CREATING NATURAL USER INTERFACES FOR THREE DIMENSIONAL SKETCHING

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Science

by

John DeCorato

August 2015

© 2015 John DeCorato

ALL RIGHTS RESERVED

ABSTRACT

Your abstract goes here. Make sure it sits inside the brackets. If not, your bios-ketch page may not be roman numeral iii, as required by the graduate school.

BIOGRAPHICAL SKETCH

The author was born in Manhattan, New York on August 21st, 1991. Currently, he resides in Staten Island, New York. In 2009, he started undergrad at Cornell University. After graduating with a Computer Science degree, he entered the Program of Computer Graphics the following year.

I dedicate this thesis to my family and friends whose support helped make this possible.

ACKNOWLEDGEMENTS

First, I would like to thank my parents, Douglas and Carolyn, my brother Michael, and my girlfriend Athena for their constant support.

I am very grateful to Professor Donald Greenberg for providing me with the opportunity to study at the Cornell Program of Computer Graphics. The knowledge, experience, and opportunities I have gained throughout the degree would not have been possible without someone as incredible as Don running the show.

I would like to thank Professor Kavita Bala for providing me with many opportunities to expand my knowledge of computer graphics. I would also like to thank her for being an advisor on my thesis committee.

TABLE OF CONTENTS

Biographical Sketch . . . iii
Dedication . . . iv
Acknowledgements . . . v
Table of Contents . . . vi
List of Tables . . . viii
List of Figures . . . ix

1 Introduction 1

2 Background / Related Work 3
2.1 Two-Dimensional Image Representation on Computers . . . 3
2.1.1 Raster Graphics . . . 3
2.1.2 Vector Graphics . . . 4
2.1.3 Adaptive Distance Fields: Mischief . . . 6
2.2 3-D Sketching in CAD . . . 8
2.2.1 CATIA Natural Sketch . . . 9
2.2.2 ILoveSketch / EverybodyLovesSketch . . . 10
2.2.3 Hyve 3D . . . 10
2.3 3-D Sketching in 3-D . . . 11
2.3.1 Virtual Reality / Augmented Reality . . . 11
2.3.2 3-D Printing . . . 13

3 Input Technologies and Human Computer Interaction 14
3.1 Input Classifications . . . 15
3.1.1 Modality . . . 15
3.1.2 Direct and Indirect . . . 16
3.2 Input Devices . . . 17
3.2.1 Keyboard . . . 17
3.2.2 Pointing Devices . . . 18
3.2.3 Touch Screen Devices . . . 20
3.2.4 Summary of Input Technology . . . 28
3.3 Advanced User Interaction: Input using Gesture . . . 29
3.3.1 Types of Gestures . . . 30
3.3.2 Multi Touch Gestures . . . 31
3.3.3 Pen Gestures . . . 34
3.3.4 Three Dimensional Gesture Recognition . . . 35
3.4 Summary . . . 36

4 Creating a Sketch 37
4.1 Representation of a Sketch . . . 37
4.2 Spline Curves . . . 39
4.2.1 Algorithm . . . 42
4.3 Summary . . . 43

5 Sketching in 3D **45**
5.1 Ray Casting 45
5.1.1 Generating the Ray 47
5.1.2 Ray-Triangle Intersection 49
5.2 Acceleration Structures 51
5.2.1 Bounding Boxes 52
5.2.2 BVH Tree 52

6 Line Rendering **56**
6.1 Problems With Available Line Renderers 56
6.2 Creating Joins From the Line Definition 58
6.3 Implementation 60

7 Conclusion **66**

LIST OF TABLES

3.1 A comparison of touch technology 29
6.1 Section Variables 58

LIST OF FIGURES

2.1 Zooming in on a Raster Graphics Image [3] 4
2.2 Creating a Raster Image in Sketchbook 5
2.3 Zooming in on a Vector Graphics Image [3] 6
2.4 Raster definition of an image vs Vector definition 7
2.5 An example of the infinite canvas in Mischeif. 8
2.6 Using Natural Sketch to Add Detailing to a Car. From [1] 9
2.7 An example of using Hyve3D to draw inside of a virtual environment. 10
2.8 Augmented Reality Sketching using the Gravity tablet and Headset 12

3.1 A diagram of a user pressing a resistive touch device [5] 22
3.2 A user touching a surface acoustic wave sensor [5] 23
3.3 A user touching a surface capacitive sensor [5] 25
3.4 A user touching a projected capacitive sensor [5] 27
3.5 A visual representation of the core set of touch gestures. [7] . . . 32
3.6 Examples of pen gestures and their associated functions. [2] . . . 34

4.1 An example of a sketch 38
4.2 An input stroke (a) is provided to the application as (b) a sequence of point samples. 38
4.3 An example of an artifact in the spline generation caused by oddly spaced sample points. 41
4.4 Using the distance from the control points for termination. 41

5.1 Ray Tracing in Computer Graphics 46
5.2 Rough Diagram of Projecting Stroke onto Drawing Surface . . . 46
5.3 The ray from the eye $\mathbf{e}$ to a point on the screen s 48
5.4 An example of a ray checking and failing for intersection with a box 53
5.5 A comparison between the bounding volumes of a node using a naive implementation (left) and a surface area heuristic (right). . 55

6.1 Two line segments connected together, as per default GL_LINES implementation. Note the gap from no join. 57
6.2 An example of the flaws of native line rendering in OpenGL using GL_LINES. 58
6.3 Diagram of a miter join with parameter labels 59
6.4 Diagram of a bevel join with parameter labels 60
6.5 Left: An example of a miter join artifact from a sharp curve. Right: Artifact post-correction. 60
6.6 Left: A curve using default GL_LINES. Right: Implemented system 61

6.7 The line adjacency data structure. Each instance of the geometry shader polygonalizes the line segment (p1, p2) 62
6.8 Example polygonization of a line with miter joins. 62
6.9 Even when rotating the drawing plane such that it is no longer visible, our lines remain visible. 64

CHAPTER 1

INTRODUCTION

In this work, I implement a sketching interface that allows the user to create drawings in three dimensions. The interface is capable of using modern input devices to approximate the act of real world sketching as closely as possible. These devices include a display capable of multi-touch input, as well as a special electronic pen which relays extra information not sent through regular touch input. For the sketch itself, I implement a spline-based data structure in order to store high quality, three dimensional strokes, that can be zoomed in without loss in quality. I also implement a three-dimensional line rendering library, as native line rendering implementations are poor quality.

When creating a three-dimensional model, such as a building or a character in an animated film, much of the work is done on a computer using Computer-Aided Design (CAD) software. However, the initial designs are still done using two-dimensional sketches, rough drawings not intended as the finished work. Sketches are generally not highly detailed works, as they intend to only capture the essentials of a final design. Through a number of rough sketches, a three-dimensional form can be created through the representations of perspective and volume. This two-dimensional information is then used as a reference when designing the final three-dimensional model. Details of modern approaches to content creation on a computer are given in Chapter 2.

In architecture, there is a push towards creating buildings that reduce energy consumption though heating, cooling, and lighting. If an architect has a three dimensional model, it can be analyzed to predict how well it uses energy. However, if the building has poor results, the architect can only make superficial changes to the structure of the building, since much of the design process has already been approved. If we are able to digitize the early phase design process such that a rudimentary three-dimensional model can be made, architects can analyze their models earlier in the design process, and use this information to better design energy efficient buildings.

Sketching is an old method of expressing ideas, and has a variety of techniques associated with the practice. Many professionals have been reluctant to use computer software, because the skills they have used and trained themselves in do not transfer to the digital medium. In recent years, technology has advanced to where the creation of specialized user input devices can allow better emulation of traditional sketching techniques. These devices are explored in Chapter 3.

In our interface, a user sketches through an interaction window displaying a scene. This scene contains geometry of some form that the user can draw on. Every time the user draws on an object, a stroke is created. In Chapters 4 and 5, we discuss how the rough, two dimensional user input is transformed into a high-quality, three-dimensional spline curve. This spline is rendered in real time using OpenGL. However, the native support for displaying curves is limited, as many artifacts appear in the final curve when naively attempting to render them. Details on the approach we use to display our curves in high quality is discussed in Chapter 6.

CHAPTER 2

BACKGROUND / RELATED WORK

2.1 Two-Dimensional Image Representation on Computers

At it's most basic form, a sketch can be described as an image drawn on a planar two-dimensional surface. With computer displays, there are two standard methods for working with two dimensional images: raster graphics and vector graphics. These underlying data structures have a large impact on the types of tools that can be designed to create, modify, and display the resulting images.

2.1.1 Raster Graphics

Raster graphics is an image format that uses a two-dimensional grid to represent each pixel in the image. A raster image is characterized by its width and height in pixels and by its color depth, the number of bytes per pixel. The depth value specifies the color for each pixel, usually by identifying the magnitude of the pixel's RGB components. The reasoning behind representing images by this method is that today most computer monitors have bitmapped displays. Today, although there are many standard formats, almost all displays consist of rectangular arrays of square pixels, and the bandwidth from the display's memory is sufficient enough to dynamically render multi-megapixel images.

When creating and editing raster graphics images, the software directly manipulates pixel values, also known as pixel editing. This simplifies creating tools for editing raster graphics, since each tool can manually define how pixels are

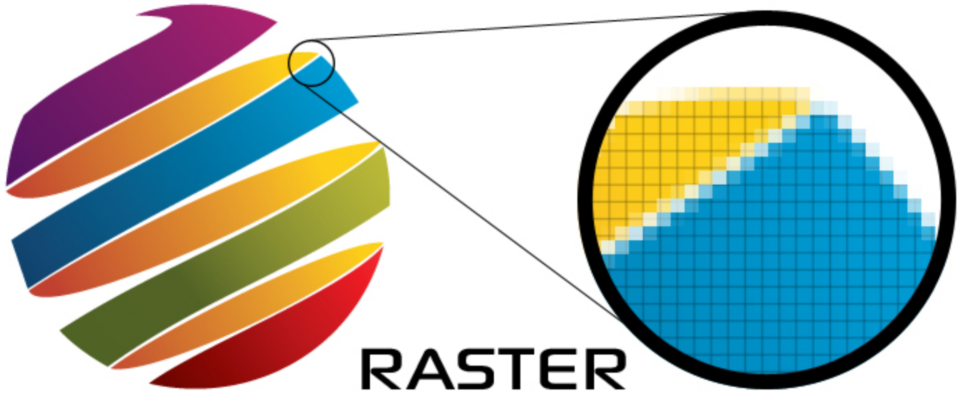


Figure 2.1: Zooming in on a Raster Graphics Image [3]

effected based on where and how an input occurs. Unfortunately, the ultimate quality of an image based on raster graphics is limited by the fact that the picture is resolution dependent. If you were to continuously zoom in on a raster image, eventually the image would suffer from image degradation.

Examples of popular raster graphics software are Corel Painter, Adobe Photoshop, Microsoft's Paint.NET and MSPaint, the open-source GIMP software, and Autodesk's Sketchbook.

2.1.2 Vector Graphics

Vector graphics is the representation of an image by the use of geometrical primitives such as points, lines, curves, shapes and polygons. Each of these primitives has a defined xy coordinate of the work space and determines the direction of the vector. Vectors can also be assigned a variety of properties such as its color and thickness. Because of their mathematical nature, they are theoretically similar to three-dimensional computer graphics, but the term specifically refers to

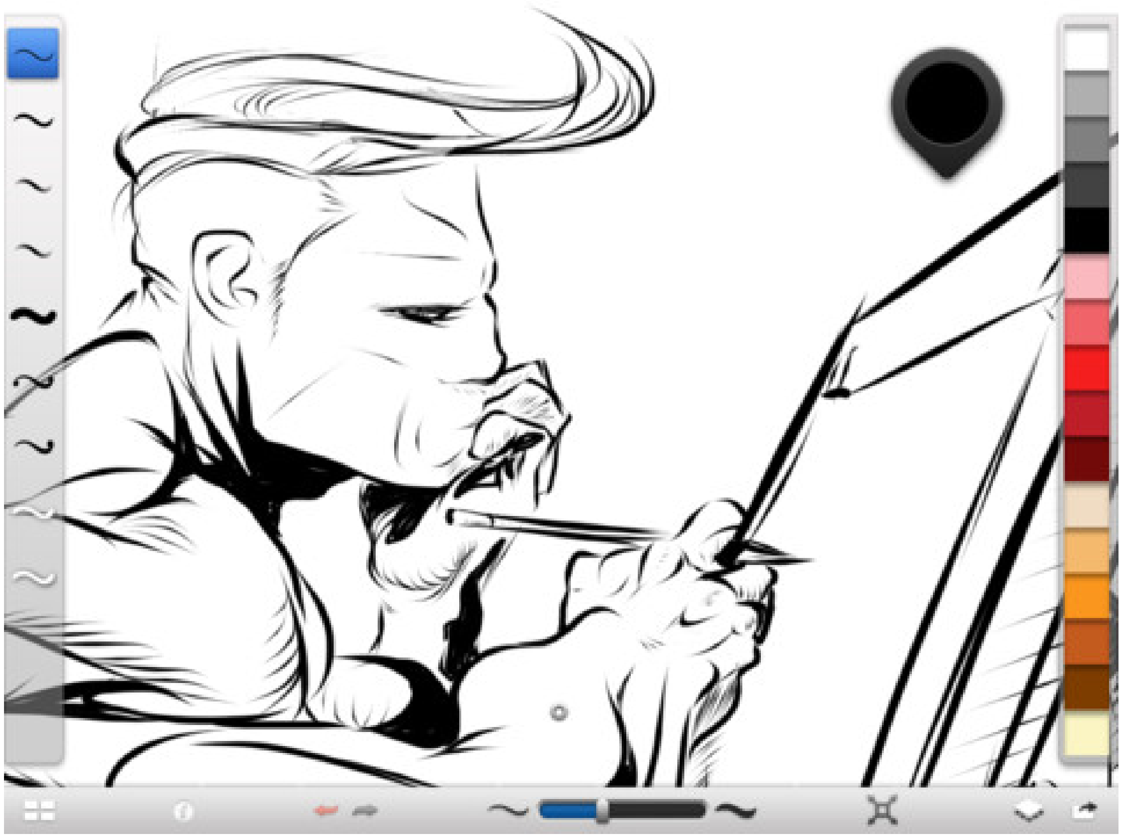


Figure 2.2: Creating a Raster Image in Sketchbook

two-dimensional images; in part to distinguish them from raster graphics. Vector graphics are primarily used for line art, images drawn with distinct straight or curved lines. For example, early CAD systems mostly used calligraphic black and white displays and rendered images in vector graphics formats. However, in modern times, vector graphics are now converted to raster graphics formats when used outside of vector specific editing software.

