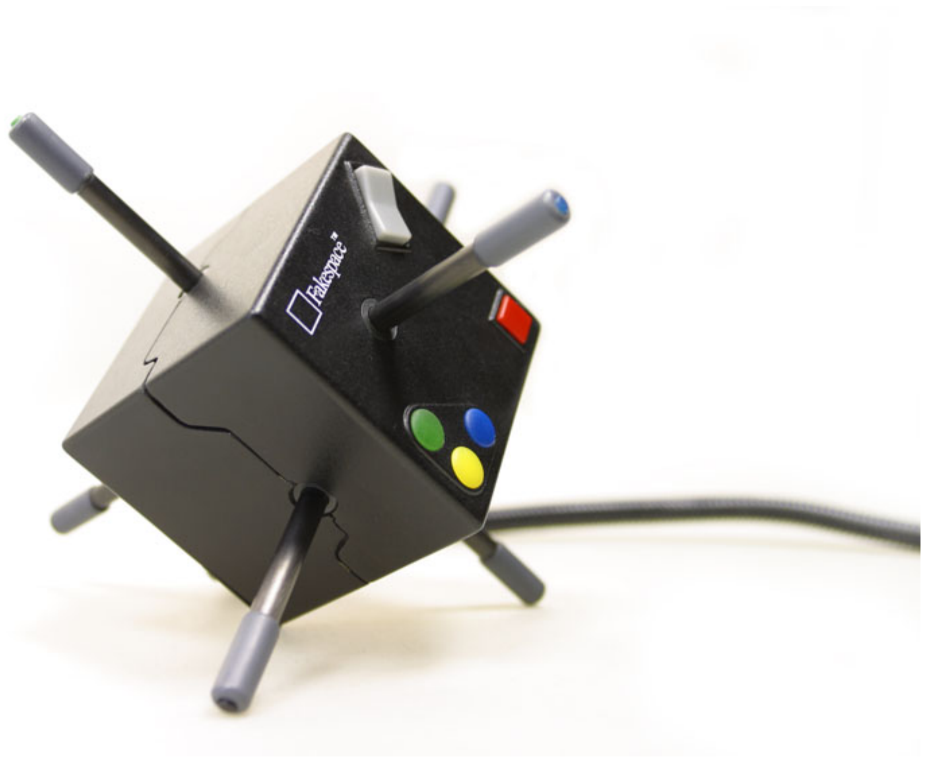


Figure 2.6: FakeSpace Cubic Mouse
Source: Photo by Dr. Ernst Kruijff [12]

2.2 Software and Tools

This project made use of a number of existing pieces of software as fundamental building blocks. This section briefly explains what they are and why they were used for the project.

2.2.1 Unity

Unity, also known as Unity3D, is a cross-platform game engine that is available from Unity Technologies. The engine itself has been used to develop video games for desktop platforms, consoles and mobile devices due to its out-of-the-box multi-platform support.

Despite being a game engine, Unity has also been used to build high quality and high fidelity simulations by organisations like NASA's Jet Propulsion Laboratory [17], the

National Oceanic and Atmospheric Administration (NOAA) [18], and many others [19]. It is able to provide real-time physics simulation via Nvidia's PhysX engine and can handle up to thousands of objects. It is also very quick and easy to build powerful demos using Unity, which made it suitable for the project.

There were a number of other options available for development, including software such as Quest3D or Vizard, but these were professional-grade pieces of software with prices easily reaching thousands of pounds [20]. Unity, on the other hand, has a free edition that provided all the necessary features for development (with additions, detailed below). Unity had also previously been used in the CAVE and is still actively being used by other researchers for projects involving the CAVE, so it was a natural choice.

2.2.2 MiddleVR

Unity is a game engine used to create executable files primarily for traditional two-dimensional flat displays, and is not able to create executables for the 4-screen CAVE layout out of the box. In order to make Unity able to do this, some sort of middleware is required.

MiddleVR is a piece of middleware that is designed to make the process of configuring virtual reality devices easier for developers. It also works with a large number of peripherals, including the InterSense tracking system utilised in the CAVE system, and provides a number of abstractions and APIs that make the development of a number of behaviours simpler.

Unity actually provides an MS&T (Modelling Simulation and Training) bundle that includes support for various platforms and peripherals, including the CAVE, but it requires a Unity Pro license ($1,500 or $75 per month for at least 12 months) and a further $7,500 for the bundle. While the bundle provides a lot of useful features, the majority of the features are not useful for the project (e.g. Unity iOS Pro or Unity GridFloat GIS Importer), so it made little sense to consider this bundle for the project.

As with Unity, MiddleVR had already been used in the CAVE before, and all the existing peripherals, including the CAVE system itself and InterSense position-tracked glasses and wand, were already configured for use with it. This made MiddleVR the obvious choice for the project.

Chapter 3

Concept and Requirements

The task for the project was very broad and undefined, which meant that the scope of the project needed to be narrowed down. An initial conceptual design and a detailed set of formal requirements were created to assist in further defining the project.

3.1 Initial Concept

One of the most important aspects of the tool is to be able to generate and extrude meshes in a simple and intuitive manner. However, in a virtual reality system such as the CAVE, there are very limited ways to actually interact with the scene. In fact, the only way to interact with the scene is via the use of a position-tracked “Wand” and the buttons on top of it.

As the wand is fully tracked in the 3D space of the CAVE, providing full six degrees of freedom, the obvious solution here is to use a combination of the wand’s position and button presses as input for the tool. A profile curve can be attached to the end of the wand and (using the position tracking data) follows the wand around

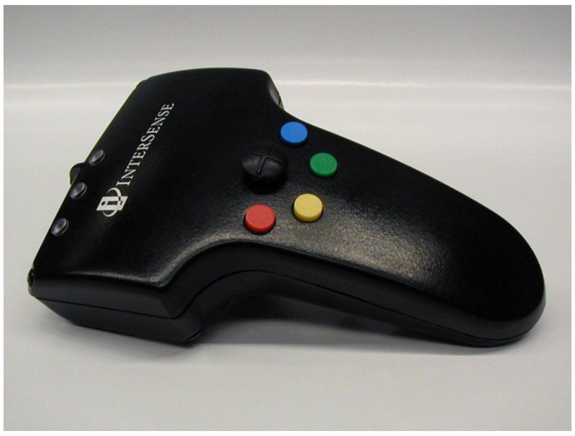


Figure 3.1: A similar wand to the one in the CAVE
Source: Photo by Oliver Kreylos [8]

