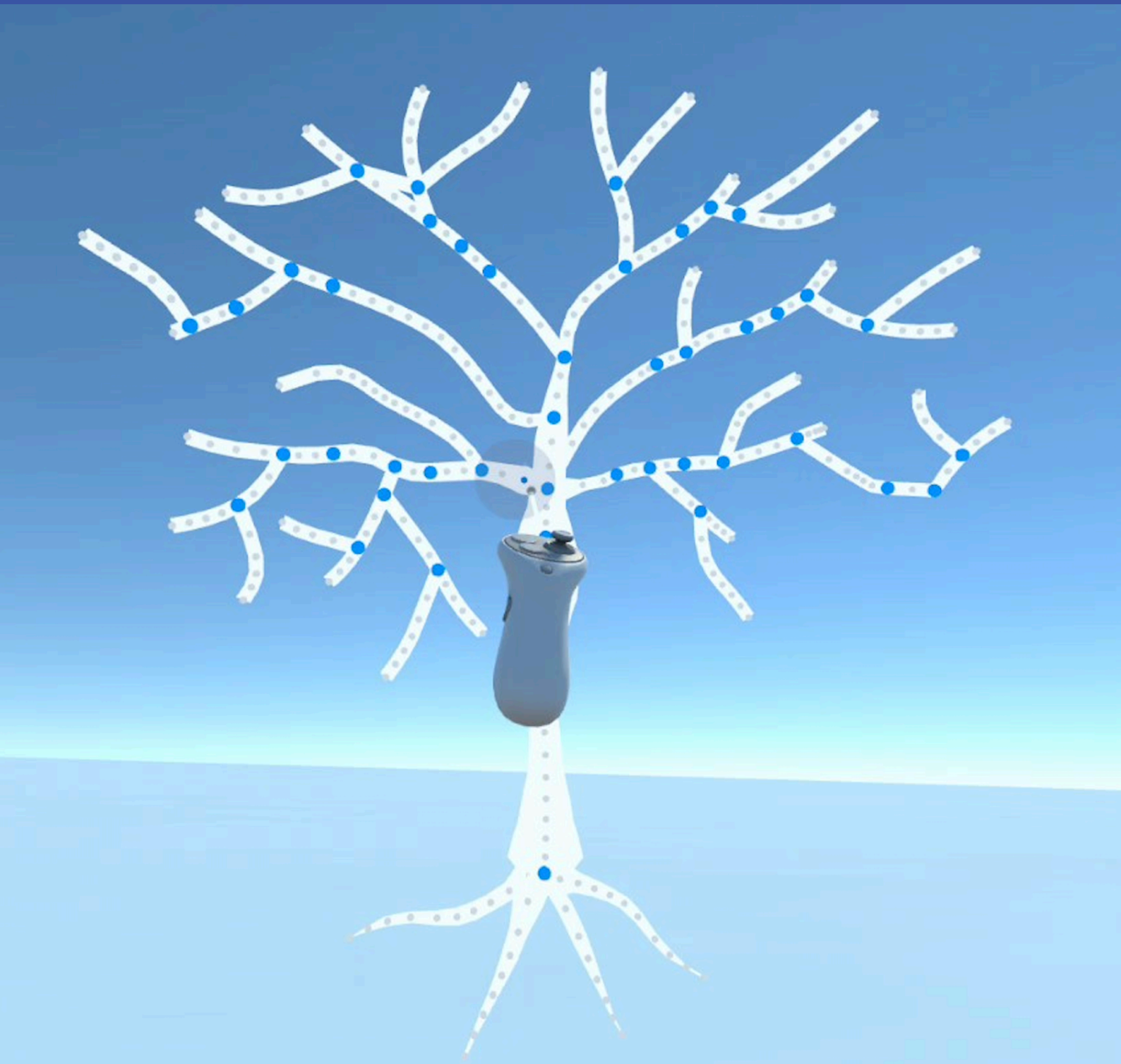
DTU Compute
Department of Applied Mathematics and Computer Science

VR skeleton stroke based drawing system for 3D model creation

Master Thesis



VR skeleton stroke based drawing system for 3D model creation

Master Thesis
March, 2025

By
Krystallia Zoi Angeli, MSc Computer Science and Engineering, DTU

Supervisor
J. Andreas Bærentzen, Associate Professor, Department of Applied Mathematics and Computer Science, DTU

Cover photo: Krystallia Zoi Angeli, 2025
Published by: DTU, Department of Applied Mathematics and Computer Science, Richard Petersens Plads, Building 324, 2800 Kgs. Lyngby Denmark
www.compute.dtu.dk

Approval

This thesis accounts for 35 ECTS and has been prepared over seven months at the Department of Mathematics and Computer Science, at the Technical University of Denmark, DTU, in partial fulfillment for the degree Master of Science in Engineering, MSc Eng.

It is assumed that the reader has a basic knowledge in the areas of Virtual Reality and Algorithms.

Krystallia Zoi Angeli - s230416

_____

Signature

Kongens Lyngby, March 28, 2025

_____

Date

Abstract

This thesis explores a Virtual Reality stroke based modeling system that enables users to create 3D models by sketching skeletons in an immersive environment. Traditional 3D modeling techniques are often complex and require prior experience with specialized software. The proposed system aims to be a more intuitive and user-friendly alternative, allowing novice users to draw mid-air freehand strokes that are then converted into 3D meshes by adding volume, through a custom plugin developed.

While VR provides an immersive way of drawing, it also introduces challenges, such as reduced stability and precision due to the lack of a physical drawing surface. To address these issues, the system developed provides real-time editing tools and automatic stroke adjustment, that help users improve the accuracy and appearance of their sketches without needing to redraw them from scratch.