Vector graphics data structures offer a number of advantages compared to raster approaches. First, they are based on mathematical expressions, which means they are resolution independent. Zooming in on the image does not cause image degradation like in raster graphics; the image will remain smooth.

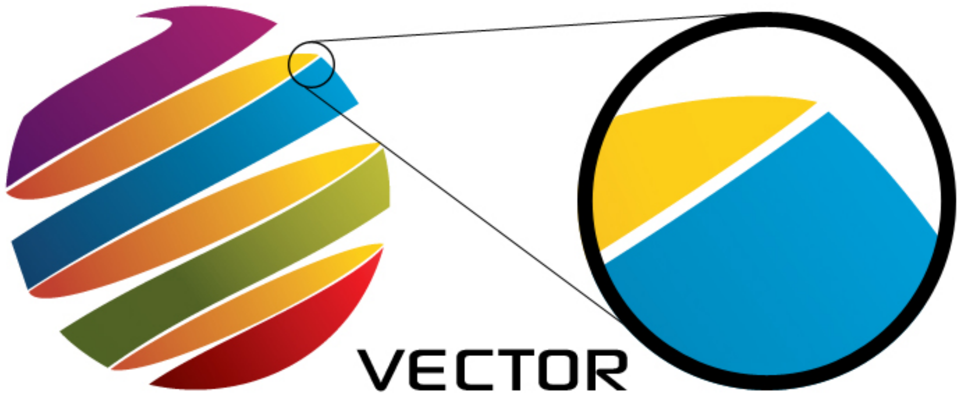


Figure 2.3: Zooming in on a Vector Graphics Image [3]

Second, objects made using vector graphics are independent from their visual representation. This allows for easy and accurate editing of primitives, provided they are contained in a vector graphics workspace. For example, say we have an image of a circle covering a part of a square. In vector graphics, the circle can be moved without effecting the square beneath, because the system knows that the two objects are a circle and a square. This type of editing is not possible in a single raster graphics image, because with the underlying pixel representation there is no way for the data structure to know the definition of objects in the image.

Examples of popular vector graphics editing software are Adobe Illustrator, Corel Draw, and Inkscape.

2.1.3 Adaptive Distance Fields: Mischief

Mischief is a pseudo-vector graphics editor created by Made With Mischief, now owned by The Foundry. Although its systems are based on vector graphics, it

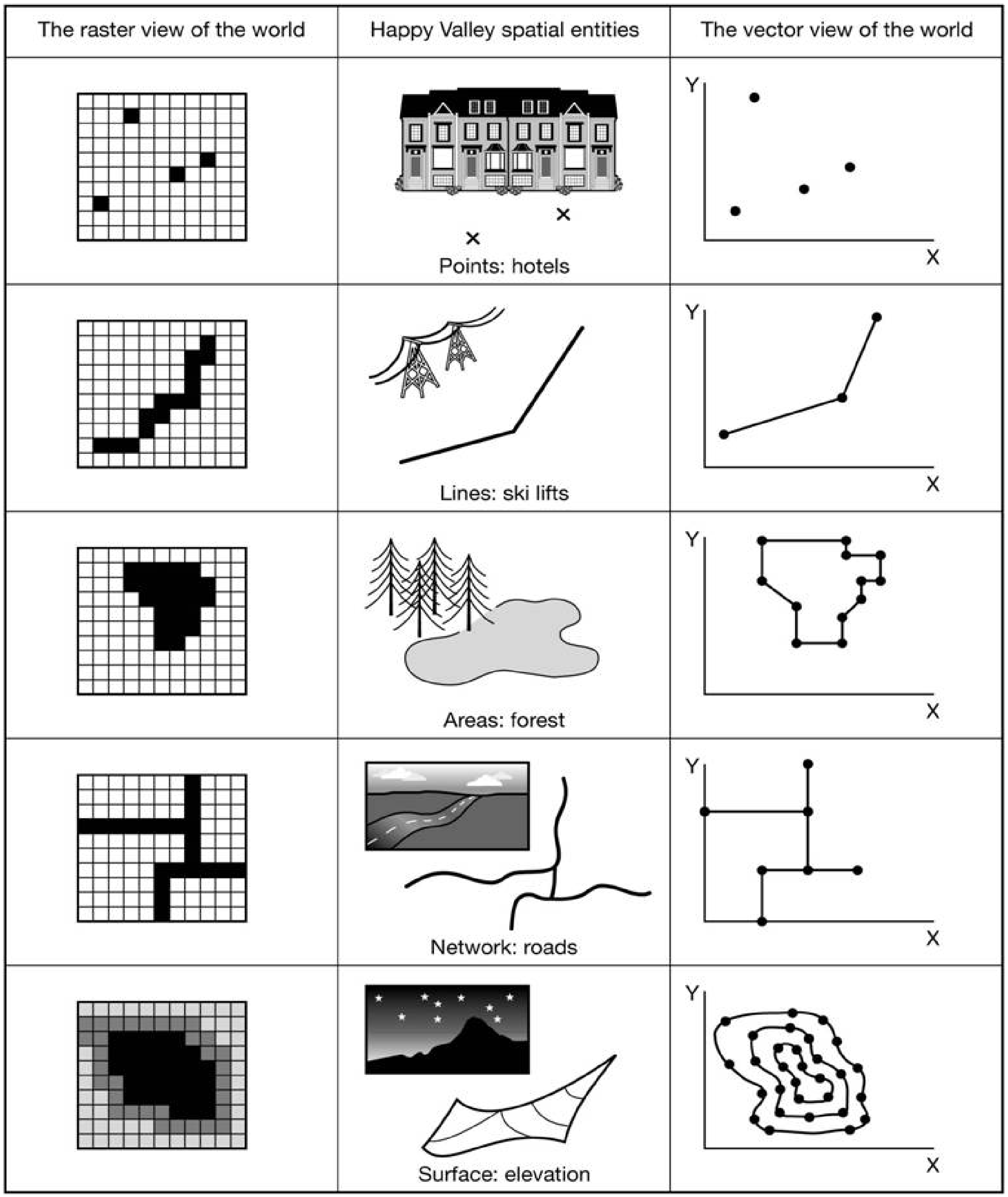


Figure 2.4: Raster definition of an image vs Vector definition

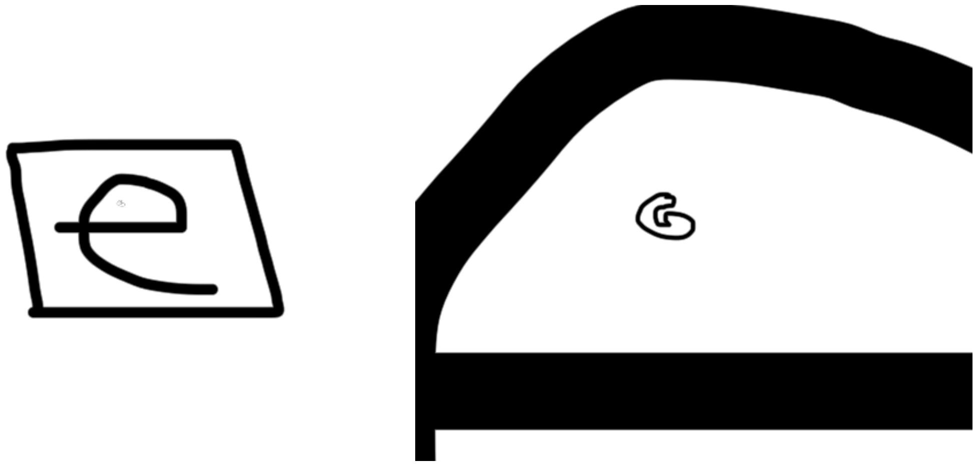


Figure 2.5: An example of the infinite canvas in Mischeif.

does not allow for the precise editing of curves as seen in traditional vector graphics programs such as Adobe Illustrator. Instead it attempts to use vector graphics to simulate real world art techniques, similar to Sketchbook. This is made possible by using a data-structure called Adaptive Distance Fields, which stores vectors in a tree data structure instead of the traditional vector format. Since the tree data structure ends up looking like a pseudo-raster, raster brushes and pens can be implemented in this system by defining what shapes go inside of the data structure. Adaptive Distance Fields also greatly reduce the storage size of large scale vectors, making it possible for the implementation of their infinite canvas; their infinitely zoom-able, translatable workspace. This unique data structure allows for incredible detail and unlimited scale.

2.2 3-D Sketching in CAD

There have been other attempts to utilize 3-D sketching in the design process. In this section, we will go over some methods others have implemented either

(a) Beginning the sketch (b) Finishing the sketch

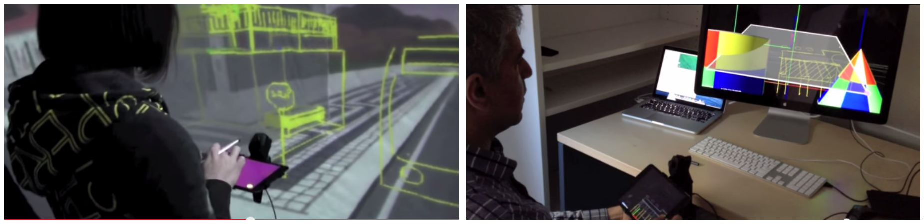
Figure 2.6: Using Natural Sketch to Add Detailing to a Car. From [1]

in research or in industry.

2.2.1 CATIA Natural Sketch

Natural Sketch is a feature inside of the CATIA modeling software by Dassault Systemes. Natural Sketch allows the user to draw on a virtual plane, a 3-D model, and the plane where the screen lies in the 3D environment. It features the abilities to alter the pen style, alter the number of control points used to make the post sketch curve, automatically change the camera view to align with the drawing plane if one is being used, copy and alter individual strokes, and generate models from the 3-D sketch.

Natural Sketch uses a two-phase design system. First, the user does a ”rough sketch”, where lines are rendered exactly as drawn. Afterwards, the user can trace over their already drawn lines to produce smooth curves from the initial sketch. The traced curves are stored as Catmull-Clark splines. While this system closely resembles something we would like to implement, it is currently only available in a solid modeling environment, making it incompatible with many tools most architects use to create 3-D models.



(a) Using the tablet to draw (b) Manipulating the drawing plane

Figure 2.7: An example of using Hyve3D to draw inside of a virtual environment. [4]

2.2.2 ILoveSketch / EverybodyLovesSketch

EverybodyLovesSketch is a 3D curve sketching system from the University of Toronto's Dynamic Graphics Project Lab. It features a pen based gesture system, allowing the user to execute functions using rapid strokes, circles, and other defined gestures. Other features include dynamic sketch plane selection, single view definition of arbitrary extrusion vectors, multiple extruded surface sketching, copy-and-project of 3D curves, free-form surface sketching, and an interactive perspective grid. This project is based off of previous work by the same lab, ILoveSketch, which is the base 3D sketching functionality of the EverybodyLovesSketch project.

2.2.3 Hyve3D

Hyve3D[4] is an infinite virtual sketching environment from the University of Montreal. It uses two screens; a computer monitor to show the 3-D environment, and an iPad to draw. The sketching plane represented by the iPad is shown in the virtual environment, and is manipulated by moving and rotating the iPad in the real world. The user then pins the sketch plane in place and proceeds to draw at leisure. The advantage of this system is that it combines real world manipulation with virtual representation, eliminating the need for complex user interfaces and gestures. The disadvantage is that this kind of movement has no one-to-one feedback between the real world and the virtual, meaning that it is difficult to judge how your movements of the iPad effect the drawing place without confirming it visually.

2.3 3-D Sketching in 3-D

3-D content creation on a traditional computer screen is limiting. Any type of input or user interface can never overcome that one dimension of the workspace will always be inferred, due to the two dimensional output. The result has been leveraging a number of emerging technologies that deal with output that is experienced in three dimensions.

2.3.1 Virtual Reality / Augmented Reality

Recently, there have been two key technologies developed with the intention to immerse the user in a virtual environment; virtual and augmented reality. Both of these involve head-mounted displays (HMDs) that display two dimensional images. Despite the images being flat, the system takes advantage of how humans see, such that the user feels the presence of actually being inside of the virtual environment. The advantage to using these virtual systems is the user can use their sense of depth to fully understand the space they are working in. The downside that these systems have is since their input methods are essen-

Figure 2.8: Augmented Reality Sketching using the Gravity tablet and Headset

tially drawing in midair, there is no tactile feedback similar to a pen pushing against a piece of paper. An example of an augmented reality approach to 3-D sketching is Gravity Sketch, and one of a virtual reality approach is TiltBrush by Skillman & Hackett.

2.3.2 3-D Printing

So far, we have only explored the idea that 3-D sketching is only possible in virtual environments, due to the inherent two dimensional techniques artists have used for centuries. However, groups have experimented with using 3-D printing to create pens that can sketch simple 3-D models; examples being the Polyes Q1 Pen and the 3Doodler. These pens work similarly to hot glue guns, except instead of glue, the pens secrete ABS plastic that quickly hardens as it exits the tip of the pen. Like the digital approaches, these pens lack tactile feedback, and rely purely on the user's sight to sketch. Unfortunately, between the apparent structural instability of even small models created by the pen, as well as the lack of flexibility in the way the pen can be used, it appears that this route is impractical for creating the types of large and intricate structures seen in modern design, and is at best a toy.

CHAPTER 3

INPUT TECHNOLOGIES AND HUMAN COMPUTER INTERACTION

An important goal for any user interface is to have the human-computer interaction be as intuitive and natural as possible. Natural interaction is critical for a sketching interface; while younger, more technologically willing professionals flock to digitized content creation software, those classically trained prefer to stick to physical methods. In order to convince the latter to adapt to a new system, the transition process must be as smooth as possible, so as many of the techniques utilized in the physical system should be emulated as closely as technology allows in the digital system.

A key element of bridging the gap between physical and digital tools are input devices, computer hardware used to control electronic devices. For a long time, the most common input devices for digital environments have been the mouse and keyboard, and are rather unintuitive ways to move and control the computer. However, in recent years there have been sweeps of advancements in the human-computer interaction field, which has produced a wide array of input devices. In this chapter, we will discuss human-computer interaction principles used to decide how to best utilize input devices, as well as discuss the particular devices needed to create a good sketching system.

3.1 Input Classifications

3.1.1 Modality

In human computer interaction, a modality is a channel of sensory input/output between the human and the computer. Modalities can be split into two general types: human-computer, and computer-human modalities.

Human-computer modalities describe the ways humans input information to the computer. These modalities are devices and sensors attached to the computer. Common input device modalities are keyboards, pointing devices, and touch screens, while more complex modalities are computer vision, speech recognition, gesture recognition, and orientation. In sketching, the equivalent of a human-computer modality is the sketch implement. Whether the tool is a pen, pencil, charcoal, or something else. the artist provides "data" to the sketching surface through his strokes by sending information like position, angle, and pressure.