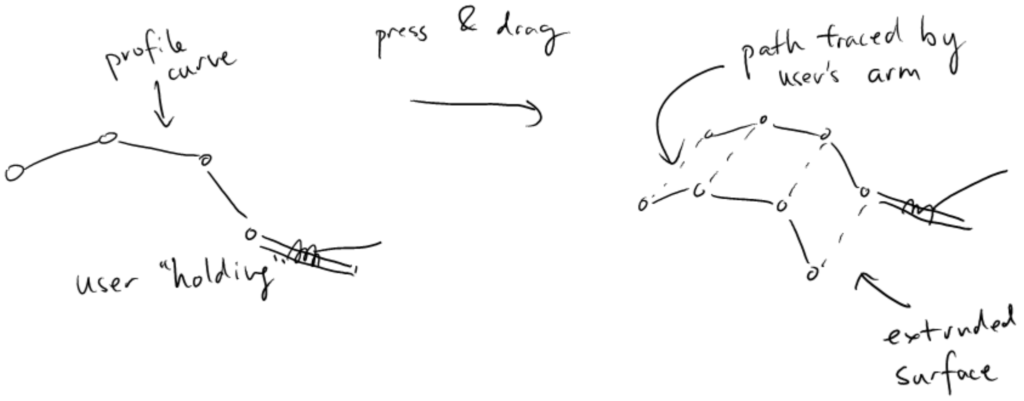


Figure 3.2: Rough concept sketch of the handle and its usage

in the 3D space. This essentially creates what resembles a physical hand-held tool, such as a hand shovel, that consists of a handle (the wand) and a blade attached to the handle (the profile curve). Upon pressing and holding a button and then moving the handle, the user traces out the 'path' of the swept surface and it would be dynamically rendered as the handle moves, as illustrated in fig. 3.2.

Compared to a 2D modelling program, this type of design would offer more intuitive interactions by presenting a familiar way to interact with the world, albeit a virtual world. As the wand isn't restricted in any way, so the "handle" can be moved and positioned as the user likes, without needing to deal with the issues of 2D projection that traditional modelling program users face.

Similarly, the amount of information presented to modellers using traditional programs is somewhat limited. This tool would offer higher quality visual feedback to the user in the form of real-time rendering that isn't fixed to a specific position. Users are able to directly see how changes affect the surface, and can freely move their heads and body to change their perspective instead of having to resort to dragging a camera around. Humans also have a sense called Proprioception which provides them with information on the relative positions of their body parts. By using their arm to physically trace out the path of the surface, the user already has an intuitive understanding of the surface, even without observing the surface from all angles [21], which should allow users to more accurately create their desired shapes.

3.2 Requirements

Two sets of requirements were defined, divided into functional and non-functional requirements, following common software engineering practices. These requirements were derived from the project task details, the initial design concept, and based on personal preference and experiences using traditional modelling software.

Presented below are the final requirements for the project after being refined based on various aspects during the design and implementation stages.

3.2.1 Functional Requirements

In software engineering, functional requirements of a software project describe the behaviours and functions required of the software. The following are the functional requirements for the project, with some reasoning as to why they were important.

- The user must be able to create meshes and extrude them.

As the project task was to create a tool for creating swept surfaces, it was obvious that the tool would need to be able to extrude meshes.

- The user should be able to manually place a set of points
- There could be an option for the system to automatically place a set of points

A mesh has to start and end somewhere. If no end point is defined, the mesh cannot really be rendered. Initially, the idea was that a user would be able to 'trace' a swept surface to create it automatically. However, it was quickly realised that automatic generation was not ideal (discussed further in section 4.2). As such, there is a definite need for manual point placement, and automatic placement became lower in priority.

- The user should have full six degrees of freedom on placing points

A traditional modelling program is limited in how a user can move or interact because the user has to work on a two dimensional projection of the three dimensional shape. This means that some tasks may be more difficult to perform. By using a virtual reality system such as the CAVE, a user is able to make use of all six degrees of freedom, so the tool should take advantage of this.

- The user must be able to define the profile curve using any number of points
- There could be an option for splines to create smoother curves without the need to manually define all the points
- The user should be able to modify the profile curve on the fly

The concept of a swept surface heavily relies upon a profile curve. As such, it must be possible for a user to define their desired profile curve. Initially, the second requirement did not exist, but was added during the design refinement stage (further details in section 4.2). Being able to modify the profile curve on the fly would also mean that users have more freedom to create their desired surfaces and shapes.

- The user should be able to remove a set of points
- The user should be able to modify individual points to affect the overall surface
- The user should be able to increase the fidelity of the model by inserting an extra set of points in the surface

If a user makes a mistake or decides they want to make changes to the curve, they should be able to do so without restrictions. Removing a set of points and modifying individual points makes it possible to correct mistakes. However, if a user decides they want to add more contours to the surface, that would not be possible unless they manually shifted all the points. As such, it was important to consider the possibility of inserting new points in between existing points.

- The surface should be automatically immediately updated when any points are changed

One of the major issues with existing modelling programs is that they do not provide immediate feedback to the user. By providing instant feedback on any updated points, the user would be able to immediately see how the change affects the surface and the states in between the original point and the new point.

3.2.2 Non-Functional Requirements

Non-functional requirements usually describe specific criteria that the software is required to meet.

- The tool must be compatible with the CAVE
- The tool must be compatible with existing peripherals and sensors used in the CAVE (if needed)
- The tool should be simple and intuitive to use
- The tool should run efficiently and smoothly even with a reasonably large number of points

Chapter 4

Design and Analysis

Given that the tool had to be compatible with the CAVE and that Unity and MiddleVR is currently the go-to solution for development in the CAVE, it made sense that this would be the starting point for the project. However, the usage of the Unity meant that the design of the tool was limited to the structure that Unity supports, which can be likened to a hierarchical scene graph.

4.1 Base structure

Unity is primarily a game engine and it is designed to make it easy for developers to quickly create functional game prototypes. Games built using Unity revolve around the concept of `Scene`s and objects (called `GameObjects`) placed in a `Scene` that can interact with each other. These objects can have `Component`s, such as `Script`s, `Collider`s, `Material`s, and so on, attached to them, which provide them with additional functionality or even a "physical" shape and appearance.

There are a number of objects and `Component`s that almost every Unity project would use and provide some of the most basic functions to a scene. These are a

`Camera` so users can actually see the `Scene`, a `Light` to illuminate `GameObjects` in the `Scene`, a `Plane` with a `Collider` on it to act as a 'floor' so users can stand without "falling" in the scene.

In order to create custom behaviours, it is possible to write `Scripts` (in either C# or JavaScript) that use Unity's API to manipulate `GameObjects` and other `Components` dynamically. `Scripts` themselves are still `Components`, so they can be manipulated by other `Scripts` too.

As the project is also a virtual reality project utilising MiddleVR, the MiddleVR `Components` also need to be added to the project. The MiddleVR `Components` create an abstraction for a number of things, including the peripherals used in the CAVE. This allows the peripherals to be referenced and interacted with from within Unity projects, for example by attaching one or more `GameObjects` to a tracked peripheral.