The evaluation of the system focuses on intuitiveness, efficiency, and accuracy, assessed through user studies and sketching metrics. The results show a high level of user enjoyment with participants finding the system easy to learn and engaging to use. These findings show the potential of VR sketching tools for 3D modeling, particularly for users with no prior experience in traditional modeling software.

Acknowledgements

I would like to thank my supervisor, Andreas, for his valuable guidance, insightful suggestions, and continuous support throughout the project. I also extend my gratitude to the participants of this study for testing the software and providing helpful feedback. Last but not least, I am deeply grateful to my family and loved ones for their constant support and motivation during the entire process.

Contents

1 Introduction

1.1 Motivation

Computer technology has become an amazing tool for creative thinking for many years. One of the most basic forms of human visual expression and communication is drawing, and Virtual Reality (VR) is the best way to display scenes that were previously unimaginable. In addition, VR technology plays a significant role in a number of fields, including science, education, medicine, and the military [1].

Traditional 3D modeling systems like Blender [2], Maya [3], or CAD software [3] require significant expertise and are complex for beginner users. Many drawing based VR applications work by drawing directly in mid-air using physical tools and having strokes appear synchronously at the same physical location.

With commercially available input modalities (such as hands and VR controllers), it is difficult to enable precise drawing in unrestricted mid-air environments. For instance, the idea of using 3D sketching as a medium during conceptualization, typically excites designers, but they frequently express frustration with drawing in VR because of the lack of control and accuracy over their strokes [4].

Figure 1.1: Mid-air drawing [5]

Sketching curves mid-air in VR (figure 1.1) is a promising but also challenging task. In the early phases of design, the ability to create freehand sketches directly in 3D space, in contrast to drawing on paper (or other flat surfaces), has significant consequences for how users may visualize and convey 3D shapes and spaces [6]. Researchers and users of stroke based immersive 3D sketching, have expressed the difficulties they had while trying to accurately sketch 3D shapes. This imprecision was frequently seen as a drawback of Virtual Reality and Augmented Reality (VR/AR) 3D sketching. In order to get around their lack of control, many users requested tools with features that beautify strokes from freehand sketches, to make them look aesthetically pleasing [7], to add haptic feedback, visual depth cues, scaling, stroke snapping, and editing tools [6]. As a result, VR provides a natural 3D drawing environment but existing software still has limitations in usability, precision, and also real-time conversion to 3D models.

Researchers have studied for years sketch based 3D modeling. Most existing sketch based 3D modeling methods are using 2D sketches on paper or a touch screen panel as input because of their accuracy and user experience [8] [9] [10] [11]. Then, 3D models are being generated, but the existing research focuses mainly on reconstructing 3D models using deep neural networks from 2D or 3D sketches [12] [13] [14], without employing accurate real-time generation of 3D models from sketches.

1.2 Scope

This thesis aims to develop a VR skeleton drawing system for the creation of 3D models. Our goal is to address common challenges users face when designing in mid-air while enabling real-time conversion of 3D skeleton sketches into 3D meshes.

The software allows users to sketch 3D curves in mid-air, with features such as stroke beautification, junction snapping, and point clustering to enhance accuracy and precision. It also includes editing tools that let users refine their sketches by drawing over existing lines, adjusting curves, and modifying line point radius, because accurate curves are important in drawing skeletons. Additionally, users can generate a 3D mesh from their sketches in real-time, with extra tools for previewing sketches, viewing the mesh wireframe, and saving the sketch and the final 3D model.

Moreover, the software is designed for novice users with no prior experience in 3D design or modeling. It prioritizes intuitiveness and ease of use, making 3D creation user-friendly.

The application is developed for the Meta Quest 3 headset, utilizing controllers as the primary input device. Final 3D sketches and meshes can be saved and exported as `.obj` files, which are compatible with most 3D modeling software for further editing. However, this application is designed for drawing skeletal structures and generating 3D meshes rather than editing them, as it does not include features for sculpting, texture mapping, or animation.

1.3 Goal

The primary aim of this thesis is to develop an intuitive, VR-based freehand drawing application that allows users to create 3D models by sketching skeletons. The system will convert hand-drawn strokes into 3D models, providing an innovative approach to immersive 3D modeling.

Objectives:

- Assess intuitiveness and efficiency: Investigate how intuitive and fast it is for users to draw 3D curves in a VR environment. This includes analyzing the impact of assistive and editing features, such as stroke adjustment, mirroring, and stroke smoothing.
- Measure task efficiency: Quantify the time taken to complete drawing tasks and track moves and tools used by users. This will help identify areas of improvement in the system's usability and responsiveness and also user preferences.
- Evaluate the accuracy and quality of generated 3D meshes: Assess the effectiveness of the VR system in transforming hand-drawn skeleton sketches into accurate 3D meshes.
- User Experience assessment: Collect feedback on user satisfaction regarding comfort, ease of learning, and preference. This will involve a questionnaire for identifying user's preferences and level of enjoyment while using the system, as well as any usability challenges that may arise.

- Performance evaluation: Measure the performance of the system in terms of processing time, frame rate (FPS), CPU/GPU load. This will help show the feasibility of real-time interaction in VR environments and identify potential performance problems.

1.4 Structure Overview

This thesis includes six main chapters, starting by explaining foundational concepts to the implementation and evaluation of a VR based 3D modeling system using hand-drawn curves.

Chapter 1, *Introduction*, presents the motivation behind this research, defining the problem, scope, and goals. It provides an overview of the challenges in VR-based sketching and outlines the structure of the thesis.

Chapter 2 and 3, *Basic Concepts and Literature Review*, examine existing work in relevant fields, including sketch based modeling, VR interaction methods, and polygonal mesh generation. This chapter discusses different approaches and highlights research gaps, explaining how this study contributes to addressing them.