Computer-human modalities describe the ways the computer outputs information to communicate with humans. This requires stimulation of one of the human senses: sight, hearing, taste, smell, touch, balance, temperature, and pain. Of these, modern input devices generally communicate using sight and hearing, because they are capable of sending information at much higher speeds, as well as being the more common ways humans communicate with each other. There are also small uses of haptics, general vibrations or other movement, to provide feedback, but generally they are accompanied by visual and auditory cues.

While working on this project, it became apparent that professionals are very sensitive to a complex combination of input and output modalities, some of which are not reflected well by current hardware capabilities. For example, an architect commented while drawing on the large display that he felt uncomfortable because the digital pen pressing against the display didn't feel like a physical pencil brushing against a piece of paper. The difference in friction and materials meant that he was unsure of how things like pressure and angles effected his drawing. This problem is a result of both the limitations of the input device, as well not providing proper feedback for his actions. On this project, we focus more on the input modalities, but for a truly realistic sketching system, specialized hardware must be utilized in order to closely emulate these intricacies.

3.1.2 Direct and Indirect

Direct and indirect input refer to how the input space corresponds to the display space. With direct input, the two are directly correlated. On a touch screen, where you touch is where the input is recorded. With indirect input devices, the input space is only relative to the input space. Moving a mouse three inches can correspond to movement of any kind on the computer screen. In desktop settings, moving the mouse to the left three inches can cause the pointer on screen to move three pixels, or three hundred pixels. In a three-dimensional application it can cause the user to move left, or turn left.

Indirect input devices have the advantage of being more application independent. However, this comes at the cost of accessibility. A mouse can do many things, but the user must learn what it does in the current context. Direct input methods tend to be intuitive while limited in scope. The user knows touching a screen at a spot causes an action at that spot.

When physically sketching, most if not all interactions are direct. There are minute indirect details such as different drawing implements can produce different strokes with the same input, but on a high level there is a one-to-one spacial correlation between actions and results. Therefore, for this project, it is important to minimize any indirect interaction between the user and the system. For any indirect interaction, we should try to minimize the complexity of the difference between the input and output spaces.

3.2 Input Devices

3.2.1 Keyboard

A keyboard is an indirect input device using an arrangement of buttons or keys, acting as mechanical levels or electronic switches. Common keyboards are typewriter style devices, with many buttons representing alphanumeric characters as well as a small number of additional function keys. Desktop keyboards usually have from 100 to 105 keys, while laptop and other small device keyboards contain less. All standard keyboards have a typing area used for letters of the alphabet, numbers, punctuation, and other basic characters. However, there are also composite devices that have keyboard like features, such as video game controllers. Game controller buttons offer contextual and situational functionality depending on the application in use.

There are certainly auxiliary areas in good sketching software where a keyboard of some kind should see use; for example, saving files. But for the core of the application, the user should never use one. Many design applications such as Maya use a combination of function and character keys to perform actions, bringing a significant learning curve in order to use these applications effectively. This results in an interface that is unnatural, and requires application specific skill-sets. For this project, we try to avoid use of the keyboard as much as possible.

3.2.2 Pointing Devices

A pointing device is a device that more easily allows a user to input spacial data to a computer. Many common input devices fall under this classification. CAD systems and graphical user interfaces allow the user to control and provide data to the computer using physical motion. These movements are then echoed on the screen in some way, whether it be by an on-screen cursor, or some change in the visual output. Pointing devices are usually controlled by either physical movement of an object, or touching a surface. Examples of devices based on motion are the mouse, trackball, and joystick, and those based on touch are the graphics tablet, stylus, touchpad, and touch screen.

Mouse

A mouse is a small, hand-held device that is pushed over a flat, horizontal surface. Older mouses used a physical ball at the base of the device, combined with sensors to detect when the ball rotates. When the mouse moves and rotation is sensed, the distance and directional information is sent from the mouse to the computer. A more modern approach is the optical mouse, which uses infrared light instead of a roller to detect changes in position.

The mouse is the oldest pointing device, and used with every desktop computer. The mouse works in an indirect space as the position of the mouse and the that of the cursor on screen are completely unrelated. When a mouse of moved ten inches to the left, the movement of the mouse cursor on screen is not ten inches, but some function with ten inches as input. As discussed previously, we would like to eliminate indirect input where ever possible. While it will be possible to use the mouse with our application, it should be seen as legacy functionality.

Tablet

A graphics or digitizing tablet is a special tablet that is similar to a touchpad. However, it is controlled with a digital pen or stylus that is held and used like a normal pen. An alternative control device for tablets is called a puck. This is a mouse like device that can detect absolute position and rotation. Professional pucks have a reticle which allows the user to see the exact point on the tablet's surface targeted by the puck, for detailed CAD work. Graphics tablets are commonly used to create 2D computer graphics because of their input similarities to traditional drawing techniques. Tablets are very commonly used for digital sketching applications. However, many graphics tablets are used in combination with a screen, meaning there is still a visual and physical disconnect between the sketch and the output.

Pen

As mentioned above, touch panels can be used with rigid styli to simulate the experience of writing. However, the pen itself can also be digitized to capture additional information for various applications. An active pen is an input device that includes electronic components and allows users to write directly onto the display surface of a computing device. The active pen's electronic components generate wireless signals that are picked up by a built-in digitizer and transmitted to its dedicated controller, providing data on pen location, pressure and other functionalities. Additional features enabled by the active pen's electronics include palm rejection to prevent unintended touch inputs, and hover, which allows the computer to track the pen's location when it is held near, but not touching the screen. Most active pens feature one or more function buttons (e.g. eraser and right-click) that can be used in place of a mouse or keyboard.

3.2.3 Touch Screen Devices

A touch screen can be considered as an input device layered on top of a display. Users provide input by touching the screen with one or more fingers, or with a special stylus/pen. This method of input allows users to directly interact with what is displayed, as opposed to using an indirect input device such as a mouse or touchpad.

One major advantage touch screens offer over other input devices is ease of use. While frequent computer users are familiar with using a mouse and keyboard, touching icons on a screen is intuitive even for those with limited to no computer experience. This ease of use can reduce the learning curve and increase productivity when using user interfaces. Touchscreens are also faster to use than traditional input methods. When a user interacts with a computer using a mouse and keyboard, there are many small adjustments the user needs to make; they need to locate the pointer, and adjust for the mouse acceleration. Direct interaction allows for users to interact with the computer without worrying about correlating the interaction space to the virtual space.

There are many ways to build a touch screen. The key points in any implementation are to recognize one or more fingers touching a display, to interpret the command these touches represent, and to communicate with an application. The three main types of touch sensing technologies are resistive sensing, surface acoustic wave sensing, and capacitive sensing.

Resistive Sensing

Resistive touch panels are composed of two thin, flexible sheets coated with a resistive material and separated by a small gap. A resistive touch monitor features a simple internal structure: a resistive panel is placed on top of a glass screen, with a polyester film screen on top of the panel used as the contact surface. Pressing the surface of the film screen causes the electrode-covered sheets between the film and glass panels to come into contact, resulting in the flow of electrical current. The point of contact is identified by detecting this change in voltage. See figure 3.1 for an example of a user touching a resistive screen.

Resistive technology is low cost due to the simple structure of the touch screen and controller circuit. Analog senors have high resolution, the most common being 4096 by 4096 dots-per-inch (DPI), as well as high accuracy, while con-

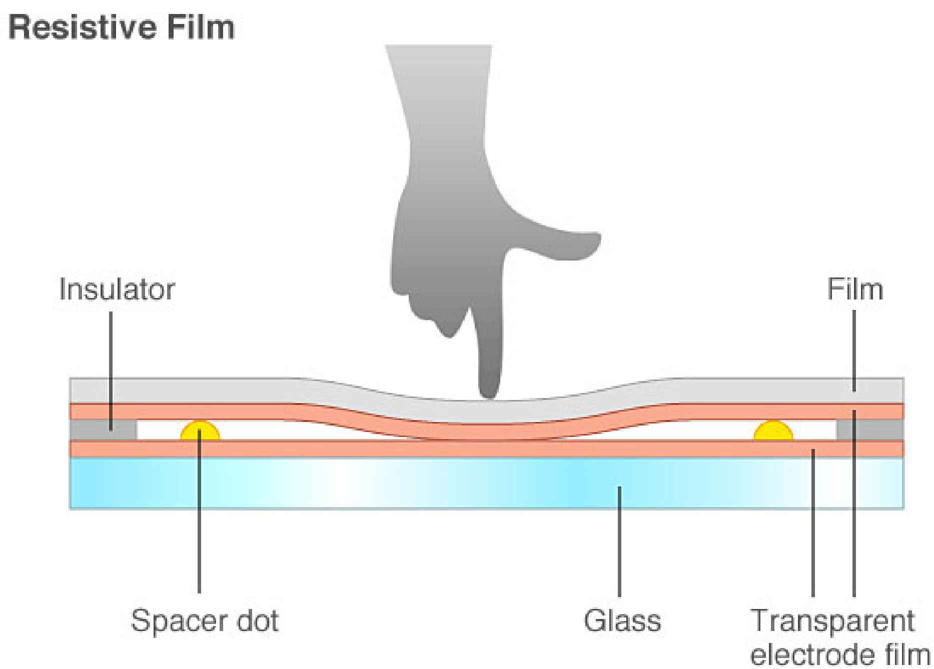


Figure 3.1: A diagram of a user pressing a resistive touch device [5]

suming low amounts of power during operation. This DPI refers to the number of sensing dots-per-inch, and is uncorrelated to the display DPI. Resistive touch screens can be used with any object, since the system only requires contact. This allows for users to use pens and gloves to interact with a device. Disadvantages include it's poor responsiveness compared to other developed sensing methods, generally working with harder presses, lower light transmittance causing a reduction in screen quality, and a decrease in accuracy in screen sizes about 24 inches. While high resolutions would allow for more accurate and more detailed sketching, the poor responsiveness would make resistive panels a poor substitute for traditional drawing media.

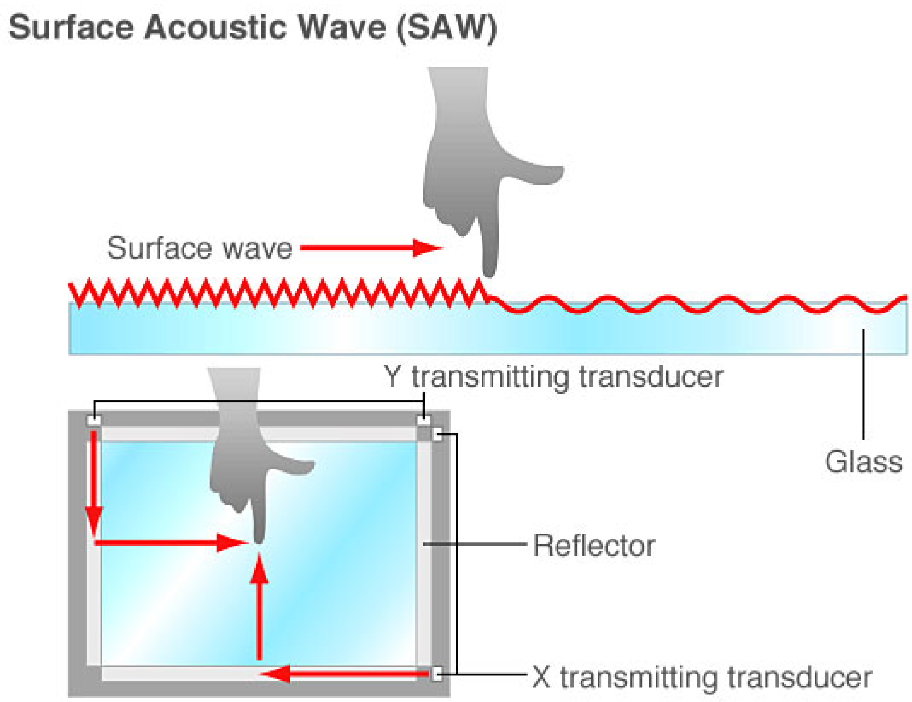


Figure 3.2: A user touching a surface acoustic wave sensor [5]

Surface Acoustic Wave (SAW) Sensing

Surface acoustic wave (SAW) touch panels were developed to achieve bright touch panels with high levels of visibility; mainly to address the drawbacks of low light transmittance in resistive film touch panels. These are also called surface wave or acoustic wave touch panels. Aside from standalone LCD monitors, these are widely used in public spaces, in devices like point-of-sale terminals, ATMs, and electronic kiosks.

These panels detect the screen position where contact occurs with a finger or other object using the attenuation in ultrasound elastic waves on the surface. The internal structure of these panels is designed so that multiple piezoelectric transducers arranged in the corners of a glass substrate transmit ultrasound surface elastic waves as vibrations in the panel surface, which are received by transducers installed opposite the transmitting ones. When the screen is touched, ultrasound waves are absorbed and attenuated by the finger or other object. The location is identified by detecting these changes. See Figure 3.2 for a diagram of a user touching a surface acoustic wave sensor.

The strengths of this type of touch panel include high light transmittance and superior visibility, since the structure requires no film or transparent electrodes on the screen. Additionally, the surface glass provides better durability and scratch resistance than a capacitive touch panel. Another advantage is that even if the surface does somehow become scratched, the panel remains sensitive to touch. On a capacitive touch panel, surface scratches can sometimes interrupt signals. Structurally, this type of panel ensures high stability and long service life, free of changes over time or deviations in position. Weak points include compatibility with only fingers and soft objects (such as gloves) that absorb ultrasound surface elastic waves. These panels require special-purpose styluses and may react to substances like water drops or small insects on the panel. This sensing method, like resistive sensing, is more suitable for public displays that see heavy use. While high quality displays are desirable, the relative inflexibility of input capabilities limit SAW powered display for use in sketching applications.