4.2 Design refinements

While the initial concept was quite detailed, it soon became clear that not all aspects were ideal. There were two main refinements that were already mentioned in the Requirements (section 3.2) that will be detailed below.

4.2.1 Profile curve definition

The profile curve was initially to be defined completely by points. After attempting to define smooth curves manually, it became clear that it was very intricate and frustrating process, and that manual efforts could only produce subpar results. As such, the idea of generating smooth curves using control points (e.g. Bézier curves) was added to the requirements.

4.2.2 Automatic point generation

The initial design for surface generation was for a user to hold a button and then drag the wand to trace out a path for the swept surface. This would involve automatically laying points along the traced path as a mesh requires starting and ending positions.

However, a number of issues were identified with this model. The most importantly issue was that it would be extremely hard or impossible to create very detailed or well articulated swept surfaces because the user would not be able to control where points were placed.

The initial idea was for the point placement to be based on distance. Once the handle was a certain distance away from the previous set of points, a new set would be placed. As this was the cause of the above problem, some alternative ways of automated point placement were considered.

One possible solution was for point placement to be based on changes in the direction of the path. As such, if the user drew a path that contained a corner, the corner would definitely have a point inserted, regardless of how close it was to the previous set of points. However, this approach also had a few issues associated with it. Particularly, what angle should be considered the threshold for determining whether the direction has changed or not.

If the threshold angle is too high, it means only major direction changes (e.g. 90°) would have points inserted, which would mean that smaller curves would not be created properly. Alternatively, if the threshold angle is too low, a single curved path could easily create up to hundreds of points. This is even more true if the input (the traced path) is shaky, which can often be the case due to the fact that it's a human arm operating the tool. This would result in a very jagged curve due to all the extra sets of points inserted by the shakiness of the input.

Another solution considered was to use small pause in the trace as a gesture indicating that a set of points should be inserted at that specific position. This would be close to a manual point placement, but does not require direct interaction by the user. However, this interaction was deemed to be unintuitive and could cause similar jaggy curves due to shaky input or even unintentionally insert points due to slight hesitation by the user.

As a result, the final solution was simply to introduce manual point placement. This would give users full control over the surface they wanted to create, allowing them to place as many points as they wanted, wherever they wanted. The behaviour would be more complex than just the original "press-and-drag", but the overall interaction remains somewhat similar in that a smooth arm-sweeping motion can still be used. The only difference would be that the user presses a button each time they want a point to be introduced.

4.2.3 Program architecture

Figure 4.1 illustrates what the basic initial architecture of the tool was intended to be like before any implementations were attempted. Aside from the base structure, the key sections of the program were the Handle and the Surface Generator.

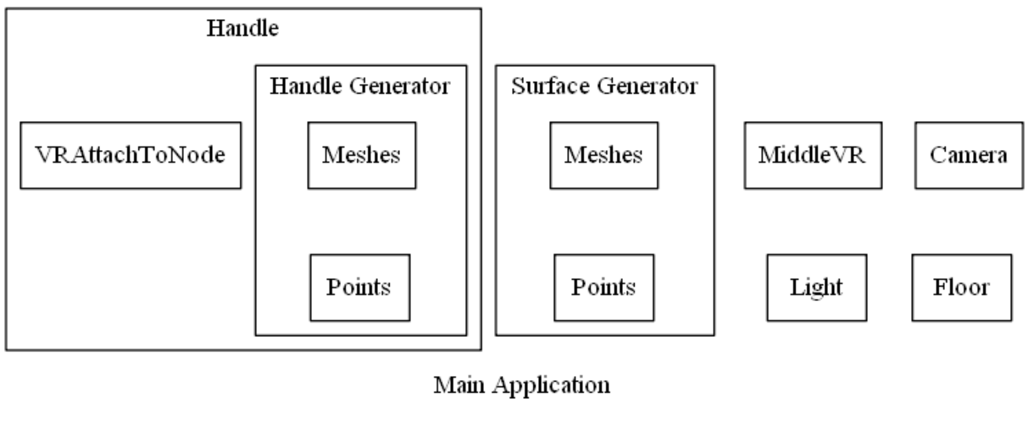


Figure 4.1: Initial architecture

The handle contains a `VRAttachToNode Script` that is provided by MiddleVR, used to attach an object to a specific abstracted node. In this case, it is the `HandNode` which abstracts the wand's position. The other portion of the Handle is the Handle Generator, which contains the points or control points that make up the "blade" of the tool and generate meshes in between the points.

The other portion of the program is the Surface Generator, which would be responsible for generating the resultant surface. When the appropriate button on the wand is pressed, the surface generator would insert a set of points at the same position as the handle, and link it up to the existing surface (if one exists already).

Despite the initial architecture design, as implementation began, a number of aspects had to be modified to achieve the desired result due to unforeseen issues arising. The final architecture is similar to that illustrated by figure 4.2. The exact details of what was changed and why are covered in chapter 5.

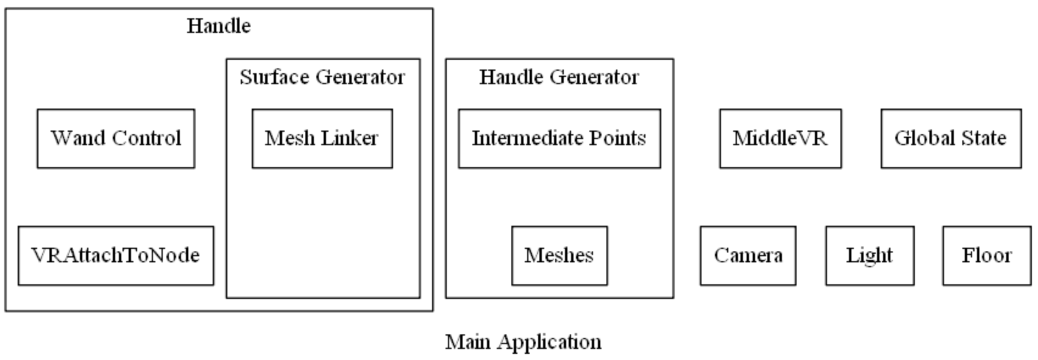


Figure 4.2: Final architecture

Chapter 5

Implementation

The project was implemented in an iterative and incremental development model, with a single prototype built into a functional tool feature by feature. These features were identified based on the requirements and the design.

It's worth noting that Unity was a brand new engine and tool for myself and I had no prior experience using or developing for it. Building a tool using Unity was a big learning experience.

5.1 Overview

The major implementation milestones for the project were centred around different features and requirements for the project. The starting point for implementation was on the creation of the handle, which incorporated various aspects of mesh generation and manipulation, as well as an implementation of the de Boor algorithm.

Once the handle was complete, the focus shifted onto the behaviour of the handle, and the interactions that a user could perform, including the placing of points, which was the core functionality of the tool. Further interactions included editing points and on-the-fly mode switching for editing and mesh generation.