Chapter 4, *Methodology*, describes the technical approach used in developing the VR sketching system. It details the stroke processing techniques, intersection detection methods, clustering algorithm, and mesh generation process. Additionally, it explains the software features and interaction mechanisms designed to improve usability.

Chapter 5, *Implementation*, focuses on the practical realization of the system. It covers the VR drawing workflow, KD tree based intersection detection, point clustering techniques, and the use of the GEL library for mesh generation. The chapter also describes the user interface design and integration within the VR environment.

Capter 6, *Tests and Results*, presents the evaluation of the system through early informal testing, expert user sketches, and a structured user study. The results are analyzed to assess usability, accuracy, efficiency, and preference, providing feedback into the software's effectiveness.

Finally, chapter 7 and 8, *Discussion and Conclusion*, summarize the findings of the study and discuss the limitations of the current approach. It also suggests potential improvements and outlines directions for future research in immersive 3D modeling using hand-drawn curves.

2 Basic concepts

2.1 Traditional 2D sketching

For many years, sketching has been essential for design and conceptualization. On one hand, traditional 2D sketching is rooted in human cognition and training from an early age, as it is a natural tool for artists and designers. On the other hand, Virtual and Augmented reality have enabled users to draw strokes mid-air without a specified canvas, making them very engaging mediums in these domains.

A key advantage of 2D sketching, is how ergonomic it is. In contrast with 3D sketching, which can be challenging for mid-air drawing, 2D sketching provides a physical surface that enhances steady strokes and reduces user fatigue. Traditional 2D sketching also offers greater control over strokes, making it easier to create detailed, accurate drawings and has become *"as accurate as handwriting"* according to Arora et al. [11].

However, a major limitation of 2D sketching is the difficulty of conveying absolute depth on a flat (2D) surface. Artists often use multiple perspectives, construction lines, and scaffolds to represent 3D objects in 2D. In comparison, 3D sketching in VR/AR environments inherently incorporates depth, eliminating the need for simulated depth techniques used in flat sketches [11].

2.2 Sketching in VR

VR sketching allows users to sketch freely in 6 degrees of freedom (6DOF), offering an unconstrained drawing experience. Unlike traditional 2D sketching, users can directly create volumetric structures in real-time 3D space, making it useful for rapid prototyping and conceptual design.

Sketching in VR offers the user the ability to express themselves without constraints but at the same time it is very difficult to draw accurate curves in mid-air [15][16]. It was found that it is challenging sketching even simple strokes, such as straight lines and circles. The absence of physical support also leads to increased arm fatigue and reduced accuracy over extended use.

Immersive environments were also found to engage more extensive visual and spatial brain functions compared to traditional 2D environments. VR sketching applications also have the potential to activate and improve specific neural processes associated with design related processes and spatial thinking [17].

2.3 Curves in 3D modeling

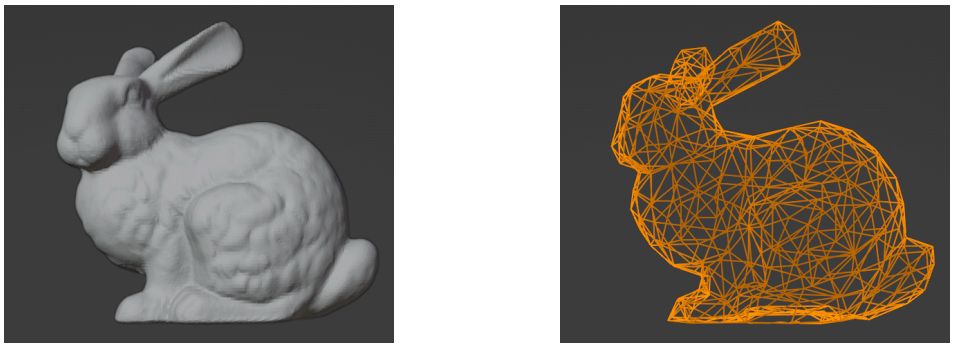
Curves are very important in 2D and 3D sketching, forming the basis for smooth, continuous shapes. In traditional 2D sketching, curves are drawn freely and controlled by hand movements, while in 3D modeling, curves serve as guides for surfaces, skeletons, and complex shapes.

Unlike polygonal meshes, which approximate surfaces using flat polygons, curves provide a mathematical representation of smooth shapes. Curves are commonly used in sketch-based modeling, where strokes define the shape of a 3D object, as well as in computer-aided design (CAD) [3] [18], where they play a role in both design and animation. In rigging, curves are also used to define motion paths and skeletons. Curves are also

essential in skeleton-based modeling, where they act as structural representations for 3D forms.

2.4 Polygonal mesh

Once a 3D sketch is created, it needs to be converted into a structured representation, usually a polygonal mesh. 3D mesh models play a major role in 3D modeling, a process used to design and represent 3D objects, environments, and spaces. As a key component of 3D modeling, these meshes are widely used in fields such as architecture, construction, interior and product design, as well as animation and film production, but they are also essential for visualization and prototyping in various domains, including video game development, Virtual and Augmented reality.



(a) Bunny object (b) Bunny mesh

Figure 2.1: 3D Mesh model

Meshes made of polygons, and triangles in particular, are necessary for representing shapes, and their significance is only growing. They provide the ability to directly manipulate shapes, compared with the past when it was needed to change the shape to a different representation [19].

A 3D mesh model (see figure 2.1) is a 3D representation of an object, consisting of polygons and it consists of a collection of faces, edges and vertices that define shape and structure of the object. Two neighboring vertices, each of which is a coordinate or point in three dimensions, are connected by an edge. The faces that create the surface of the object are known as polygons and are enclosing the edges. The polygons used in a 3D mesh are typically quadrangles or triangles, that can be subdivided into lines and vertices in X, Y, Z coordinates. Polygonal mesh representation is mainly used and useful in geometry processing and manipulation [19].