Capacitive Sensing

Capacitive touch panels use the natural flow of electricity through the human body, also called body capacitance, as the input signal. They are most commonly used in consumer level hardware such smart phones, tablets, and LCD

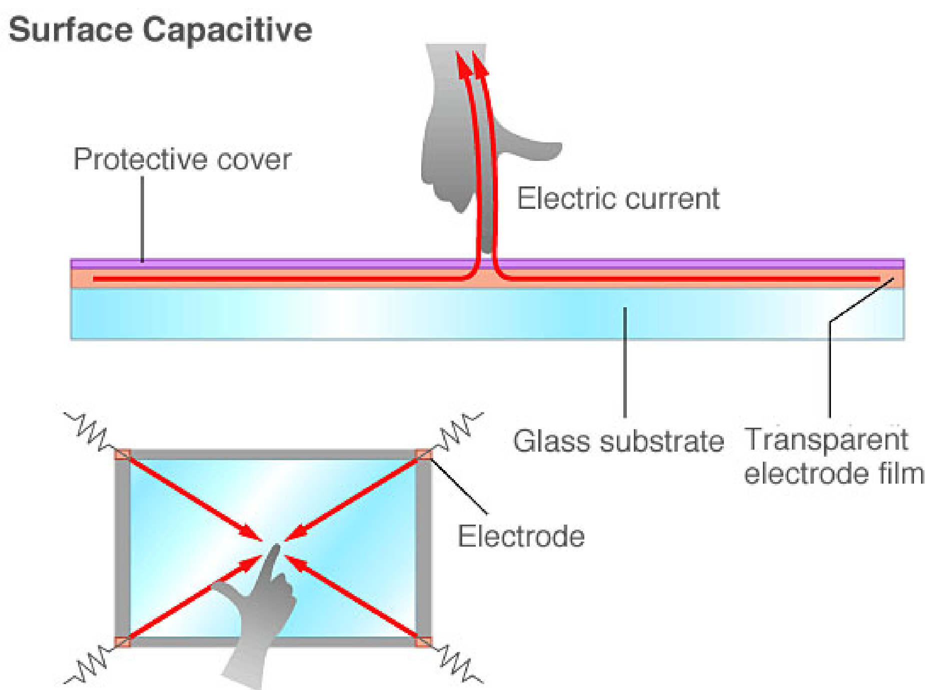


Figure 3.3: A user touching a surface capacitive sensor [5]

monitors. They are constructed from a wide variety of materials, such as copper, Indium tin oxide (ITO), and printed ink. Unlike resistive film touch panels, capacitive touch panels do not respond to touch by clothing or standard styli. They feature strong resistance to dust and water drops and high durability and scratch resistance. In addition, their light transmittance is higher, as compared to resistive film touch panels allowing for higher quality displays. There are two types of capacitive technology; surface capacitive and projected capacitive systems.

Surface Capacitive Sensing is often used for larger sized displays (over 14 inches) that are used by the general public. This accentuates their high durability and high screen quality. Surface capacitive displays can be seen on ATM machines, ticket kiosks, arcade games, automation devices in factories and offices, and in the medical industry.

A surface capacitive panel is constructed using a glass sheet. A transparent conductive coating is placed over the sheet, and a glass protective coating is placed above that. Electrodes are placed on the four corners of the panel.

If the same phase voltage is imposed to the electrodes on the four corners, then a uniform electric field will be formed over the panel. When a finger touches the panel, electrical current will flow from the four corners through the finger. The ratio of the electrical current flowing from the four corners will be measured to detect the touched point. The measured current value will be inversely proportional to the distance between the touched point and the four corners.

A surface capacitive touch panel has a simpler structure than a projected touch panel. This allows for lower cost in production, high durability, and high visibility due to the main structure being a single glass layer. However, it's simplistic structure also means it is structurally difficult to detect two or more contact points simultaneously. Surface capacitive can only detect bare finger touches, although some may detect touches through a thin pair of gloves. Some surface capacitive displays support pen writing, but not simultaneous pen and touch.

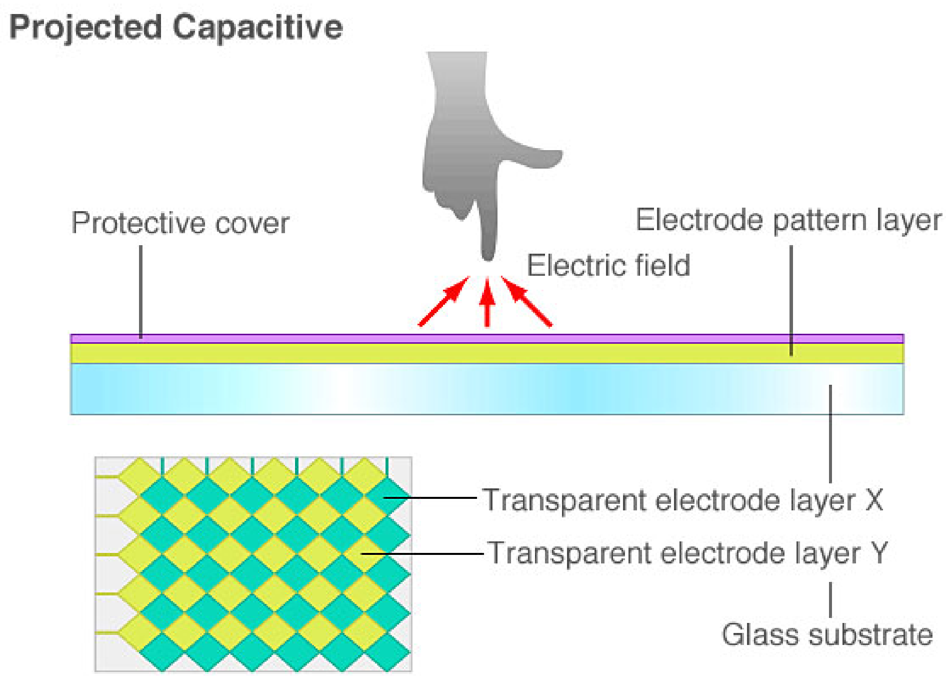


Figure 3.4: A user touching a projected capacitive sensor [5]

Projected Capacitive Sensing is often used for smaller screen sizes than surface capacitive touch panels. They've attracted significant attention in mobile devices. The iPhone, iPod Touch, and iPad use this method to achieve high-precision multi-touch functionality and high response speed. However, they have also seen use in other areas of display technology, with Microsoft's Surface Hub implementing a 55-inch projected capacitive display.

The internal structure of these touch panels consists of a substrate incorporating an IC chip for processing computations, over which is a layer of numerous transparent electrodes positioned in specific patterns. The surface is covered with an insulating glass or plastic cover. When a finger approaches the surface, electrostatic capacity among multiple electrodes changes simultaneously, and the position where contact occurs can be identified precisely by measuring the ratios between these electrical currents.

A unique characteristic of a projected capacitive touch panel is the fact that the large number of electrodes enables accurate detection of contact at multiple points (multi-touch). Smaller projected capacitive multi-touch panels, such as those found in smart phones and tablets, are made with indium-tin-oxide. However, the methods used to make small panels are poorly suited for use in large screens, since increased screen size results in a slower transmission of electrical currents across the panel, increasing the amount of error and noise in detecting the points touched. Instead, larger touch panels use center-wire projected capacitive touch panels in which very thin electrical wires are laid out in a grid as a transparent electrode layer. While lower resistance makes center-wire projected capacitive touch panels highly sensitive, they are less suited to mass production than ITO etching.

3.2.4 Summary of Input Technology

In Table 3.2.4, we provide an overview of the touch technologies discussed in the previous sections. In regards to a pure 2-D sketching interface, comparing the capabilities shows a resistive panel would likely be the best input device. It's high accuracy and resolution are very desirable traits for creating high quality, accurate sketches. However, our system intends to use multi-touch gestures to navigate the three-dimensional sketching environment. Therefore, we require multi-touch capabilities that a resistive panel lacks. As a result, we design our application around capacitive displays, as they have the best touch functionality, and their poor stylus support is handles by the use of active pen technology

Method	Resistive	Capacitive	SAW
Light Transmittance	Poor	Good	Good
Finger Touch	Excellent	Excellent	Excellent
Gloved Touch	Excellent	None	Good
Stylus Touch	Excellent	Poor	Good
Maximum Single User Touch Points	One	Ten	Two
Accuracy	Excellent	Good	Good
Durability	Poor	Excellent	Excellent
Water Resistance	Excellent	Excellent	Poor
Cost	Reasonable	Not reasonable	Not reasonable

Table 3.1: A comparison of touch technology

specially designed for capacitive displays. In particular, we use Microsoft's Perspective Pixel display, soon to be re-branded as the Surface Hub, which has basic pen capability, upwards of 4K resolution, and support for up to a hundred touch points.

3.3 Advanced User Interaction: Input using Gesture

A gesture is a form of communication where visible body action communicate particular messages. Common gestures are usually performed by hand and arm movements. Other forms of physical non-verbal communication, such as purely expressive display, proxemics, and joint attention differ from gestures, which communicate specific messages. While some gestures are ubiquitous, such as pointing, which differs little in intent from one application to another, many do not have universal meanings and are defined differently in different disciplines.

3.3.1 Types of Gestures

In *Gestures* [6], Morris describes two main types of gestures: primary and incidental.

Primary gestures are voluntary movements that a person uses with intent of communicating a message. There are three main types:

1. Emblems: These are gestures that have a direct verbal equivalent. For example, a waving of a hand upon an encounter means hello. Emblems tend to form in situations where speech is challenging or impossible. For example, airport controllers on runways communicate with gestures because the planes make it impossible to hear.
2. Illustrators: These gestures are closely linked with speech, and serve to clarify, or add to the content of the message. Illustrators are made by hand movements. A common example of an illustrator is pointing.
3. Reinforcers: There are gestures that help regulate the flow of conversation. For example, a head nod during conversation can mean that the current speaker should continue, or an upwards point might mean to wait to continue speaking.

Secondary, or incidental gestures are unintentional, but despite their lack of direct message are still important in conversation. Gestures such as grooming the hair, fidgeting, looking down or away, or looking at a clock are all examples of involuntary gestures. While they do not directly communicate, secondary gestures can still send information about the current state of their user. This is called leakage, when true feelings or attitudes are revealed despite what the overt signals are communicating. For example, a man in a rush might say, ”Yeah, I can talk” while looking at a watch. Secondary gestures are not as important for Human Computer Interaction, however care must be taken so involuntary gestures are accidentally used as input.

Human computer interaction further distinguishes between types of gestures, splitting them into two major categories.

1. Offline: These gestures are processed after the user interaction with the object. For example, drawing a circle activates a menu.
2. Online: These gestures directly manipulate an object. A common example is taking two fingers and spreading them while touching an object to zoom in on the object.

3.3.2 Multi Touch Gestures

Multi-touch gestures are predefined motions used to interact with multi-touch devices. Many modern consumer electronics like smart-phones, tablets, laptops, or desktop computers feature functions triggered by multi-touch gestures. They tend to be direct input methods with simple to understand functionality heavily based on the gesture used, allowing non-technical people to quickly configure and navigate multi-touch applications. This section will only discuss core multi-touch gestures, and not the many subsets of unique gestures that combinations of this set of core gestures can create. Visual representations of how these gestures are performed can be seen in Figure 3.5

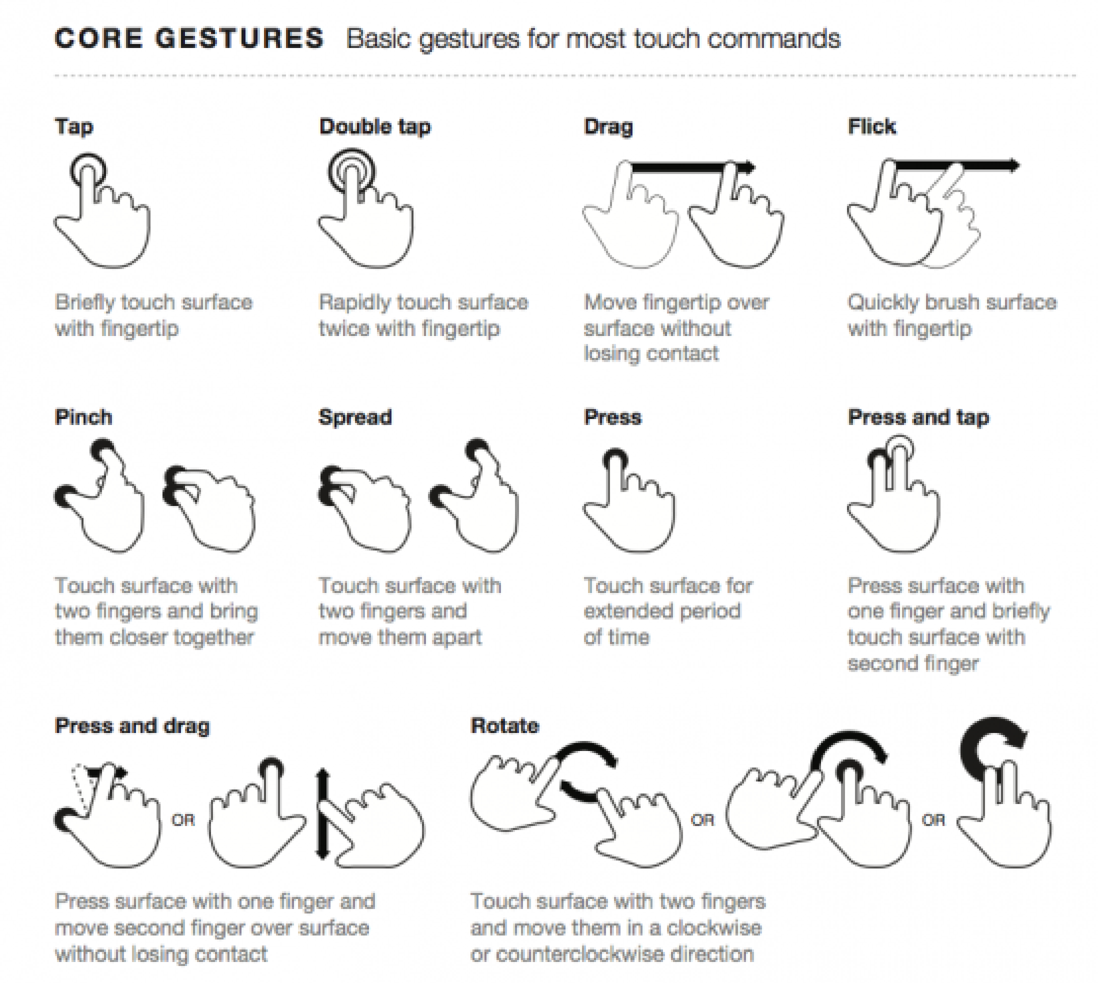


Figure 3.5: A visual representation of the core set of touch gestures. [7]

Tap

Taps are performed by quickly pressing and releasing a screen with a fingertip. Most systems differentiate between single and double taps, preformed by tapping the screen once or twice respectively. Taps are generally used for selecting items, with single and double taps offering different types of selection. Taps are both online and offline gestures, since the number of taps need to be processed, but directly interact with objects being tapped.

Drag

Drags are performed by moving a fingertip over a surface without losing contact. An alternative type of drag is the flick, which is performed by the same method as the drag, only faster. Drags are generally used to move elements, making them online gestures.