5.2 Mesh Generation

The first milestone for the project was to understand how meshes worked in Unity and create a `Script` that generates a basic mesh at a desired position.

The way meshes work in Unity is quite similar to that of traditional modelling programs. Meshes consist of a set of vertices representing each corner, and a definition of the triangles that compose the overall mesh.

```
Mesh mesh = new Mesh();
Vector3[] vertices = new Vector3[4];
for (int j = 0; j<4; j++)
        vertices[j] = new Vector3();
int[] tris = new int[] {0, 1, 2, 2, 3, 0};
mesh.vertices = vertices;
mesh.triangles = tris;
```

In order to display a mesh in the `Scene`, the mesh has to be added to a `MeshFilter`, which in turn has to be put onto a new `GameObject`. This `GameObject` then requires a `MeshRenderer` component to actually render the mesh.

```
GameObject obj = new GameObject("Mesh " + i);
obj.AddComponent<MeshRenderer>();
obj.AddComponent<MeshFilter>();
obj.GetComponent<MeshFilter>().mesh = mesh;
```

By then giving the `vertices` some co-ordinates, a rectangular mesh can be created.

5.2.1 Mesh "Extrusion"

In a traditional modelling program, the user would give the program a path to extrude along. In a real-time interactive program, that data would not exist: the user simply traces the path using the handle. As such, in order to create the effect of a mesh being extruded, the program would need to dynamically update the vertices of the mesh.

As mentioned multiples already, Unity is a game engine, and games typically have a flow of time. As such, Unity's `Scripts` are structured based on events and conditions. The above code would be written on the `Start()` method, which is called when the script is first instantiated. Unity also has an `Update()` method that gets called every frame, which allows users to update elements of the `Scene` as time progresses.

By placing code that updates the vertices at one end of the mesh based on the position of the handle points in the `Update()` method, the code gets executed every frame, and the position of the mesh is updated to match the position of the handle at every frame as well.

5.3 Handle Creation

As per the design, the handle is intended to be the "tool" that the user uses to create surfaces. Initial implementations of the handle involved multiple points placed within the handle `GameObject`, and a `Script` that generated a mesh between them.

In fig. 4.2, it is shown the Handle generator was moved out of the Handle itself and became a standalone portion of the program despite its role. Ideally, the meshes, the generator and everything related to the handle would all be encapsulated by the handle `GameObject` itself to create a clean and logical scene graph.

However, Unity uses a hierarchical scene graph structure, which means that any transformation applied to a parent node is also automatically applied to its children. Because the handle is a dynamic moving object that has a transformation applied to it every frame, the children of the handle node will also have the same transformation applied, including the meshes. This would have been fine if the meshes remained static and did not change. However, the requirements stated that the user should be able to modify the profile curve on the fly.

Unlike a regular `GameObject`, the position of a mesh can be changed in two ways: first, by transforming the `GameObject` that contains the mesh, or secondly, by changing the position of the vertices of the mesh itself. The updating of the handle meshes was approached in a similar fashion to that described in section 5.2.1, with the vertices being recalculated every frame based on the positions of the profile curve points. However, given that the handle doesn't link to anything directly, it didn't make sense to use rectangular meshes. Instead, the topology of the mesh (`MeshTopology`) was changed to a LineStrip, which is essentially a single line mesh consisting of multiple short lines that join the vertices together.

By making the handle generator and the profile curve mesh itself standalone, their `GameObject` positions never change, so the calculation of the new positions of each vertex of each mesh becomes a matter of using the co-ordinates of the points that they link.

5.3.1 Smooth curve generation

During the design phase (section 4.2), it was identified that a way to generate smooth curves using control points should be introduced as manual curve creation was extremely difficult. Initially, it was thought that Bézier curve generation would be suitable for this, due to their frequent use in computer graphics. However, it

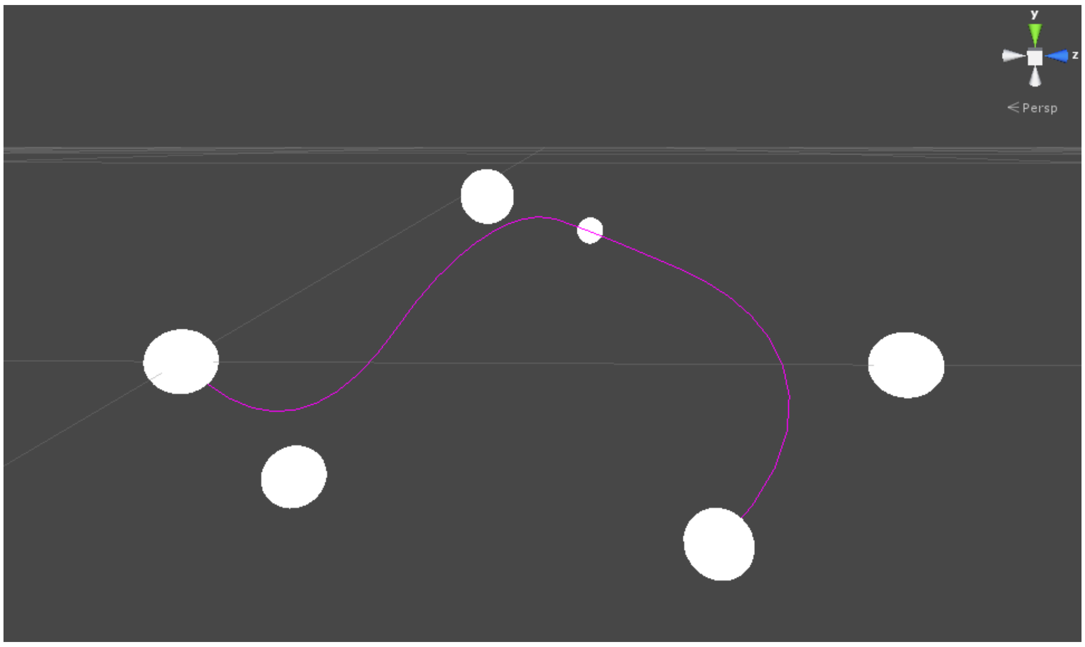


Figure 5.1: A complex 3D B-Spline

turned out that Béziers of orders above cubic were computationally expensive to evaluate. This wasn't ideal for the tool as it essentially put a limit of four control points on the profile curve. Instead, B-splines were chosen to do the job.

A well known algorithm for evaluating B-splines, the *de Boor's algorithm*, was implemented, which essentially calculates the position of a point on a spline defined by the control points for a given offset.