3 Literature Review

A fascinating area of study in computer science and industrial design has been finding simpler methods for creating 3D objects. Our system expands on previous research that attempts to use natural interaction to create an intuitive, VR based 3D sketching system that generates 3D meshes.

3.1 Sketch-based modeling approaches

3.1.1 3D Sketching in VR

Arora et al. [15] define 3D sketching as a technology driven method of drawing where: (i) marks are made in a 3D, body centered space rather than on a flat surface, (ii) a computer tracking system that captures the movement of the drawing tool, and (iii) the resulting sketch is displayed in the same 3D space, often through technologies like Virtual and Augmented Reality. This method is known for being engaging, natural, fast, and flexible, making it structured for 3D input.

Due to these benefits, several commercial applications, such as Tilt Brush [20] (now open-source OpenBrush [21]), Gravity Sketch [5], and Quill [22], have made 3D sketching widely accessible across various fields, including art, modeling, filmmaking, architecture and design.

However, positioning strokes accurately in 3D space remains a challenge due to factors like high cognitive and sensorimotor demands, depth perception issues in stereo displays, and the lack of physical support. Researchers have explored how users control and adapt to sketching in mid-air, as well as the learning curve associated with 3D sketching [23].

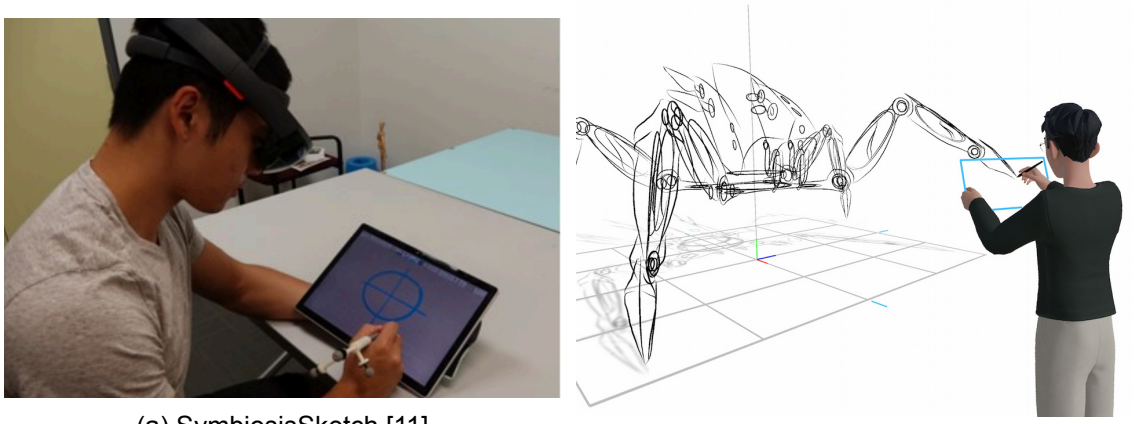
Compared to 2D sketching, the lack of a supporting surface and 6DOF offered in VR, produce the need for implementations that define more accurate curves in 3D space. Another factor influencing the approximate placement of strokes in relation to one another in sparse sketches is the absence of depth in 2D [15]. This thesis addresses these challenges by implementing a variety of different features and tools for accurate sketching.

To address 3D drawing challenges in immersive sketching, various interface solutions have been developed over the years. There is a variety of research [9] [11] [12] that focuses on using 2D sketches for 3D modeling because of more accurate sketches compared to 3D sketches. In contrast with these studies, there are also several others that are trying to solve directly immersive 3D sketching challenges like this thesis.

3.1.2 VR 3D sketching on 2D surfaces

In recent years, studies have explored the use of 2D physical surfaces as an intuitive medium for creating 3D sketches in VR and AR environments. These approaches aim to bridge the gap between the accuracy of 2D sketching techniques and the spatial complexity of 3D modeling.

SymbiosisSketch by Arora [11] is a hybrid example where they used a tablet, a 2D surface, for 3D sketching and an AR headset for the visualization of the drawing. This hybrid setup allows users to have the precision of pen-based input while benefiting from AR visualization. In a similar way, RobotSketch by Lee et al. [10] is another example where they use a tablet, an pen and multi-touch gestures in a VR workspace to sketch robots in 3D, allowing the users to view their sketch in real time. Users can sketch in 2D while immediately viewing their drawings in 3D, offering a natural workflow for conceptual design.



(a) SymbiosisSketch [11]

(b) RobotSketch [10]

Figure 3.1: 3D sketching on 2D surfaces

However, there are still challenges in ensuring depth perception accuracy, user ergonomics, and creating better interaction workflows. Our study will explore mid-air 3D sketching with a focus on creating skeletons of objects in a VR environment.

3.1.3 3D Sketching techniques in VR

Various tools have been developed over the years for real-time sketching, rendering, and editing curves directly in 3D space to enhance precision, usability, and efficiency. Recent systems use VR headsets to render these curves in real time, representing them as either ruled surface ribbons or tubes with round, cylindrical shapes [14].

Commercial VR sketching applications like Open Brush [21] and Gravity Sketch [5], allow users to create freeform strokes in immersive environments. While these tools focus on artistic expression and creative design, our approach is focused on structured 3D modeling, prioritizing the accuracy of strokes to create object skeletons.