Pinch and Spread

Pinches are preformed by taking any number of fingertips and enclosing them towards a point. Spread gestures are the inverse action. Generally these online gestures are used for magnification.

Rotate

Rotate gestures are performed by moving two or more fingertips in a circular pattern around a point. The rotation point is used as the input location. Rotation is an online gesture.

Press

Presses are performed by touching a screen for an extended period of time. Presses can be combined with other touch gestures to allow for a deeper level of user interaction. Presses are offline gestures, since the duration of the press must be processed. This action usually results in context menus which are used for enhancing information about the object.

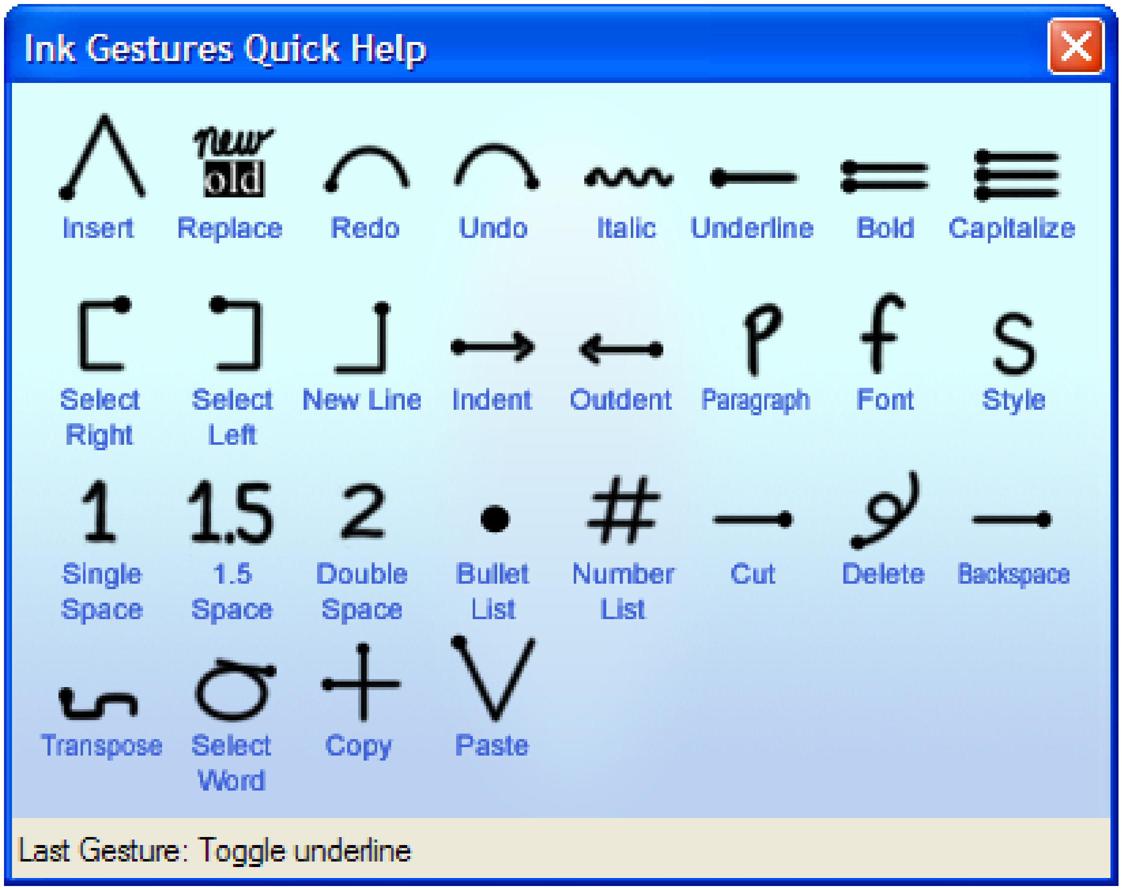


Figure 3.6: Examples of pen gestures and their associated functions. [2]

3.3.3 Pen Gestures

Pen gestures recognize certain shapes as not handwriting, but as an indicator of a special command. For example, a pig-tail shape (used often as a proofreader's mark) would indicate a delete operation. Depending on the implementation, what is deleted might be the object or text where the mark was made, or the stylus can be used as a pointing device to select what it is that should be deleted. These types of gestures are offline. Figure 3.6 shows an example of a variety of pen gestures and their associated commands inside of an application.

Pens can be used to perform many of the multi-touch gestures, but the pen is just treated as a finger in these scenarios. In order for this project to incorporate pen gestures, the pen input would need to be preprocessed before it is projected into 3-D space. While pen gestures are a useful tool, in a free form sketching environment it can be unclear if the user intends to make a stroke or a gesture, even if using the same symbol. This can potentially confuse the user, while we can use touch gestures to accomplish similar results. For this reason, pen gestures are not incorporated into this application.

3.3.4 Three Dimensional Gesture Recognition

While older forms of gesture recognition attempt to translate physical interaction with an input device in oder to form gesture based commands, more modern approaches attempt to do away with the intermediary and directly interpret motions of the human body. This is accomplished using computer vision techniques, as well as different types of cameras and senors used to capture and understand a three dimensional environment. Once this information is captured a variety of techniques can be used to analyze the scene and detect gestural information. While 3-D gesture recognition is an emerging area, and is beginning to see common use in a number of modern user interfaces, it's use for 3-D sketching is left to future work. Possible applications are using 3-D gestures to manipulate the environment in conjunction with 2-D gestures.

3.4 Summary

In this chapter we discussed how users interact with technology, an overview of common and project related input devices, and why certain interaction methods are preferable for 3-D sketching. For our application, we chose to support a subset of these techniques and devices in order to mimic real world sketching as closely as possible. We use a capacitive touch display, which supports a very large number of touch input points as well as an active pen device. From the active pen, we leverage the capability to detect sub-pixel input position, as well as the ability to detect how hard the user presses on the screen with the pen.

CHAPTER 4

CREATING A SKETCH

The most basic operation in any sketch-based modeling system is, of course, obtaining a sketch from the user. The key characteristic of a sketch-capable input device is that it allows freehand input. While a mouse is capable of this form of input, devices that more closely mimic the feel of freehand drawing using a pen, such as a digitizing tablet, are better for users to maximize their ability to draw. Devices that are both an input device and a display are particularly suited to this, because it most closely mimics traditional artist creation methods by allowing direct interaction with the sketch space, as opposed to tablets where the space between the input device and the display surface is relative.

Pencil and paper is a rich communication method. Artists convey information not only with the shape of the completed objects in the sketch, but also through more subtle methods such as pressure and stroke style. Small details can relay information about the object, such as important details; heavier lines or many small strokes usually show more focus was put in a particular area.

4.1 Representation of a Sketch

At the bare minimum, an input device should provide positional information in some two dimensional coordinate system, usually one based on the interaction window. For sketch input, the representation must at least approximate continuous movement. Sampling rates vary from one device to the next. The samples themselves may also be spaced irregularly, with sample points closer as users draw slowly or carefully, or father apart if the user draws quickly.

Figure 4.1: An example of a sketch

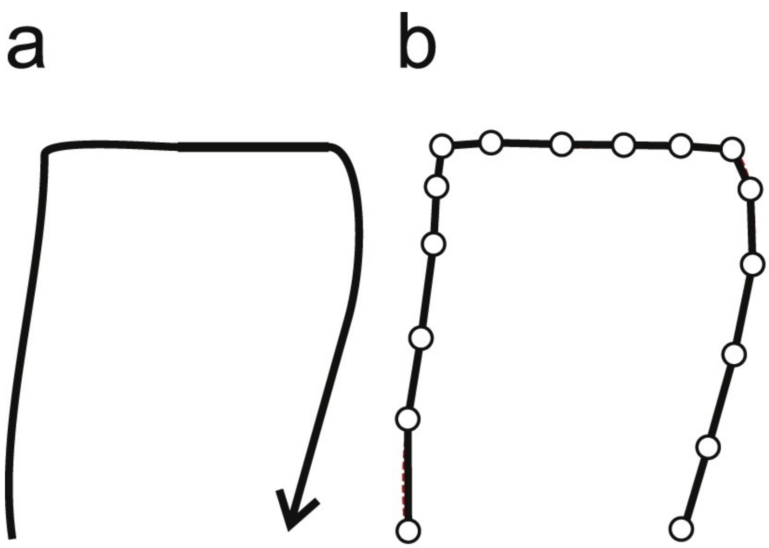


Figure 4.2: An input stroke (a) is provided to the application as (b) a sequence of point samples.

We will refer to this sampled sequence of points as a stroke. Strokes are stored as a list of points, objects containing coordinates from the sample space, sorted by time. A sketch is comprised of a large number of these stroke objects.

4.2 Spline Curves

The input data represents an approximation of the stroke input from the user that is dependent on the sample resolution of the input device, and by extension the resolution of the display. While this would work decently well assuming we are working with a static raster image, three dimensional sketching allows the user to move the camera. Moving the camera will eventually result in poor quality sketches from the stored stroke data lacking sub-sample rate information that accurately reflects the intent of the original stroke. Please see Figure for an example of this. To remedy this, a mathematical representation of the curve is needed such that the ”sampling rate” of the stroke is independent from the resolution of the input device. In computer graphics, this is commonly accomplished with spline curves.

A spline is a collection of polynomial segments. These segments can be linear, cubic, or any degree polynomial function. Splines are a common solution for modeling smooth curves from a small number of points. For this project, we use a Bzier curve function for the spline pieces. A Bzier curve is a parametric curve commonly used in computer graphics to model infinitely scaling, smooth curves. The curve is defined by control points P0, P1, ..., Pn, and is explicitly evaluated as follows:

$$
\begin{aligned}
\mathbf{B}(t) &= \sum_{i=0}^{n} \binom{n}{i} (1-t)^{n-i} t^i \mathbf{P}_i &(4.1)\\
&= (1-t)^n \mathbf{P}_0 + \binom{n}{1} (1-t)^{n-1} t \mathbf{P}_1 + \cdots &(4.2)\\
&\quad \cdots + \binom{n}{n-1} (1-t) t^{n-1} \mathbf{P}_{n-1} + t^n \mathbf{P}_n, \quad 0 \le t \le 1 &(4.3)
\end{aligned}
$$

where $\binom{n}{i}$ are the binomial coefficients, defined as

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \quad \text{for all integers } n, k : 1 \le k \le n-1 \tag{4.4}$$

with initial values

$$\binom{n}{0} = \binom{n}{n} = 1 \quad \text{for all integers } n \ge 0 \tag{4.5}$$

The curve can also be evaluated recursively, by

$$\mathbf{B}_{\mathbf{P}_0}(t) = \mathbf{P}_0 \tag{4.6}$$

$$\mathbf{B}(t) = \mathbf{B}_{\mathbf{P}_0\mathbf{P}_1\ldots\mathbf{P}_n}(t) = (1-t)\mathbf{B}_{\mathbf{P}_0\mathbf{P}_1\ldots\mathbf{P}_{n-1}}(t) + t\mathbf{B}_{\mathbf{P}_1\mathbf{P}_2\ldots\mathbf{P}_n}(t) \tag{4.7}$$

What this equation means is that a Bezier spline of order n can be defined by linear interpolation between two splines of order $n-1$. For this project, we will use a cubic Bzier function for the spline segments.

We must now determine a method for going from our sample data to a spline curve. A naive approach would be to generate a spline curve that passes through the all of our control points in the order they were generated. Any series of any four distinct points can easily be converted to a cubic Bzier curve that goes through all four points in order. While this method guarantees that the generated curve passes through all of the input control points, it is not guaranteed to generate a curve that represents the intent of the original stroke.

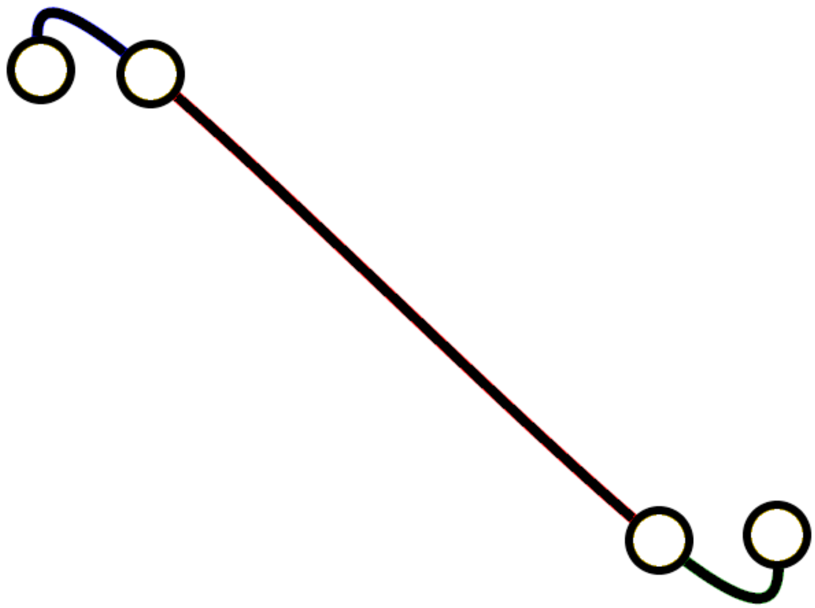


Figure 4.3: An example of an artifact in the spline generation caused by oddly spaced sample points.

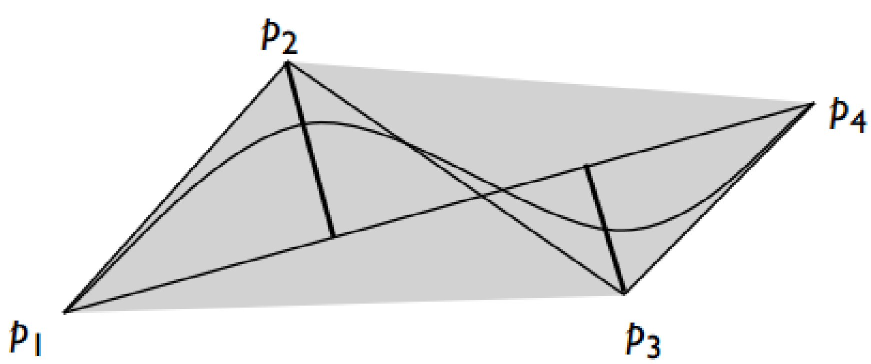


Figure 4.4: Using the distance from the control points for termination.

The approach we take is using the input samples as control points to generate a piecewise spline curve. This method is better at dealing with extreme changes in sample rates because the produced curve contains less convex and concave changes. However, it is likely that the produced curve will still be inaccurate. The curve generated will also not pass through the sampled points. However, if the sample rate is high, it will be close.