By iteratively calculating the positions of all the points on the spline to a specified level of detail, individual meshes could be created joining up all these points into a smooth profile curve. Obviously, the higher the detail level, the smoother the curve would look, but the more computationally complex the calculations would be.

```
void UpdatePoints() {
        int counter = 0;
        Vector3 point = Vector3.zero;

        for(float i = 0.0f; i < knotVector[degree + Points.childCount]; i
            += detail) {
```

```
            for(int j = 0; j < Points.childCount; j++) {
                   if(i >= j) {
                          point = deBoor(degree, j, i);
                   }
            }
            intermediatePoints[counter].transform.position = point;
            counter++;
     }
}
```

5.4 Point Placement

As detailed in section section 4.2, it became obvious that the best solution to allow users to create surfaces that they want to create is to provide a manual method for users to insert a set of points.

It turns out that the easiest way to insert a set of points at the same position as the points in the profile curve was to clone them.

```
GameObject Points = (GameObject) Object.Instantiate(gameObject,
   HandNode.transform.position, HandNode.transform.rotation);
Points.name = "Point Set " + Game.pointNo;
Game.pointNo++;
Points.transform.parent = Shape.transform;
Destroy(Points.GetComponent<VRAttachToNode>());
Destroy(Points.GetComponent<WandControl>());
```

An initial issue that was encountered was that the newly instantiated `GameObject` would inexplicably be invisible, but when another set of points was inserted, another two sets were inserted. It turned out that the new set of points would still retain all the scripts that the original `GameObject` also had. As the original was the handle, that means the `VRAttachToNode` and `WandControl` scripts were still attached to it. This caused the new set of points to also follow the Wand around and duplicate itself when the appropriate button was pressed.

5.4.1 Undoing

Humans make mistakes. There is no doubt about this. This is why modern programs that take user input (e.g. word processors, image editors, modelling programs) usually provide the user with a way to undo the last action. This tool is no different. If a user places some points and then decides the result is not what was desired, the user would want to correct it without having to restart.

To implement this feature, it was first necessary to track which set of points is the most recently placed set. To do this, a global variable was created that keeps track of the number of points that have been placed. By numbering each of the sets of points that had been created, this allows the set of points that bears the most recent number to be found by searching for the appropriate name and simply destroyed, decrementing the number of points in the process.

```
void undoPoint() {
        Destroy(GameObject.Find("Point Set " + (Game.pointNo)));
        Game.pointNo--;
}
```

5.5 Mesh Linking

Inserting points into the scene is the first step of creating a swept surface. Once a starting point has been placed into the scene, the surface needs to be rendered, as surface goes from the start point to wherever the profile curve currently is, as briefly touched upon in section 5.2.1.

However, as explained in 5.3, the code that generated the profile curve mesh was made standalone. This meant that the newly inserted set of points did not have any meshes to work with. To address this, the handle generator code was modified and adapted into a `Script` that generates the profile curve and then links the meshes up to the next set of points (or the handle if there was no next set). This `Script` was then automatically added into every newly created set of points.

As a swept surface is linearly swept along a surface, joining one set of points to another set of points via meshes was a matter of matching the the points in one set to the corresponding points in the next set. Each of the points in a set are numbered in order of creation, so linking up the mesh was a matter setting the vertices of the mesh to the position of the points.

5.5.1 Mesh Rendering

A critical issue involving the generated meshes was identified late during the development process, where the generated meshes would not always appear visible when the project is built and run in the CAVE, although all the meshes are perfectly visible when previewed in Unity. Even more curiously, the issue only affects meshes generated by the `Script` that links meshes, and not any other script, despite the implementations being almost exactly the same. The issue was thought to be caused by either improper textures being applied or back culling.

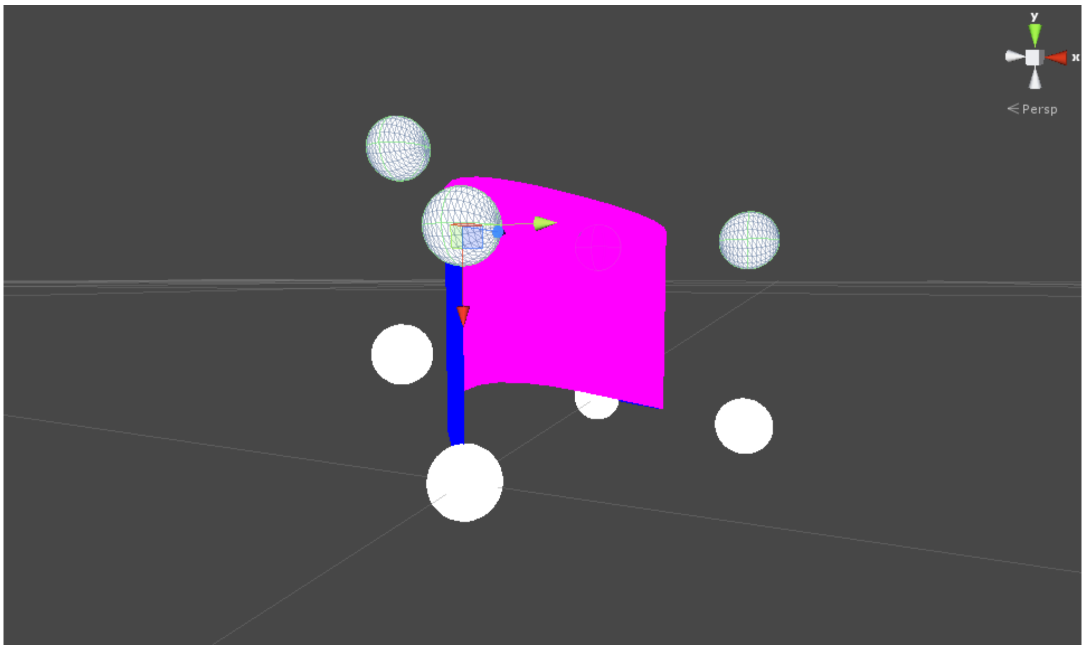


Figure 5.2: A complex swept surface

To address this, the mesh generation procedure was modified in two ways. First, a texture was properly mapped to the UV co-ordinates of the mesh. Second, the code was changed to instead generate two meshes, with triangles and normals defined in a careful and precise manner so that the meshes are back to back. This way, even if the surface is back culled, there is another mesh on the back that is facing that direction, so the mesh should still remain visible.

However, despite these attempts and lots of further troubleshooting, the cause of the problem could not be identified, nor when the issue was introduced, so the issue was not fully addressed.

5.6 Point Editing

One of the requirements that had still not been addressed in any fashion was the modification of individual points to affect the overall surface.

In order to create a more easy to comprehend experience, it was important for the "editing mode" to be distinct from the regular "mode" in some way. The solution that makes sense is for the handle to change. In the regular mode, the wand is the handle, and allows users to place points. But in "editing mode", the user has to set aside the handle. This conveys the message that the wand now has a different function.