Multiplanes [7] was developed by Machuca et al. in 2017, to help users freehand draw in VR with ”Beautification” techniques such as automatically detecting drawing planes and guides based on strokes and the controller. The guides are called ”beautification trigger points” and are created based on previous user strokes, which show geometrical relationships to previous strokes and snapping points. Multiplanes system automatically detects a drawing plane and also automatically beautifies a stroke in real-time by recognizing the shape of a stroke (as a line, arc, or circle), and it does not beautify it if the geometry of the curve does not resemble any known shape. Our system also automatically beautifies strokes in real time, smoothing curves to appear more natural while preserving their overall shape. It does not turn strokes into predefined shapes, but simply refines them to eliminate irregularities and sharp edges. Multiplanes detect and adjust strokes to fit geometric constraints, but our system preserves the user’s stroke shape while smoothing inconsistencies.

Another software developed by Machuca et al. in 2019 [24] is Smart3DGuides, where they designed a different approach, using visual guides to support freehand drawing in VR. They tested the use of three different kinds of visual guides (static and dynamic) to help the users draw more accurate strokes and straighter shapes. Unlike Smart3DGuides, which provides predefined visual guides to assist users, our system refines strokes dynamically, without requiring predefined templates.

Arora and Singh [25] developed an application to draw mid-air curves in VR, by using 3D surfaces to make it more accurate and controlled and overcome perceptual and ergonomic challenges. They tried projecting in real time, the strokes on 3D surfaces with different techniques. Contrary to this study, our application will focus on optimizing user-drawn strokes, snapping, beautifying, and editing them for better accuracy.

Cassie [26], developed by Yu et al., is another immersive freehand 3D sketching system that uses several beautification techniques, such as junction detection and smoothing of curves, similar to our system but with different implementation techniques. They also refine strokes by making them planar, but we chose not to include this feature in our system since our focus is on creating the skeleton of objects while having interactive editing tools to improve the skeleton structure. Compared to our software, they don't use editing tools, and their application is focused on *"creating connected curve network armatures"* instead of skeletons, where the shape of a curve is very important and has to be defined carefully.

3.1.4 Generation of 3D models from sketches in VR

3D sketching and sketch-based 3D modeling have been studied for many years and can be divided into two main approaches: interactive and end-to-end. Interactive methods require users to follow specific steps, gestures, or annotations, which can make the process complex and almost unapproachable for novice users. End-to-end methods, such as those using templates or retrieving existing shapes, can generate results quickly but often lack flexibility. More recent techniques use deep learning to reconstruct 3D models from sketches, treating it as a single-view reconstruction problem. However, since sketches are abstract and lack texture, extra information is needed to create high-quality 3D models [12] [27].

For example, Chen et al. [28] created Reality3DSketch, which is an immersive 3D modeling application. With a monocular RGB camera, the user can capture the surrounding area and then he can draw an object in the real-time reconstructed 3D scene. The 3D object is generated from the sketch with neural networks and placed in the desired location.

There are several methods for processing 3D drawings that focus on connected curve networks, where surface patches are defined by closed loops of interconnected curve segments [29][30][31]. Another approach is to use drawn curves for 3D shape modeling to convert them into surface models. One approach, SurfaceBrush, processes ribbon strokes created in VR applications like TiltBrush and can also work with other VR systems that support similar stroke based input [14].

Many algorithms have been developed for surfacing dense point clouds. Berger et al. [32] compared these algorithms to those used to surface the sparse stroke clouds typically produced by designers when sketching in VR or using sketch-based modeling systems. One of the most recent studies was conducted by Yu et al. [33] in 2022, focusing on transforming *"unstructured 3D sketches into piecewise smooth surfaces that preserve sketched geometric features"*. Their algorithm uses freeform 3D sketches, which are imperfect and lack structured connectivity. Traditional surface reconstruction techniques struggle with these loosely connected stroke collections, as they are typically designed for well defined curve networks or sparse point sets. To overcome this, their approach introduces an iterative segmentation and optimization process that refines an initial proxy surface, ensuring that smooth patches are formed while maintaining sharp boundaries along strokes.

While Yu et al. [33] addressed the challenge of converting unstructured 3D sketches into smooth surface patches, our system takes a different approach. We focus on developing

a VR 3D modeling tool that allows users to sketch in 3D space and generate objects through stroke based *inverse skeletonization* technique to transform freehand sketches into 3D meshes.

3.2 Graph-based methods and geometry processing

3.2.1 Skeleton-based 3D mesh generation

The generation of 3D meshes based on skeletons is crucial for many applications in computer graphics, animation, interactive modeling, and implicit surface reconstruction. Using skeletal frameworks as representations, these methods enable mesh creation, and deformation, making them very important in interactive modeling and real time applications.

One of the most significant applications of skeleton based 3D mesh generation is *interactive implicit modeling*, where skeletal structures serve as a foundation for constructing smooth 3D surfaces. Bloomenthal and Wyvill [34] introduced an interactive implicit modeling framework that utilizes skeleton-based methods to define implicit surfaces with smooth transitions and natural deformations. Instead of explicitly constructing a mesh, *implicit modeling* defines the surface mathematically as an isosurface around the skeleton. The skeleton acts as a set of control points, and then it generates a smooth 3D surface around it. The output is an implicit surface, not an explicitly structured quad mesh.

Zanni et al. [35] introduced a method for *scale-invariant integral surfaces*, showing how skeleton modeling techniques can be used to reconstruct smooth surfaces from skeletal inputs. This approach generates a smooth 3D surface that automatically adapts to changes in scale, while keeping important features based on mathematical techniques. Unlike methods that create structured quad meshes, their system produces flexible, smooth surfaces that respond well to deformation.

The approach used in this thesis shares similarities with existing skeleton-based 3D mesh generation techniques. *Inverse skeletonization* [36] is a technique used in 3D modeling where a structured polygonal mesh is generated from a skeletal representation and is used in this thesis. In the next section (3.2.2) this technique will be further analyzed.