In practice, not generating a curve that is exactly the same as the input sample points is not as necessary as one would think. The most important part is the curve should match the form of the original curve. If the detail of the curve is important, then the user will probably draw more slowly, allowing for a greater number of sample points, and generating a spline that closely resembles the shape of the detailed section. Also, matching control points perfectly is a poor decision when working with pixel grids, as extremely detailed sampling produces input in a staircase pattern. Matching input exactly would make diagonal straight lines impossible. The simpler and more consistent shape of the recursive B-spline curve makes it a better choice than the inverse control point calculation, as seen in practice by a large number of applications implementing spline approximations, including Mischief.

4.2.1 Algorithm

We begin with a list of control points, numbered from 0 to N-1. Segment i of the curve is influenced by control points i-1, i, i+1, and i+2. Using this definition we can generate N-3 segments from the N control points without falling off the ends of the sequence.

Since we can't render a spline, we need to approximate the curve by subdividing it into small line segments. We're going to divide each Bezier curve into a set of connected linear components, with the intent that a large enough number will look sufficiently smooth. Subdivision of these segments occurs as follows:

1. Begin with points $\{P_0, P_1, P_2, P_3\}$, which define a Bzier curve, and a number $u, 0 \leq u \leq 1$
2. Define $L_1 = (P_0 + P_1) * u$, $H = (P_1 + P_2) * u$, and $R_2 = (P_2 + P_3) * u$
3. Define $L_2 = (L_1 + H) * u$, and $R_1 = (H + R_2) * u$
4. Define $M = (L_2 + R_1) * u$
5. Create two new Bzier curves using $\{P_0, L_1, L_2, M\}$ and $\{M, R_1, R_2, P_3\}$
6. Repeat steps 2 through 5 until a termination criteria is met. Possible criteria include distance between control points, and distance between control points P_1 and P_2 and the line between P_0 and P_3.

If the termination criteria is small enough, then the curve will appear very smooth, even when zooming in to the curve. This algorithm produces a smooth B-spline curve starting at control point 0 and ending at N-1.

4.3 Summary

In this section, we described how we turn user input over a set of pixel coordinates into a curve approximating the intent of the stroke. We believe intent is more important than perfect accuracy since in practice, replicating curves that perfectly match input data results in poor quality curves. This is because of the finite resolution of a pixel grid, where as in real world sketching, the concept of 'input resolution' does not exist. Using a B-spline algorithm with cubic Bezier components, we can calculate a smooth spline curve for our stroke input.

CHAPTER 5

SKETCHING IN 3D

In a 2-D sketching application on a graphical tablet, a user draws on a two dimensional plane that mirrors the surface of the input device. However, in a 3-D sketching application, the user draws curves on arbitrary three dimensional planes and objects. The most basic example is a simple plane: a user can orient a plane in three dimensional space, and the draw on the tablet. The strokes on the tablet are then projected onto the plane, thereby creating a 3-D stroke.

A similar technique exists in rendering algorithms in computer graphics, called ray tracing. Ray tracing is primarily used for the creation of realistic images; it determines visible surfaces at the pixel level. Starting at points on the lens of a virtual camera, rays are sent into an environment until they intersect with geometry. In between the environment and the camera is a virtual image plane, a grid representing the pixels of the final image. Where a ray intersects with the image plane determines what pixel the ray effects. A subset of this algorithm can be used to solve for the stroke projection onto a virtual drawing object, the ray casting algorithm.

5.1 Ray Casting

Ray casting is the use of rays, a line with an endpoint and a direction, to test for intersections with surface geometry. These computations are the fundamentals of ray tracing computer graphics algorithms, used to solve a variety of problems.

A common use of ray casting is object selection in an interactive 3-D appli-

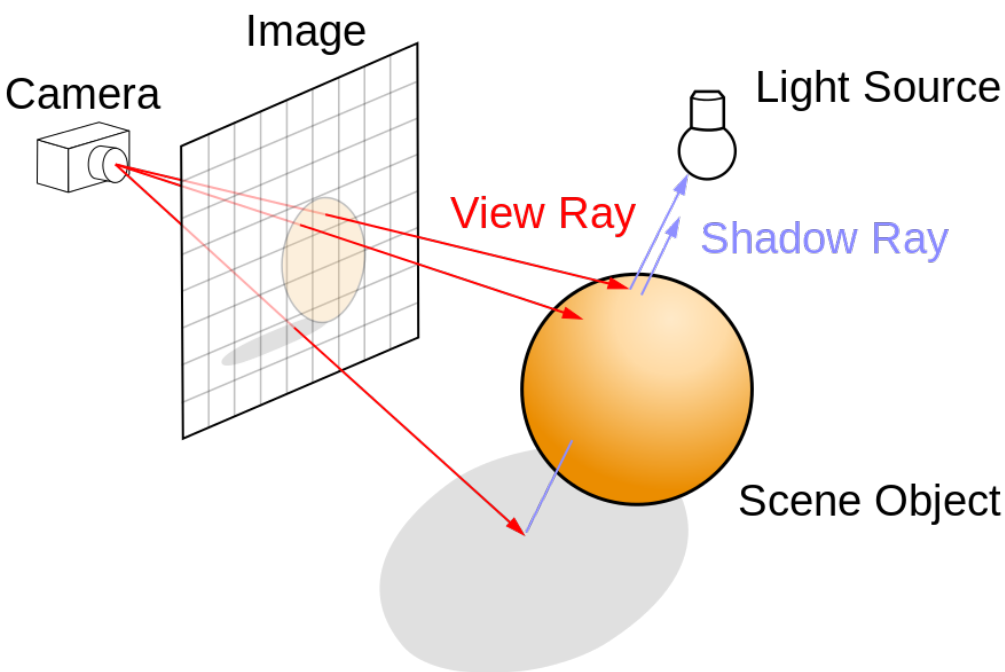


Figure 5.1: Ray Tracing in Computer Graphics

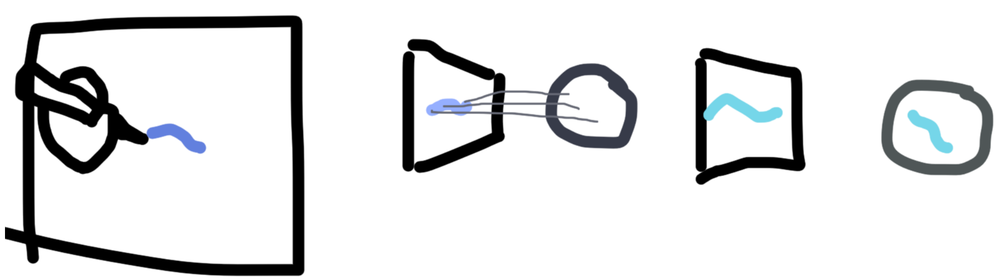


Figure 5.2: Rough Diagram of Projecting Stroke onto Drawing Surface

cation. The mouse is clicked in pixel (i, j), and the "picked" object is whatever is "seen" though the pixel. Using ray casting, a ray is created at pixel (i, j) and sent in the virtual viewing direction. If it intersects, then the object at the point of intersection is selected. Through this example, we can see a method of using ray casting in order to project a stroke into the virtual space. The ray casting algorithm is utilized as follows:

- The user draws a stroke on a 2-D plane, giving a set of sample points
- An imaginary plane is created in the virtual space representing the image plane.
- Rays are cast into the three dimensional scene from the imaginary plane based on the sample positions.
- The intersections with scene geometry represent the new 3-D positions of the stroke

Once we intersect all of the sample points with the scene geometry, we can use the spline techniques described in the previous chapter to create a 3-D curve approximating the appearance of the projected 2D stroke.

5.1.1 Generating the Ray

A ray is mathematically defined as an origin point and a movement direction, usually represented as a parametric line. The ray is generated in the same way view rays are created in ray tracing; assuming we have an eye $\mathbf{e}$ and a point on the screen plane $\mathbf{s}$, the line is defined by:

$$p(t) = \mathbf{e} + t(\mathbf{s} - \mathbf{e}) \tag{5.1}$$

Given a value t, we can determine any point that lies along the line, with $p(0) = \mathbf{e}$ and $p(1) = \mathbf{s}$. For positive values of t, if $t_1 < t_2$, then t_1 is closer to the eye than t_2, and if $t < 0$, then the point is behind the ray.

Computing a ray means solving for s using the relative coordinate axis of the camera, the aspect ratio of the display, and a pixel position (i, j). s is calculated as follows:

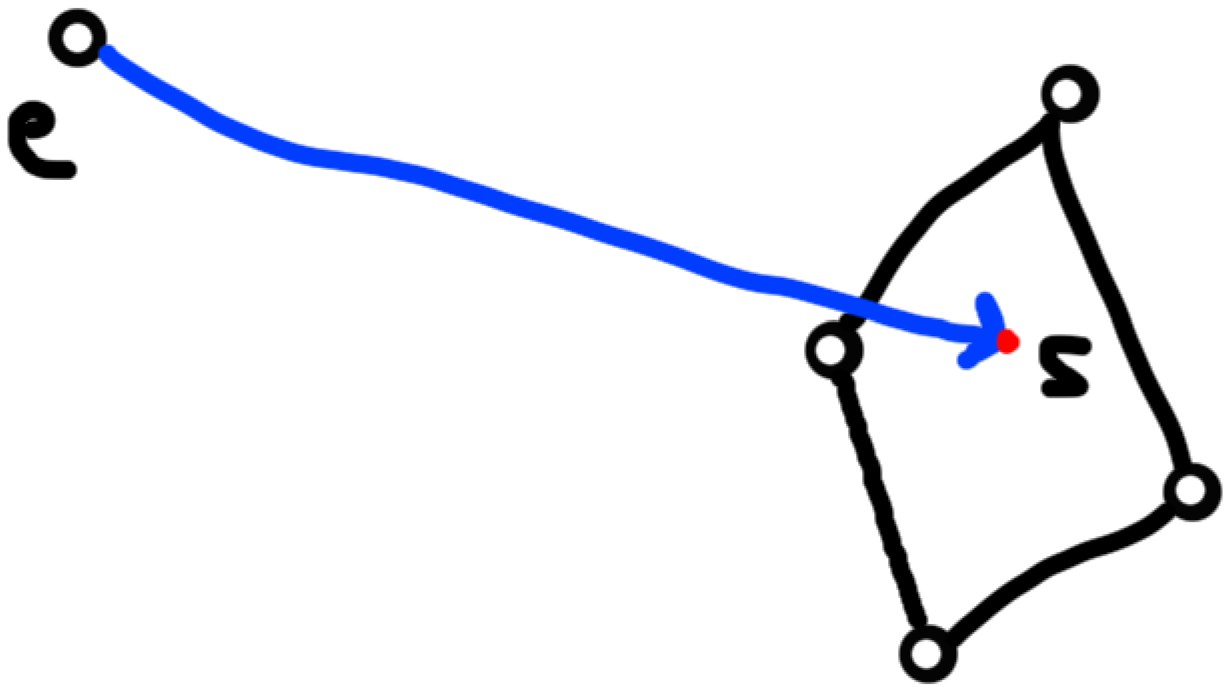


Figure 5.3: The ray from the eye **e** to a point on the screen s

- Moving from the eye to the view plane by using a defined distance for the near plane (w_s) and the forward view direction (**w**) of the camera
- Calculate the a $u_s v_s$ coordinate on the view plane based on the pixel (i, j).
- Move u_s units in the camera's right direction (**u**), and v_s units in the up direction (**v**).

This process can be expressed as:

$$\mathbf{s} = \mathbf{e} + u_s\mathbf{u} + v_s\mathbf{v} + w_s\mathbf{w}. \tag{5.2}$$

5.1.2 Ray-Triangle Intersection

Given a ray $\mathbf{e} + t\mathbf{d}$, where $\mathbf{e}$ is the origin of the ray and $\mathbf{d}$ is the direction, we want to find the first intersection with any object where $t > 0$. In this section, the intersection the most basic computer graphics primitive, the triangle, is discussed. Any object in a real-time computer graphics application is constructed or approximated using triangles, including curved surfaces such as spheres. Thus, being able to intersect with a single triangle will allow 3D sketching on any object used for the application.

For our algorithm, we will intersect a ray with a parametric plane that contains the triangle. Once intersected, we use barycentric coordinates to check if the intersection point is contained within the boundaries of the triangle. Note we could eliminate this check if we want to draw on an infinite intersection plane. This method requires only the vertices of the triangle.

To intersect a ray with a parametric surface, a system of equations is created where the Cartesian coordinates all match:

$$x_o + tx_d = f(u, v) \tag{5.3}$$

$$y_o + ty_d = g(u, v) \tag{5.4}$$

$$z_o + tz_d = h(u, v) \tag{5.5}$$

Here there are three equations and three unknowns (t, u, and v). When a parametric surface is a parametric plane, the parametric equation can be written in vector form. If the vertices of the triangle are $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$, then the intersection occurs when

$$\mathbf{e} + t\mathbf{d} = \mathbf{a} + \beta(\mathbf{c} - \mathbf{a}) + \gamma(\mathbf{c} - \mathbf{a}). \tag{5.6}$$

β and γ are two of the three barycentric coordinates of the triangle. If $\beta > 0, \gamma > 0$, and $\beta + \gamma < 1$, then the intersection point lies inside of the triangle; otherwise it hits the plane outside the triangle. If there are no solutions, then either the triangle is degenerate or the ray is parallel to the parametric plane.

To solve for $t, \beta,$ andγ, equation 1.4 is expanded from the vector form to equations for each of the three coordinate planes.

$$x_o + tx_d = x_a + \beta(x_b - x_a) + \gamma(x_c - x_a) \tag{5.7}$$

$$y_o + ty_d = y_a + \beta(y_b - y_a) + \gamma(y_c - y_a) \tag{5.8}$$

$$z_o + tz_d = z_a + \beta(z_b - z_a) + \gamma(z_c - z_a) \tag{5.9}$$

This can be rewritten into a standard linear equation of the form $Ax = b$:

$$\begin{bmatrix} x_a - x_b & x_a - x_c & x_d \\ y_a - y_b & y_a - y_c & y_d \\ z_a - z_b & z_a - z_c & z_d \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} x_a - x_o \\ y_a - y_o \\ z_a - z_o \end{bmatrix} \tag{5.10}$$

This can be solved using Cramer's rule: given a system of n linear equations for n unknowns, represented as $Ax = b$, where the $n \times n$ matrix A has a nonzero determinant, and the vector $x = (x_1, \ldots, x_n)^{\mathrm{T}}$ is the column vector of the variables, the system has a unique solution. This solution is given by

$$x_i = \frac{\det(A_i)}{\det(A)} \qquad i = 1, \ldots, n \tag{5.11}$$

where A_i is the matrix formed by replacing the i-th column of A by the column vector b. Solving gives the solutions:

$$\beta = \frac{\begin{vmatrix} x_a - x_o & x_a - x_c & x_d \\ y_a - y_o & y_a - y_c & y_d \\ z_a - z_o & z_a - z_c & z_d \end{vmatrix}}{|A|} \tag{5.12}$$

$$\gamma = \frac{\begin{vmatrix} x_a - x_b & x_a - x_o & x_d \\ y_a - y_b & y_a - y_o & y_d \\ z_a - z_b & z_a - z_o & z_d \end{vmatrix}}{|A|} \tag{5.13}$$

$$t = \frac{\begin{vmatrix} x_a - x_b & x_a - x_c & x_a - x_o \\ y_a - y_b & y_a - y_c & y_a - y_o \\ z_a - z_b & z_a - z_c & z_a - z_o \end{vmatrix}}{|A|} \tag{5.14}$$

where A is given in equation 1.8, and $|A|$ denotes the determinant of A.