5.6.1 Detaching and reattaching the handle

In order to stop the handle from following the `HandNode`, its parent needed to be changed to another node. This node would have to preserve the position of the handle as it would be confusing if the handle were to suddenly move or disappear. To achieve this, a dedicated `GameObject` would be dynamically created (or reused if already existing) that gets moved to the position of the wand. Then the parent of the handle `GameObject` is set to the dedicated `GameObject`.

```
void detachFromHand() {
        Vector3 position = transform.position;
        NewHandNode.transform.position = position;
        transform.parent = NewHandNode.transform;
}
```

Reattaching the node to the hand was a bit more of a challenge. Simply switching the parent back to the `HandNode` did not cause the intended result. Instead of moving back to where the `HandNode` currently was, the handle stayed where it was, but all transformations were applied onto it.

Clearly, this was unintentional behaviour. The cause of this issue turned out to be related to how Unity handled the positions of objects. If an object is transferred to another parent, its physical position and orientation don't change, but the original position it had gets converted into a `localPosition`, i.e. its position relative to the parent. Thus by setting the `localPosition` and `localRotation` to zero, the handle is able to snap back to the position of the wand regardless of where the wand was detached.

```
void reattachToHand() {
        transform.parent = HandNode.transform;
        transform.localPosition = Vector3.zero;
        transform.localRotation = new Quaternion(0.0f, 0.0f, 0.0f, 0.0f);
}
```

5.6.2 Transforming the point

Transforming individual points is by no means an innovative new feature; it is present in traditional modelling programs too. But as before, users of a traditional modelling program have very limited freedom of movement. On the other hand, the user of this tool is standing in a virtual environment with full six degrees of freedom for input. Thus it made sense to make use of this freedom to provide users with a powerful way to modify points.

After attempting to write a `script` that mirrored the transformations applied to the wand to a 'selected' point, it was discovered that MiddleVR actually provided a `Script` called `VRWand Interaction` that did exactly this. By enabling the wand after the handle was "detached", this `Script` would allow users to point at a point and move it around.

5.7 Switching Mesh Modes

The final feature that was implemented was the ability to switch mesh generation mode between smooth (B-Spline) and linear. Both of the generation methods had previously been written during the Mesh Generation phase, but it required the user to manually change the script attached to the Handle Generator.

Unfortunately, the code base had been changed considerably since then that they couldn't simply be merged. Instead, a complete rewrite of the code was initiated, re-factoring and cleaning up the entire code to make it more easy to understand and more extendible.

One of the key issues faced was what to do if a user had already started generating a surface. A generated B-spline surface has far more points than a linearly interpolated surface, which means that it was actually impossible to join a spline surface to a linear surface, and vice versa. As such, the decision was made that in order to switch mode, the existing generated surface (if any), any intermediate points and meshes, and the existing handle all had to be destroyed and then re-created.

Chapter 6

Results

6.1 Testing

Testing is an integral part of any development project to ensure that the project is fully functional, and this project was no exception. However, the nature of project differs from most software engineering projects.

The project is largely dependent on the physical interaction and input provided by the user to create the visual graphical objects (i.e. swept surfaces). A unit testing environment could not hope to replicate the same results, and writing fully automated tests to check whether the surfaces are being created correctly or points are being translated properly is a tremendous task given that the amount of meshes and points in a scene can easily be in the thousands. On the other hand, visually confirming the shape is a far quicker and simpler task.

The project also is designed to work in the CAVE with MiddleVR, but the CAVE is not available at all times. MiddleVR also requires a license to use, so it was infeasible to test the tool outside of the CAVE or run automated test suites on the project when development was not occurring.

As a result, the majority of the testing was done in the form of manual procedures carried out to test each individual feature as development occurred within the CAVE itself. These procedures systematically tested each of the features as they were implemented as well as features previously implemented, essentially providing regression testing.

6.2 Informal user study

A small-scale informal user study was carried out to evaluate how intuitive and easy to use the tool actually was. Five participants were recruited, and each one was given a brief explanation of the tool and then given two simple tasks: first, to create a basic swept surface and then edit three points on it, and second, to model a sphere.

For the first task, two of the participants quickly figured out how to achieve the task, and completed it. The remaining participants, once informed which of the buttons on the wand did what, were also able to complete the task. It was evident, however, that there was no communication from the tool about what button did what, and that this should be addressed in future.

The second task proved to be more challenging to users. The intended solution was for them to modify the profile curve to form a semi-circle and then create a sphere by rotating the wand. However, what actually happened was that users didn't consider changing the profile curve at all and resorted to manually moving points. The reason behind this was that they didn't realise they could do that.

Obviously, this is a flaw in the design of the tool, as there actually is no way to modify the profile of the curve from within the tool. The tool has to be exited, then the profile modified and then the tool rebuilt in order to use a new profile curve.

While the user study was short and informal, it has allowed some important insight

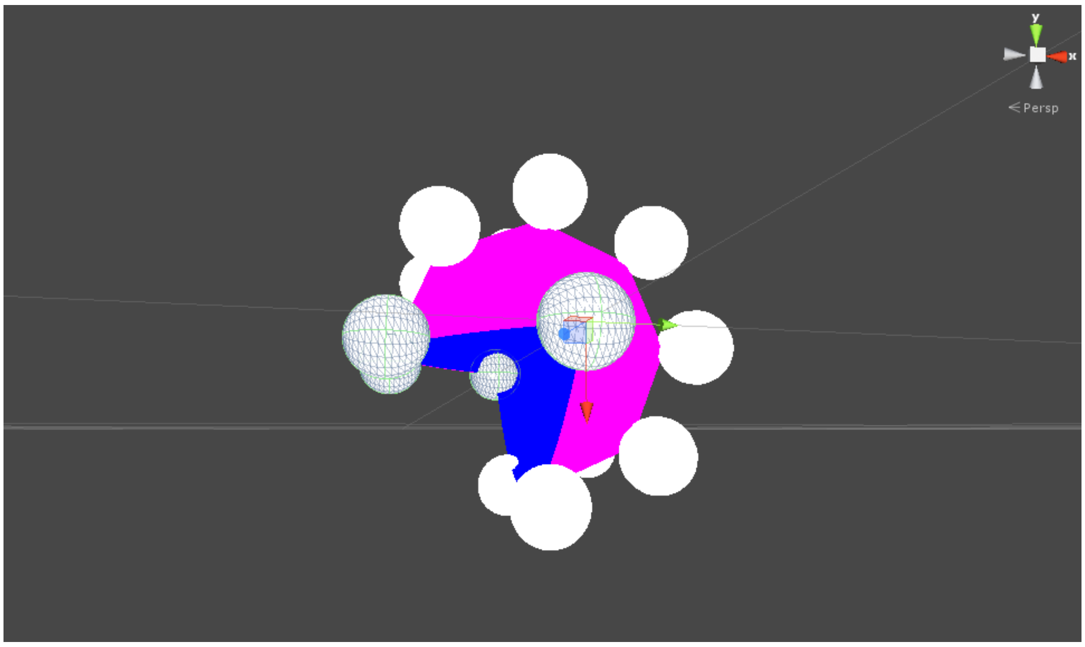


Figure 6.1: A sphere being created. Pink surface is outside, blue is inside

to be obtained. The tool is currently still a prototype, so it wasn't expected to be perfect. However, the study did show that the tool is indeed quite easy to understand and intuitive to use, which was one of the primary goals.