These methods rely on an explicit skeleton that serves as a structural guide for generating smooth and well-connected meshes. However, the key difference in this thesis is that in isosurface methods, the final mesh is an unstructured triangle mesh, which can be difficult to edit. These meshes usually require extra steps before they can be used for sculpting. The method in this thesis uses Face Extrusion Quad (FEQ) meshing to create structured quad meshes with clean geometry, making them easier to refine, animate, and sculpt.

3.2.2 Inverse skeletonization for mesh generation

Skeletal structures are frequently used in 3D graphics as armatures for creating 3D models. Skeletal representation also works well as a shape abstraction, since the nodes define junctions where parts meet and the edges represent parts [36]. This provides us with the capability to convert a skeletal representation (with nodes and edges) to a polygonal mesh, which is the approach we will implement in our application. This is called *"inverse skeletonization"* [36] (see figure 3.2) and it creates coarse shapes. It provides us with the ability to generate a basic shape from scratch, by designing a skeleton that is simpler and easier than directly creating a corresponding polygonal mesh, especially for novice users.

The Face Extrusion Quad Mesh algorithm plays an important role in inverse skeletonization, as it enables the reconstruction of 3D models from skeletal representations. It is a meshing technique used to generate quad meshes from an input surface by extruding its faces [37]. This method allows for thickness control, branch merging, and refinement

of the generated geometry. Instead of keeping the graph as a thin line from a graph or a sketch, it extrudes faces (quads) around the edges to give the graph thickness. This creates a solid structure rather than just a set of lines. To summarize, FEQ transforms a graph skeleton into a structured 3D model, bridging the gap between stroke-based modeling and final 3D mesh representation.

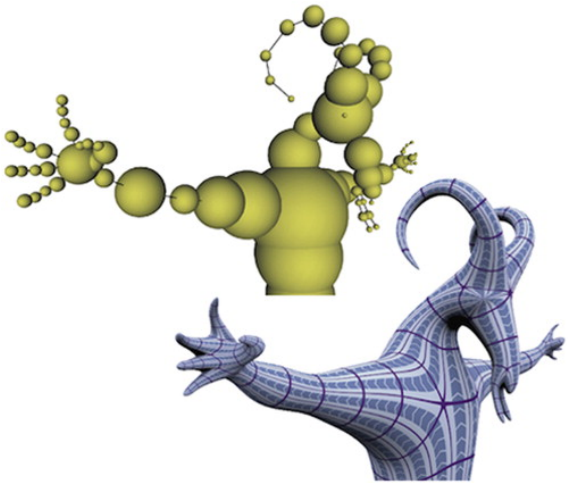


Figure 3.2: Inverse skeletonization [36]

The GEL library, offers a functionality for inverse skeletonization, enabling the creation of FEQ meshes from graphs. *"Geometry and Linear algebra Library"* (GEL) library [38] is a collection of C++ classes and functions distributed as source code and it is used for geometry processing tasks [19]. Specifically, it can be used to reconstruct meshes from point clouds but it also provides many more capabilities for geometry manipulation like Laplacian smoothing and curvature flow, which may be useful for refining meshes generated from sketches. Tools like this are very important in VR because these applications demand techniques that are computationally efficient for real-time interactions. This is the library that will be explored to implement the plugin that generates a 3D mesh from a sketch.

3.3 Usability and effectiveness of VR modeling

3.3.1 User experience in VR

Designing for Virtual Reality is a complex challenge. When executed well, VR experiences can be engaging and enjoyable, but poor design may lead to frustration and discomfort. Many issues stem from a lack of understanding of human perception, interaction, and design principles. User Experience (UX) research in VR explores the causes of motion sickness, strategies for creating engaging content, and best practices for designing and refining VR applications effectively [39]. In this thesis, we tried to minimize lag to avoid motion sickness as much as possible by using fast algorithms, trying different techniques and prioritizing the experience.

Natural user interactions offer significant advantages for artists using VR head-mounted displays (HMDs), enabling more intuitive and immersive engagement with virtual environments. By mimicking real-world actions, these interactions reduce cognitive load and enhance usability [23].

According to a study conducted by Machuca [40] in 2024 *"there is no "holy grail" UI for 3D sketching as one interface cannot satisfy the needs of all application domains"*. By

measuring usability, we can evaluate how good a User Interface (UI) is at fulfilling its purpose. However, it is important to evaluate UI and interaction techniques together. That means that results may vary depending on the experience of the user using the 3D design application and the VR headset being used. To measure usability, we also have to evaluate the 3D sketching interface, taking into account factors such as the learning curve, ease of use, efficiency, and accuracy [40]. Hence, these are the key aspects that will be evaluated in our user tests.

For 3D sketching in VR, providing artists with appropriate tools is essential. Unlike traditional 2D sketching, where depth and perspective are implied on a flat surface, 3D sketching requires the artist to convey spatial relationships more dynamically. Tools like Tilt Brush [20] allow artists to explore creative dimensions, incorporating elements such as movement and audio synchronization, enriching artistic expression.

Studies [41] show that asymmetric interactions, where hand gestures are integrated with controllers, can significantly reduce task completion time for complex tasks while maintaining accuracy and usability. However, for simpler tasks, single controller interactions remain equally effective. Despite usability scores being generally high across different input methods, many users preferred the two-controller setup due to its stability, reliability and accuracy compared to gesture-based alternatives, which is why they were chosen as the input method for this software.

However, another study [42] that compared one-handed and two-handed gesture-based user interfaces in VR found that users generally favored one-handed interactions due to their flexibility. Nevertheless, two-handed interactions offered greater control, particularly for tasks involving 3D object manipulation. The choice between one-handed and two-handed interactions should depend on the complexity of the task.

3.3.2 Best practices in VR

Effective VR interaction design follows several key principles. Using natural controls that mimic real-world movements helps create an intuitive user experience. Providing clear feedback through visuals, sound, or haptic responses ensures that users receive immediate confirmation of their actions. Moreover, user comfort is essential, requiring careful attention to ergonomics and movement constraints to reduce motion sickness and fatigue. Additionally, accessibility should be prioritized, allowing users with different abilities and preferences to interact with the software [43].