5.2 Acceleration Structures

An acceleration structure must be used in order to give sub-linear time for ray object intersection for complex objects. In a naive implementation, the ray caster would iterate over all triangles in the scene to check for intersections, giving $O(N)$ performance. For sufficiently large values of N, this would be very slow, thus a "divide and conquer" algorithmic approach is used, creating an ordered data structure to speed up the intersection process.

While many approaches exist, this project implements a simple bounding volume hierarchy (BVH) tree (Rubin & Whitted, 1980).

5.2.1 Bounding Boxes

A bounding box for a point set S in N dimensions is the box with the smallest measure (area, volume, or hypervolume) within which all the points lie. For this project, we use axis-alligned bounding boxes (AABBs), which are constrained so their edges lie parallel to the Cartesian coordinate axis. A key operation in any intersection acceleration structure is computing the intersection of a ray and a bounding box. For this, it is not necessary to calculate where the ray hits the box, only if it is hit at all, as the box itself is a final piece of geometry.

The fastest method for intersecting a ray with an AABB is the slab method. The idea is to treat the bounding box as the space contained inside of N pairs of parallel planes. For each pair of planes, two pairs of t values, $t_m in$ and $t_m ax$ are solved for the segment that is between the two planes. If the largest $t_m in$ is smaller than the smallest $t_m ax$, then some portion of the ray is contained within all three planes.

5.2.2 BVH Tree

A bounding volume hierarchy (BVH) is a tree data structure on a set of geometric objects. All objects are wrapped in bounding volumes that form the leaf nodes of the tree. These nodes are then grouped into small sets and enclosed within larger bounding volumes. This grouping occurs recursively until there is a single bounding volume at the top of the tree. By using this data structure, the complexity of the a ray cast on a complex scene can be reduced from linear based on the number of objects to logarithmic.

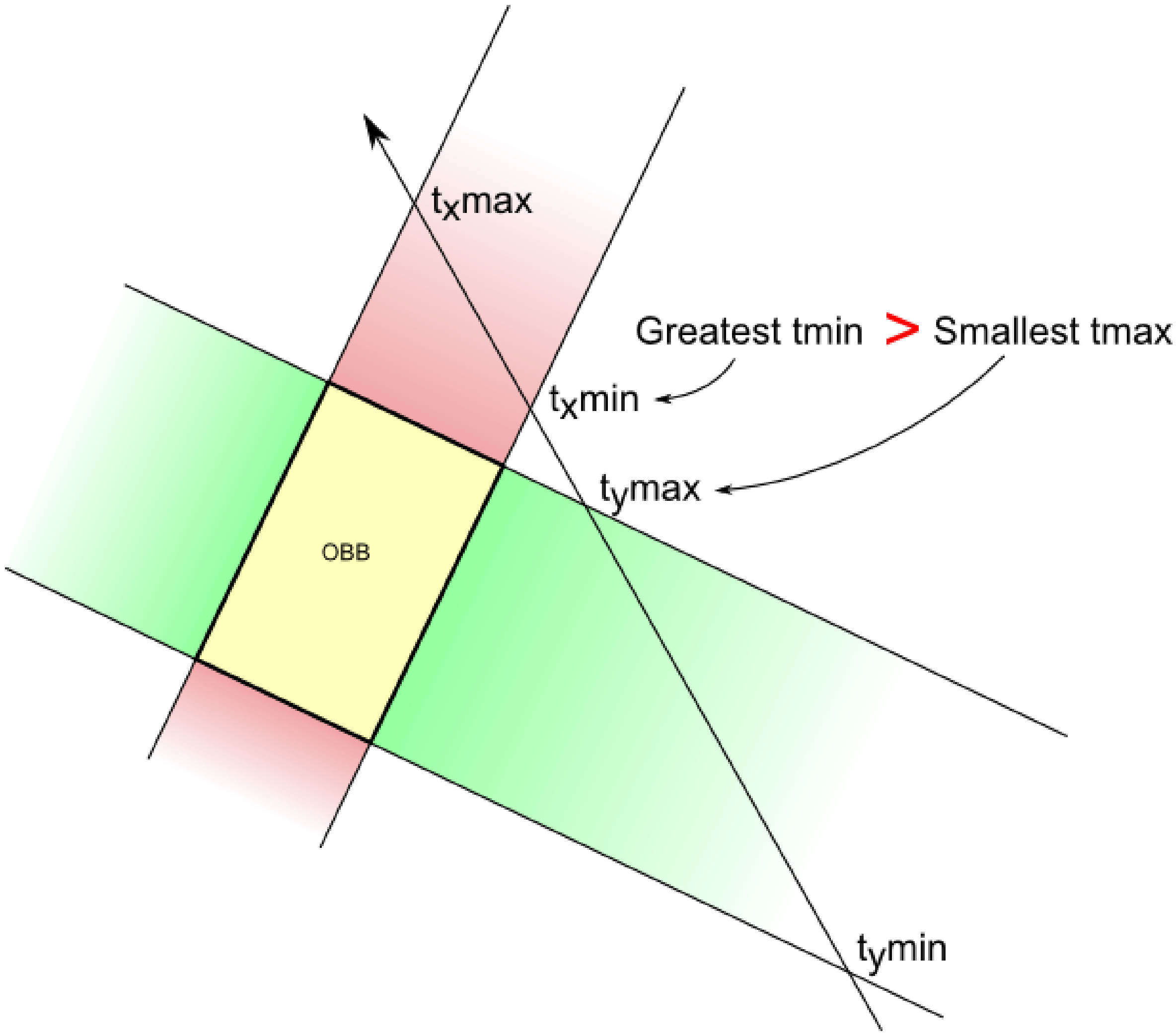


Figure 5.4: An example of a ray checking and failing for intersection with a box

A good BVH tree has the following properties:

- The nodes in a given sub tree should be close to each other. The lower down the tree, the closer the nodes should be.
- Each node in the BVH should be of minimum volume.
- The sum of all bounding volumes should be minimized.
- The volume of overlap of sibling nodes should be minimized.

- The BVH should be balanced with respect to both its node structure and its content. Balancing allows as much of the BVH to be pruned when not traversed.

Because of the number of properties a good BVH tree has, a good algorithm must be used for constructing the tree. The current bast algorithm is the Surface Area Heuristic (SAH). The idea behind SAH is to balance both the number of objects contained in a bounding volume while minimizing the surface area of the volume itself. This is an exceptionally slow heuristic, so while providing the fastest solution at runtime, the construction cost must be considered for use in real time applications.

Basic BVH Algorithm

1. Decide on a cost function that will check the quality of your object division
2. Create Bounding Box for current Node.
3. Sort along all 3 axis and find the axis with the best cost, and find the object position that will be your center.
4. Divide the objects.

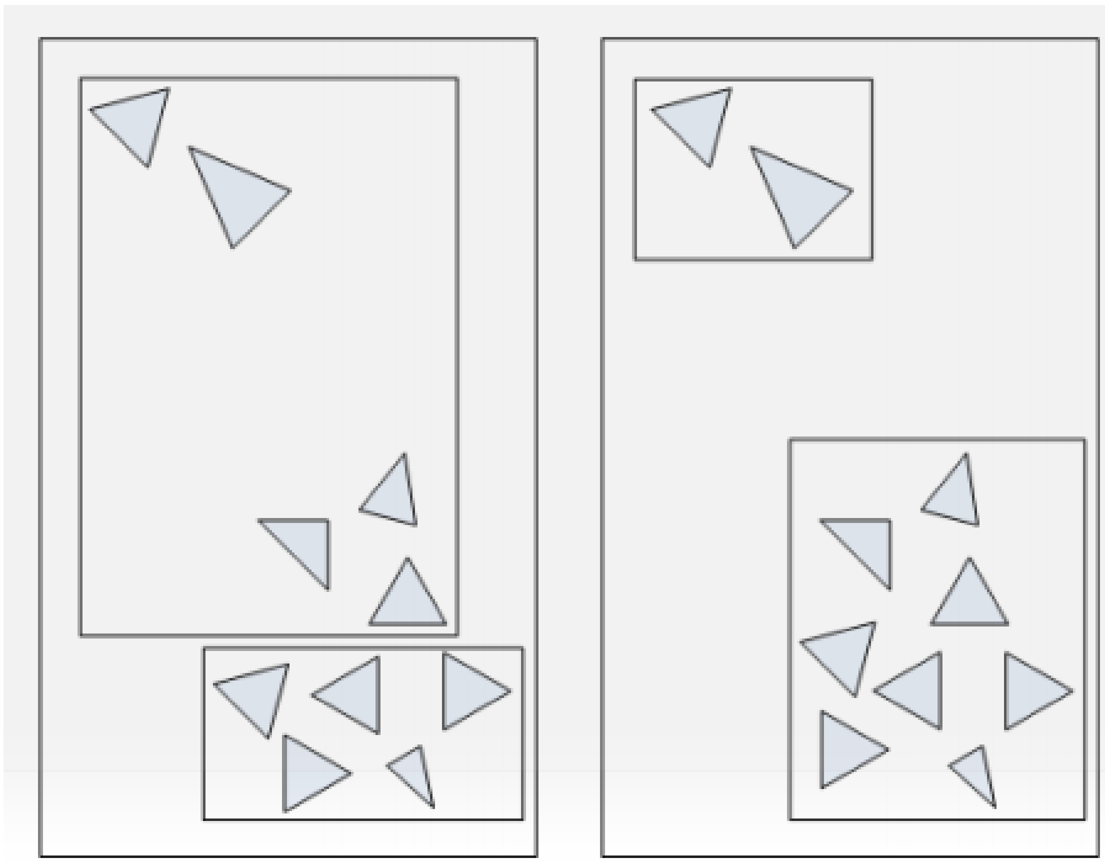


Figure 5.5: A comparison between the bounding volumes of a node using a naive implementation (left) and a surface area heuristic (right).

CHAPTER 6

LINE RENDERING

As a sketching application, it is important for curves to be rendered with high quality. While graphics hardware has support for line rendering, it's implementation is only suitable for line segments, with significant issues when trying to render curves. In this chapter, we discuss the system this project implements for curve rendering.

6.1 Problems With Available Line Renderers

OpenGL has support for a variety of line primitives: lines, line strips, and lines with adjacency data. However, these primitives are not suited for rendering curves at high quality. This is because of the method OpenGL uses to render line segments: as a rectangle between the two segment points. While this method is suited for common uses of lines in 3D applications, for example wire frames, it does not extend to curve rendering because of the gaps that appear between segments. It is reasonable to think that if we use a curve that is subdivided to a fine enough degree, then the holes will be so small that they would not be visible. However, in practice, even finely subdivided lines have visible holes around any curve.

To correct the holes, this project implements a custom line renderer that constructs a triangle mesh from the line definition. Constructing a mesh for the line allow allows for implementation of a wide variety of stroke types, including strokes with various end caps or variable width based on stroke direction.

Additionally, OpenGL lines have inconsistent support for anti-aliasing

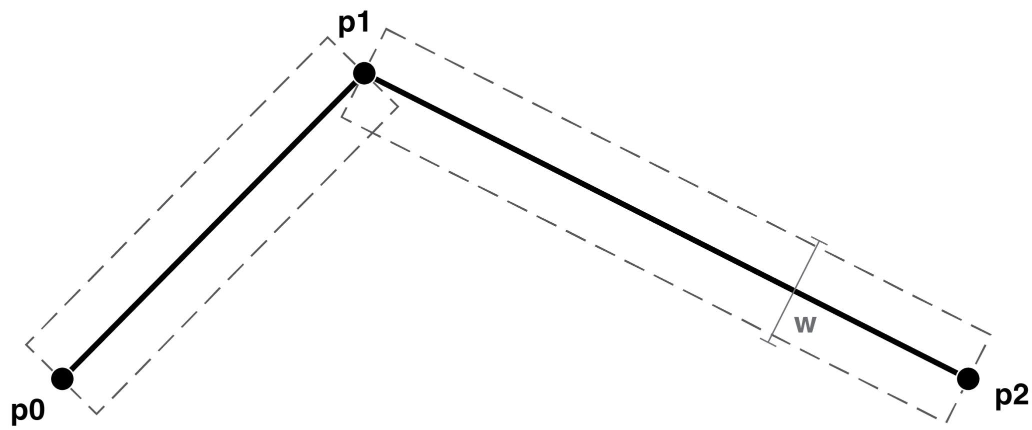


Figure 6.1: Two line segments connected together, as per default GL_LINES implementation. Note the gap from no join.

across all hardware. By switching to a triangle mesh, we leverage the native anti-aliasing support for triangles without worrying about consistency. As we can see in the above image, without anti-aliasing rendered lines appear jagged and poor quality. In order for our system to be appealing as a drafting table replacement, our system needs to render the highest quality lines possible.