Chapter 7

Conclusion and Future Work

7.1 Summary of the Project

This project began with a somewhat vague suggestion of exploration regarding creating and manipulating swept surfaces in virtual reality, without concrete goals beyond that. From that starting point, a set of requirements were derived based on desirable features limitations in existing modelling software. With these requirements in mind, a prototype tool was designed and implemented.

A number of setbacks and challenges were encountered during the development phase, the biggest of which was developing for a brand new environment. Without a clear understanding of how everything worked together at first, it was difficult to get started on the project and produce desired results in a timely manner.

Finally, a very small informal user study was carried out to evaluate the tool how successful the tool was.

7.2 Evaluation

The goal of the project was to explore and create a tool for creating and manipulating swept surfaces, which was completed. The requirements for the project were self-defined and refined throughout the project, but most of the identified requirements were addressed in some fashion, with the majority directly implemented into the tool.

However, regrettably, there were a few issues that could not have been fully addressed, particularly the modification of the profile curve from within the tool (see section 6.2), and the critical bug identified late during the implementation process (see section 5.5.1).

The project followed an agile iterative development process, which was suitable for fast moving and frequently changing projects as it allowed developers to adapt. Given that this project's requirements were not rock solid, an agile approach was suitable.

The aims for the project included learning how to use the Unity game engine, as well as gaining a deeper understanding about three-dimensional interactions, both of which were satisfactorily met.

A number of aspects of virtual reality and interactions were exposed further, and the background reading piqued interest in a number of topics, especially 3D user interfaces and feedback (from the system to the user).

7.3 Future Work

This project has only produced a proof-of-concept prototype for a tool for creating and manipulating swept surfaces. There remains a great deal of potential work that can be done in future.

Saving and Exporting

As a prototype, the current tool lacks a lot of features that one would expect from a tool. For example, it is currently impossible to save the work that a user does to resume in a future session. Similarly, it is also impossible to export the created (or 'modelled') surface in any form to be used in another project or otherwise.

Visual appearance and user interface

The tool is also lacking in polish, and likely can be improved in terms of visual appearance. Currently, meshes created by the tool use pre-defined colour values, and control points are represented by large white spheres for easy identification. These are all straight from Unity without any customisation.

An issue that was identified from the informal user study was that the tool provides no information on how it should be used properly. Many interactions are hardcoded to specific wand buttons, but nothing tells the user which button does what. Some form of heads-up display could be explored as a method of providing this information to users.

Profile curve editing

One major area for exploration is the editing of the profile curve. At the moment, it is impossible to change the profile of the curve while the tool is running. In order to create different shapes, one has to either configure the profile building the tool or manually edit all the points after placing them.

There are many potential routes that this exploration could take. One could consider the possibility of making the handle editable when it is detached from the wand. Alternatively, one might create a new scene dedicated to modifying the profile. Finally, one could even consider using an extra device, e.g. a touch-screen tablet, to modify the profile curve on-the-fly without needing to switch modes or scenes, as illustrated in fig. 7.1

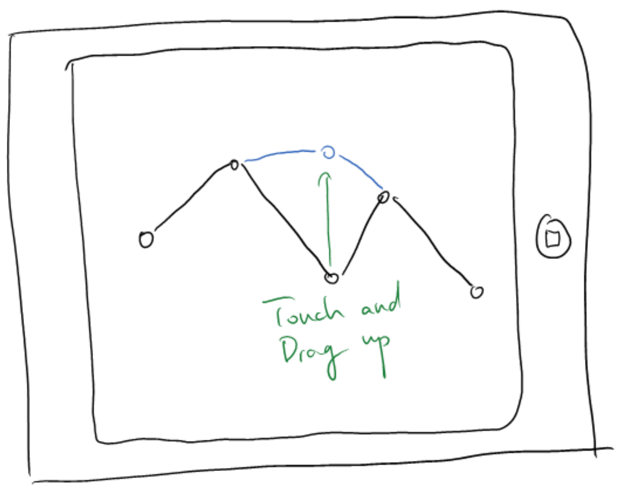


Figure 7.1: Rough concept sketch of a touch-screen tablet app

7.4 Thoughts

This project provided me with a very unique opportunity to work with the fast developing field of virtual reality, which was both exciting and challenging. The project was a valuable and enjoyable learning experience.

Bibliography

[1] Encyo. File:CATIA cre.jpg. [Online]. Available: https://en.wikipedia.org/wiki/File:CATIA_cre.jpg [Accessed: Apr. 9, 2014]

[2] Dassault Systèmes. (2013) 2013 SolidWorks Help - Swept Surface. [Online]. Available: http://help.solidworks.com/2013/English/solidworks/sldworks/t_Swept_Surfaces.htm [Accessed: Apr. 9, 2014]

[3] Autodesk, Inc. (2009) Alias Help: Surfaces > Swept Surfaces > Extrude. [Online]. Available: http://www.autodesk.com/techpubs/aliasstudio/2010/files/SurfacesSweptsurfacesExtrude.htm [Accessed: Apr. 10, 2014]

[4] A. Kovacevic. Creating Swept Surfaces. [Online]. Available: http://www.staff.city.ac.uk/~ra600/ME2105/Catia%20course/CATIA%20Tutorials/wfsug_C2/wfsugbt0203.htm [Accessed: Apr. 9, 2014]

[5] J. Williamson. (2014) Automotive simulation technology is virtually here. [Online]. Available: http://www.manufacturingdigital.com/technology/automotive-simulation-technology-is-virtually-here [Accessed: Apr. 04, 2014]

[6] WorldViz, LLC. (2013) WorldViz Brings First of Its Kind Virtual Reality to Conference and Class Rooms. [Online]. Available: http://www.worldviz.com/press-release/worldviz-brings-first-of-its-kind-virtual-reality-to-conference-and-class-rooms [Accessed: Apr. 10, 2014]

[7] D. Poeter. (2014) Hands On With the Oculus Rift DK2. [Online]. Available: http://www.pcmag.com/article2/0,2817,2455180,00.asp

[8] O. Kreylos. (2014) Pictures. [Online]. Available: http://idav.ucdavis.edu/~okreylos/ResDev/KeckCAVES/Pictures.html [Accessed: Apr. 17, 2014]

[9] D. A. Bowman, *3D User Interfaces.* Aarhus, Denmark: The Interaction Design Foundation, 2013. [Online]. Available: http://www.interaction-design.org/encyclopedia/3d_user_interfaces.html

[10] D. A. Bowman, S. Coquillart, B. Froehlich, M. Hirose, Y. Kitamura, K. Kiyokawa, and W. Stuerzlinger, "3D User Interfaces: New Directions and Perspectives," *Computer Graphics and Applications, IEEE*, vol. 28, no. 6, pp. 20–36, Nov 2008.