Users can only remember a limited number of gestures, thus when designing for motion controls, it is best to build upon a familiar set or combine gestures for more complex interactions. Ideally, interactions should be designed with natural affordances that users instinctively respond to, rather than requiring them to memorize specific poses, movements and buttons [44].

Sound plays a crucial role in enhancing immersion in VR experiences. When integrated with hand tracking and visual feedback, it can help create the perception of tactile sensations. Additionally, audio can effectively signal whether an interaction has been successful or unsuccessful. Therefore, integrating audio cues to provide clear feedback has been fou to be a very good practice in VR interaction design [44].

Traditional UX practices [45] for VR applications that were incorporated into our software to enhance user experience include: allowing users to look around before requiring interaction, clearly explaining button mappings to avoid confusion, using animated text or audio cues for instructions, and ensuring that static text is large and readable when necessary. Additionally, the system should provide tracking feedback through visual and haptic

cues. It should also be minimalistic and intuitive, reducing the number of buttons and avoiding unnecessary UI elements.

3.4 Research gaps

3.4.1 Identified limitations in existing work

Creating 3D models is often difficult for novice users due to the complexity of traditional 3D modeling software, which requires expertise in structured workflows and detailed manipulations. While some VR modeling tools provide a more natural sketching experience, they lack assistive and editing features that improve precise strokes and ease of use. Users may struggle with mid-air drawing, leading to inaccurate strokes, disconnected lines, and difficulty maintaining structural continuity.

Many existing approaches rely on neural networks for 3D shape generation, which can introduce unpredictability, require extensive training data, or surface fitting on 3D sketches that need the surface sketch. In contrast, there are no VR modeling applications that generate 3D meshes using an inverse skeletonization approach. Current solutions typically focus on surface-based sculpting rather than skeletal modeling, making it harder to create well-defined structures for further refinement in traditional 3D software.

Another challenge in usability is real-time continuous rendering and high computational demands can cause lag, negatively impacting the immersive user experience. Many similar applications also lack real-time sketch editing tools, requiring users to redraw entire sections rather than making precise modifications.

3.4.2 How this Thesis addresses the gaps

This thesis introduces a VR 3D sketching system that allows users to intuitively draw skeletons in mid-air and instantly convert them into structured 3D meshes. While surface modeling requires expertise in modeling software, our system focuses on creating object skeletons, which are easier to construct accurately. As a result, the precision of the curves becomes especially important, differentiating our approach from most of the research, where curve accuracy is often less emphasized.

To overcome common usability challenges while focusing on skeletons, the system builds on research and integrates features such as stroke beautification with a different method, for automatic smoothing of hand-drawn lines without making them look artificial. Junction snapping to ensure proper connectivity between strokes and point clustering to avoid the creation of close junctions that are not needed in skeletons. It also includes real-time editing tools that are often missing in other applications, such as stroke adjustment and point repositioning. These features support stroke reshaping, which is important in skeleton drawing, where precise and connected curves are essential.

In this project, we combine immersive 3D sketching with inverse skeletonization to generate smooth surfaces from skeletons while keeping local radius information from the lines. The developed native plugin for *inverse skeletonization* efficiently converts hand-drawn strokes into accurate 3D meshes, without relying on machine learning. The plugin runs directly on the VR headset, enabling real-time processing without the need for external computation. Users can export final meshes as `.obj` files for further editing in standard 3D modeling software.

4 Methodology

4.1 Software features

The VR sketching software includes several features to improve the drawing process, enhance the user experience and make 3D model creation easier. These features are grouped into three main categories: curve drawing, sketch editing, and mesh generation. Table 4.1 provides an overview of these functionalities.

Curve drawing features help refine strokes, create smooth junctions, and prevent unwanted shapes. Sketch editing allows users to modify their sketches by adjusting curves, mirroring strokes, moving points, and changing the radius. It also includes basic tools like undoing and clearing the sketch. Mesh generation provides users the ability to convert sketches into 3D meshes, interact with the object, save their work, and clear the mesh when needed.

Category	**Features**
Curve Drawing	Curve beautification
	Automatic junction creation
	Avoid unwanted triangle creation between junctions
Sketch Editing	Draw adjustment curves
	Mirroring strokes
	Move points
	Adjust point radius
	Preview sketch
	Undo lines
	Clear drawing
Mesh Generation	Mesh creation
	Wireframe
	Move object
	Save 3D object
	Clear mesh

Table 4.1: Feature Overview

4.2 Immersive drawing

4.2.1 Hand-drawing

Within the immersive VR environment, users can freely draw strokes in 3D space to create sketches representing the skeletons of objects. This drawing functionality is achieved by tracking the position of the user's hand. Users can draw by pressing the trigger button on the controller, and the system records the trajectory of the controller as a new stroke. To do this, a virtual pointer is attached to the controller, allowing the system to track the pointer's trajectory. By simply pressing or releasing the trigger button, users can interac-

tively generate lines in 3D space, providing a highly intuitive approach to draw curves in VR.

To ensure the user's sketches can be easily manipulated, saved, or extended, it is important to connect the individual strokes to a structure. This design is further enhanced with additional features that optimize the sketch without complicating the drawing process for the user.

Drawing in an immersive environment with six degrees of freedom (6DOF) can be challenging, so it is important to provide support that simplifies the experience for the user. This includes beautifying the strokes, detecting junctions between curves, and visually highlighting these intersections to assist the user during the drawing process.When a new stroke is created by the user, it undergoes several adjustment steps before being finalized. These adjustments, although processed instantly, ensure that the final stroke is adjusted and closer to what the user intended.