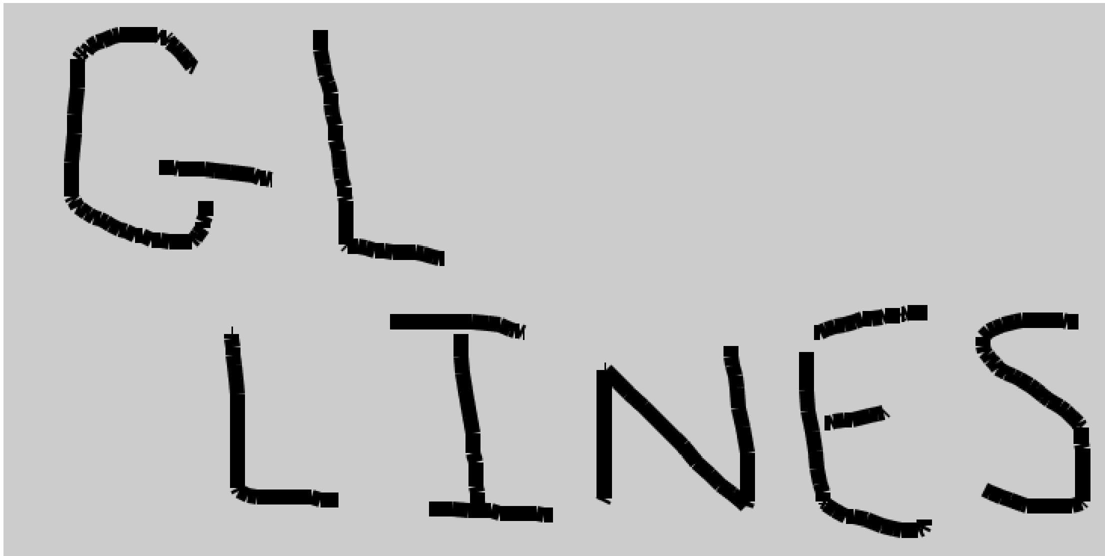


Figure 6.2: An example of the flaws of native line rendering in OpenGL using GL_LINES.

Symbol	Description
p_n	Ordered point on line in the range $[0, 2]$
n_{xy}	The normal to the line segment (p_x, p_y)
t_x	Vector from p_1 along n_{x1} with length $line\_width/2$

Table 6.1: Section Variables

6.2 Creating Joins From the Line Definition

In order to improve the appearance of our curves relative to the native OpenGL implementation, we will need to compute joins between each of our line segments. For this project, we implement three types of joins: miter, bevel, and round. Miter and bevel joins are used in conjunction with each other, while round joins are used alone.

A round join is formed by rounding out the gap such that a smooth curve is created. It is calculated by forming a circle whose center is located at p_1, with

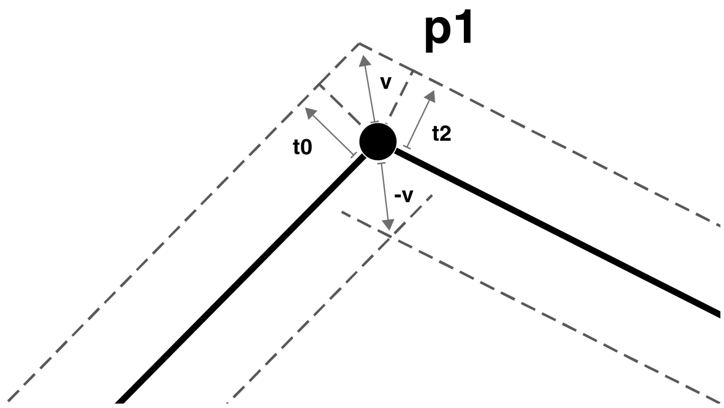


Figure 6.3: Diagram of a miter join with parameter labels

a radius of *line_width*/2. The total angle that needs to be filled can be calculated by $t_0 \cdot t_2$, and can be filled by rotating either t_0 or t_2 about p_1.

Miter joins are formed by extending the lines in the line geometry parallel to the original segment. The join is formed where these extended lines intersect, as can be seen in Figure 1.5. To calculate this, we take the line segments defined by $(p_0 + t_0, p_1 + t_0)$ and $(p_2 + t_2, p_1 + t_2)$ and compute their intersection. We call the vector from this intersection to p_1 v. We can then compute a polygon between p_1, $p_1 + t_0$, $p_1 + t_2$, and $p_1 + v$, which creates the miter join.

Miter joins have the particular issue that artifacts can arise from exceptionally sharp points in the line segments. We detect these cases by checking to see if v is larger than a certain length. If so, we use a bevel join instead. The bevel join is formed by simply creating a triangle between p_1, $p_1 + t_0$, and $p_1 + t_2$.

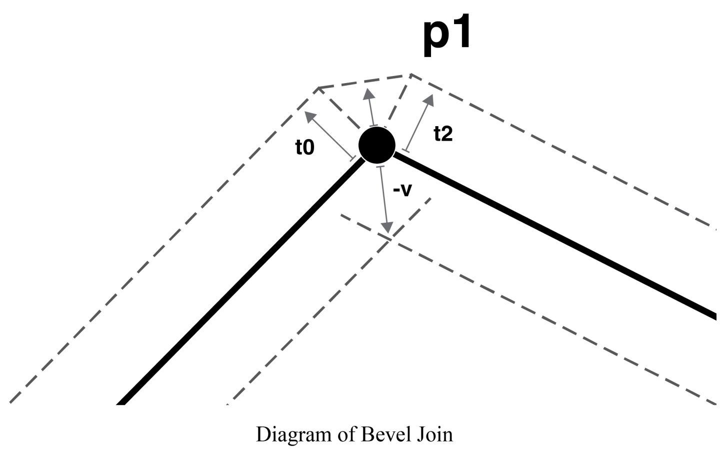


Figure 6.4: Diagram of a bevel join with parameter labels

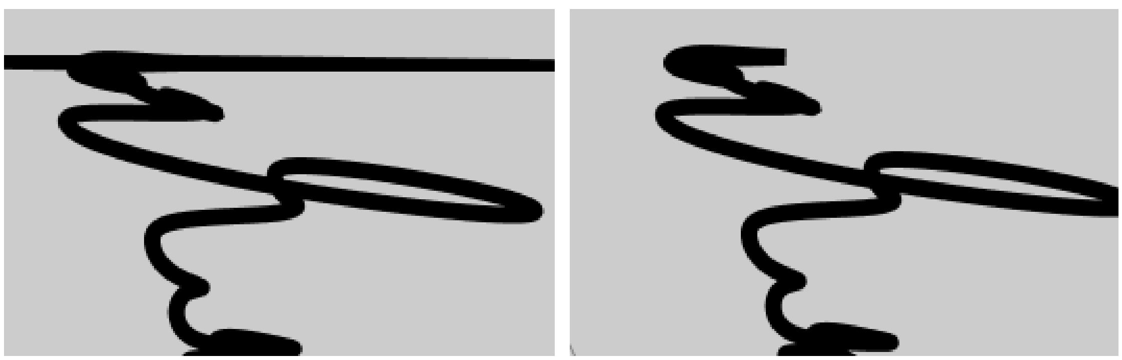


Figure 6.5: Left: An example of a miter join artifact from a sharp curve. Right: Artifact post-correction.

6.3 Implementation

Actually creating this geometry can get very expensive, especially as more and more lines are drawn. A simple polygonalization for a robust miter and bevel join implementation can create as many as six polygons for each pair of points on the curve. This does not take into account round joins, as smooth circle approximations require a large number of polygons. Combine this with spline

Figure 6.6: Left: A curve using default GL_LINES. Right: Implemented system

subdivision, and we run the risk of running into bandwith problems when transferring polygon data between the GPU and the CPU for very large strokes. To solve for this, we implement these algorithms in a geometry shader using a line adjacency data structure input. In order to reduce the number of branch operations, we polygonalize each line segment in halves, taking care of the join calculation for each side separately. An example of a polygonalized line segment can be seen in Figure 1.8.

The system can now display polygonalized, three dimensional lines and does so efficiently. However, we still need to decide how we are going to expand our lines. Approaches similar applications have taken include:

- Creating the line geometry on the screen and projecting it onto scene objects. With this approach, lines only exist on the plane they are drawn

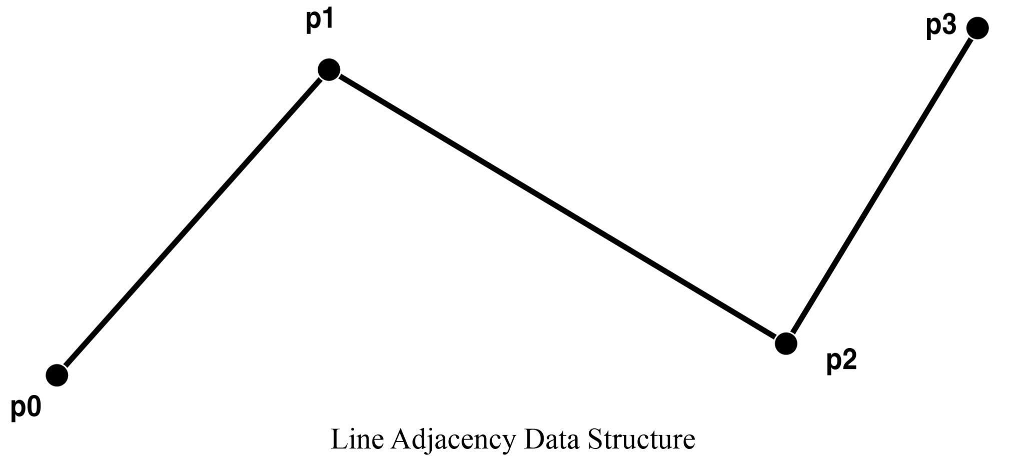


Figure 6.7: The line adjacency data structure. Each instance of the geometry shader polygonalizes the line segment (p1, p2)

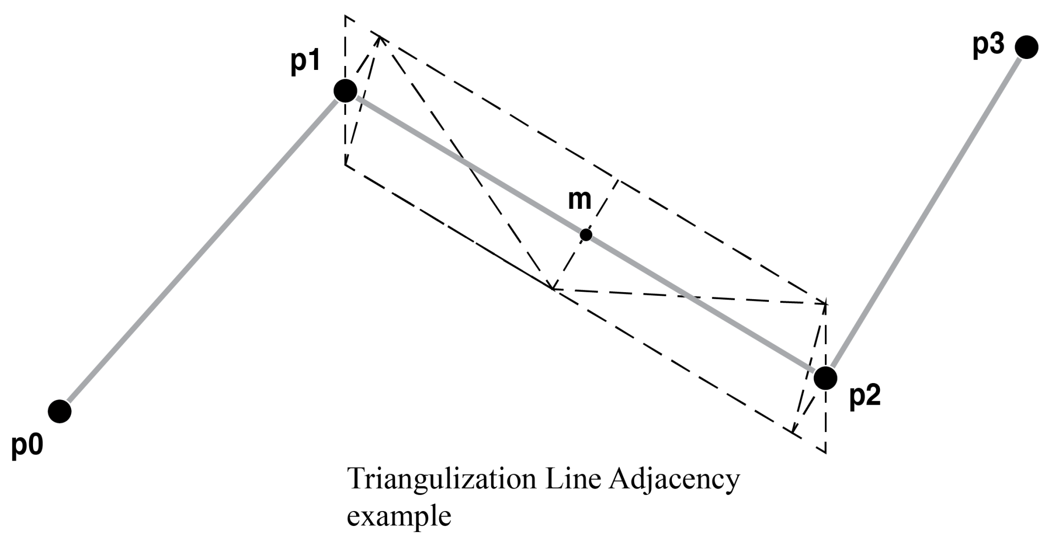


Figure 6.8: Example polygonization of a line with miter joins.

on. Because of this, it is possible to turn the camera and not see strokes the user has already mode. This approach is only suitable for applications where a base geometry is already supplied, and therefore not suitable for early stage design applications. Other shortcomings include the requirement of a large number of ray cast operations for each line created, which puts too much work on the CPU for a real time application, as well as the lines created lacking a sense of depth.

- Creating 3-D geometry from the line definition. This approach makes lines look poor when viewed from the side, as the line form will not have the same shape as originally intended. The approach also requires a high number of polygons.

We would like to allow for lines to be viewed from any direction, while still retaining their original line width and quality. The approach this application takes is to perform all of the calculations from creating line geometry in the distorted image space. By working in the distorted image space, we can guarantee the geometry is always expanded in the screen's x-y plane. Since points in the distorted space also contain a z value, our lines will have different widths depending on depth.

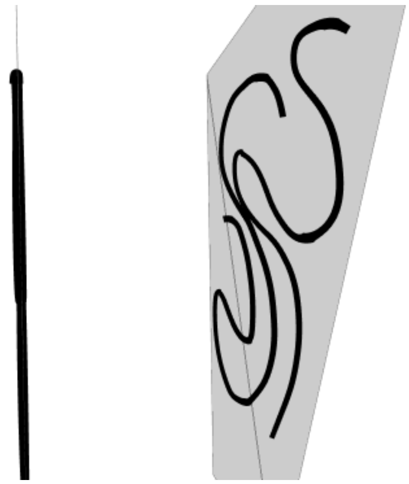


Figure 6.9: Even when rotating the drawing plane such that it is no longer visible, our lines remain visible.

Algorithm 1: Line Rendering Algorithm

1: **for** Line Adjacency data set $\{p_0, \cdots, p_3\}$ **do**
2: Covert world space line points $\{p_0, \cdots, p_3\}$ to the distorted image space points $\{p_{0s}, \cdots, p_{3s}\}$
3: Compute normals for each of the line segments $(p_{ns}, p_{(n+1)s})$
4: Enforce normals are pointing towards the outer arcs of segment pairs
5: Compute join parameters using image space variables and a defined line width
6: Triangulate for the line segment (p_1, p_2)
7: **end for**

In this chapter we have described a method for creating geometry from a line definition. We have also described how we create this 2-D geometry such that it is visible from any 3-D direction. Similar applications create the extra line geometry on the CPU. By creating our geometry exclusively on the GPU, we minimize the bandwidth needed to transfer data from the CPU to the GPU, allowing extremely detailed curves using our spline calculation.

CHAPTER 7

CONCLUSION

Summary of everything goes here. More images of the system in use and example use cases.

BIBLIOGRAPHY

[1] 3dsCATIA. *CATIA V6 — Industrial Design — CATIA Natural Sketch Showreel*. Dec. 6, 2011. URL: https://www.youtube.com/watch?v=xXYCaabNHCo.

[2] Jeroen Arendsen. *Pen Gestures*. 2006. URL: http://jeroenarendsen.nl/2006/04/pen-gestures/.

[3] The Printing Collection. *Raster Images vs. Vector Graphics*. URL: http://www.printcnx.com/resources-and-support/addiational-resources/raster-images-vs-vector-graphics/.

[4] Toms Dorta, Michael Hoffmann, and Gke Knayolu. *Hyve 3D*. 2014. URL: www.hyve3d.com.

[5] EIZO Global. *How Can a Screen Sense Touch? A Basic Understanding of Touch Panels*. 2011. URL: http://www.eizoglobal.com/library/basics/basic_understanding_of_touch_panel/.

[6] Desmond Morris et al. *Gestures*. Johnathan Cape, 1979.

[7] Luke Wroblewski. *Touch Gesture Reference Guide*. 2010. URL: http://www.lukew.com/ff/entry.asp?1071.