[11] D. A. Bowman, E. Kruijff, J. J. LaViola Jr, and I. Poupyrev, "An introduction to 3D user interface design," *Presence: Teleoperators and virtual environments*, vol. 10, no. 1, pp. 96–108, 2001.

[12] E. Kruijff. technology — Institute for Computer Graphics and Vision. [Online]. Available: http://www.icg.tugraz.at/Members/kruijff/technology_html [Accessed: Apr. 15, 2014]

[13] K. Orland. (2014) Impressions: Oculus Rift dev kits ready to bring virtual reality to the masses. [Online]. Available: http://arstechnica.com/gaming/2013/04/impressions-oculus-rift-dev-kits-ready-to-bring-virtual-reality-to-the-masses/ [Accessed: Apr. 15, 2014]

[14] ——. (2014) Facebook purchases VR headset maker Oculus for $2 billion. [Online]. Available: http://arstechnica.com/gaming/2014/03/facebook-purchases-vr-headset-maker-oculus-for-2-billion/ [Accessed: Apr. 15, 2014]

[15] ——. (2014) Microsoft: We're willing to "see how the VR space evolves". [Online]. Available: http://arstechnica.com/gaming/2014/03/facebook-purchases-vr-headset-maker-oculus-for-2-billion/ [Accessed: Apr. 15, 2014]

[16] ——. (2013) Oculus raises an extra $75 million to bring Rift headset to market. [Online]. Available: http://arstechnica.com/gaming/2013/12/oculus-raises-an-extra-75-million-to-bring-rift-headset-to-market/ [Accessed: Apr. 15, 2014]

[17] National Aeronautics & Space Administration. (2014) Explore Mars! - NASA / Jet Propulsion Laboratory. [Online]. Available: http://mars.jpl.nasa.gov/explore/ [Accessed: Apr. 8, 2014]

[18] National Oceanic and Atmospheric Administration. (2014) NOAA Earth Information System (NEIS) Overview. [Online]. Available: http://esrl.noaa.gov/neis/ [Accessed: Apr. 8, 2014]

[19] Unity Technologies. (2014) Unity - SIM Profiles. [Online]. Available: http://unity3d.com/company/sim/profiles [Accessed: Apr. 8, 2014]

[20] Act-3D B.V. (2012) Quest3D Quest3D® 5.0 Specifications. [Online]. Available: http://quest3d.com/comparison/ [Accessed: Apr. 8, 2014]

[21] V. G. Macefield, "Proprioception: Role of joint receptors," in *Encyclopedia of Neuroscience*, M. D. Binder, N. Hirokawa, and U. Windhorst, Eds. Springer Berlin Heidelberg, 2009, pp. 3315–3316. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-29678-2_4826

Appendix A

Project Plan

Name:

Andrew Li

Project Title:

Manipulation of Swept Surfaces in a Virtual Environment

Supervisor:

Prof. Anthony Steed

Aim:

To create a tool that allows users to create swept surfaces in virtual environments. The tool will be created using Unity and will work using the CAVE system, and should allow users to create a swept surface by drawing a curve and then extruding it.

Objectives:

The tool will be made in stages in an iterative fashion, building one feature at a time into the tool. The stages involved will be creating curves, extruding curves to create swept surfaces, incorporation of interactive user-inputted data and CAVE system integration. As the project proceeds, it could also be possible to explore the usage of a tablet companion app to enable users to take a cross-section of the swept surface and modify the points of the surface 'on-the-fly'.

Deliverables:

- A working and functional tool
- Design specification for the tool
- Documentation for the tool
- A strategy for testing

Work Plan:

- Project start to end October (4 weeks). Literature/past work review, define scope of project
- Mid-October to mid-November (4 weeks). Refine requirements, start the initial iterations/objectives.
- November (16 weeks) to mid-February. Continue work on iterations as defined in objectives.
- Mid-February to end of March (6 weeks). Work on Final Report

Appendix B

Interim Report

Name:

Andrew Li

Project Title:

Manipulation of Swept Surfaces in a Virtual Environment

Supervisor:

Prof. Anthony Steed

Current Status:

In accordance to the project plan, planning and development of the tool has begun. As the project does not involve research or prior work, work on the tool itself could start immediately. As specified in the plan, the tool was to be made in stages in an iterative fashion. The first stage of the tool was creating curves in Unity. As Unity (and C# by extension) was an unfamiliar environment, a period of time was spent learning how to use the engine through sample code and test projects. Once a comfortable level of familiarity was achieved, work began on drawing lines between a given number of points.

Given that the lines between the points will eventually need to be extruded, it was logical to implement this by creating meshes. Meshes can be easily calculated and updated every frame, which makes them suitable for this purpose. Initially, meshes of the LineStrip topology were used to draw the lines because they produce lines. A script was written (in C#) that took in two points, took their co-ordinates, and drew a mesh that joined the two points.

It turned out that LineStrip meshes were not very easy to extrude in the desired way, so the topology was switched back to Triangles, and the script was rewritten to then calculate two triangles using the positions of the two points and the extrusion distance to create a very thin and long rectangle that would look like a line. By parameterising the extrusion distance, the value could be updated freely at any time and the resultant mesh would change size depending on the distance. The script was when modified to take a group of points (contained within a GameObject) and do this calculation for each consecutive pair and render meshes, creating a basic swept surface.

One of the issues identified was that to produce a smooth swept surface, a lot of points within very close proximity of each other would be required to create a smooth swept curve. To address this, the idea of a Cubic Bezier or a B-Spline curve drawn using the provided points as control points was introduced. B-Splines were eventually chosen for better performance with different numbers of control points and the higher order continuity that it afforded. An algorithm that can be used for this is de Boor's algorithm which is used for evaluating B-Splines.

Remaining Work:

An implementation of de Boor's algorithm has been started but remains to be completed. Once this is completed, a switch of some sort will need to be added to allow users to change between linear and B-Spline interpolation, as there can be uses for both modes.

Once again, as specified in the project plan, CAVE system integration and incorporation of interactive user data is also needs to be done. This involves making the program work within the CAVE system as well as creating additional functionality like dynamically drawing/creating swept surfaces using a tracked Wand interface device. The planned design will involve attaching the curve cross-section to the tracked Wand device, and when a button on the wand is pressed, the want would then start 'drawing' a swept surface by dropping points along the path of the wand and joining them with a mesh.

It was mentioned in the plan a possible area of exploration is the usage of a tablet companion app to enable users to take a cross-section of the swept surface and make modifications to the points 'on-the-fly'. As the development of the tool itself has not been concluded yet, this potential exploration also remains to be done.