4.2.2 Curve beautification

When a stroke is drawn, an automatic beautification process is necessary to refine its appearance. This optimization can be implemented using various algorithms. In this software, *Laplacian smoothing* [46] is used to achieve this result 4.1. This algorithm was chosen because of its computational efficiency, making it suitable for real time applications. Laplacian smoothing algorithm is applied to a sequence of 3D points $P_{\text{new}} = \{P_1, P_2, \ldots, P_n\}$ (that form a stroke) to reduce noise and improve continuity. The smoothing for each point P_i in the line is defined as:

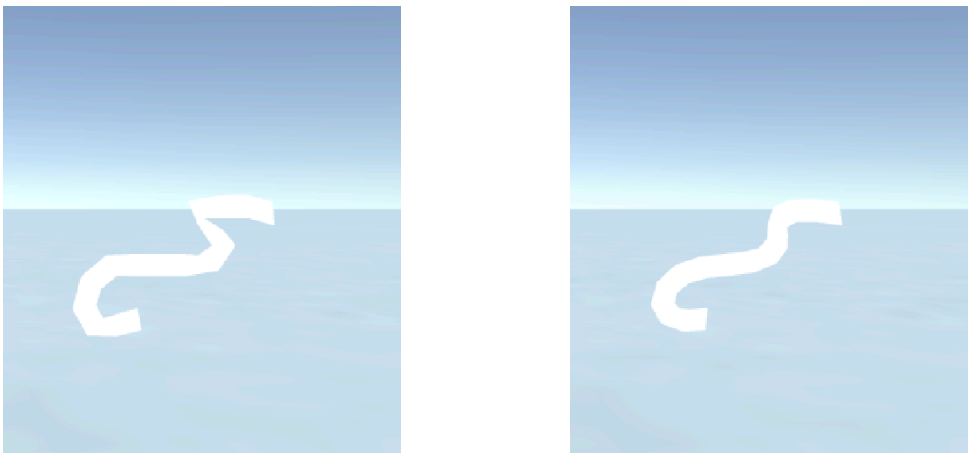
$$P_i^{\text{new}} = (1 - \alpha)P_i + \alpha \frac{P_{i-1} + P_{i+1}}{2} \tag{4.1}$$

where:

- P_i represents the position of the current 3D point in the sequence, while P_{i-1} and P_{i+1} are its neighboring points.
- The parameter α ($0 \leq \alpha \leq 1$) is the smoothing factor, which controls the influence of neighboring points on the updated position.
- P_i^{new} denotes the updated (smoothed) position after applying the smoothing operation.

To achieve a balance between maintaining the natural feel of the curve and avoiding an overly processed or unrealistic appearance, we carefully chose the smoothing parameters after several tests. The best results were achieved using a smoothing factor $\alpha = 0.5$ and performing 5 iterations. This configuration ensures that the curves appear smooth and natural, without resembling trembling lines or feeling overly artificial.

After merging intersecting points, the system applies a Laplacian smoothing step to the newly created stroke. Intersection or junction nodes (referred to as "fixed points") are excluded from the smoothing process to ensure their exact positions remain preserved.



(a) Raw curve (b) Smoothed curve

Figure 4.1: Curve beautification

4.2.3 Junction detection

When drawing multiple strokes in VR, it is desirable to identify and merge points from different strokes that are very close to each other, such as intersections or junctions to have a connected structure. There are several different ways the junction could be detected such as colliders detection, linear search etc. But to achieve this, the *KD-Tree algorithm* [47] was implemented with multiple constraints to produce the desired results effectively.

Compared to a linear search of all existing points, the KD-Tree reduces the complexity of the query, especially as the number of points increases. This efficiency is critical in real time VR applications, where timely feedback helps to maintain immersion and user interaction.

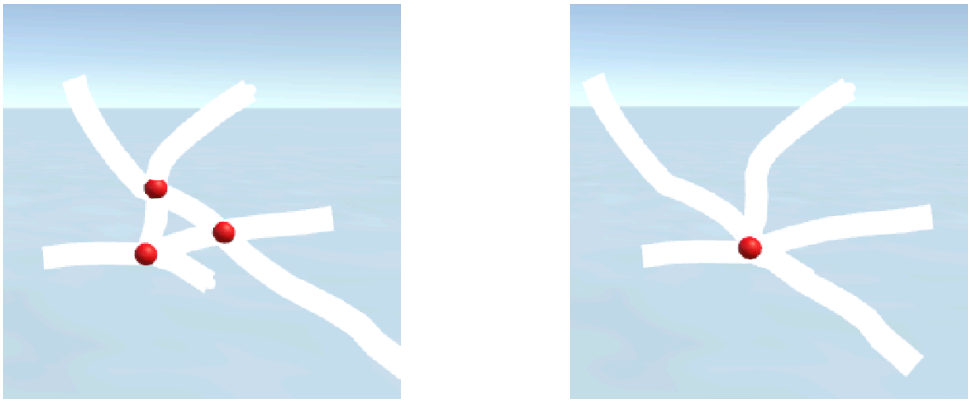
When the system needs to determine if a newly created point intersects an existing stroke, it queries the KD-Tree (for each new point) for the *nearest neighbor* and compares their distance against a small threshold. If the distance between two points is smaller than this threshold, the system treats the new point as a point that already exists and intersects, so it snaps the new point at the same position as the old point and they create a junction. After several tests, the optimal results were found at `0.02` as a threshold.

(a) Without junction detection (b) With smoothing and junction detection

Figure 4.2: Junction detection

4.2.4 Clustering and point merging

A problem that occurred when the KD-tree implementation was added to the system, was that some times, when two strokes were very close to each other, small triangles were created between them. That happened because the user was trying to connect the lines but his strokes were not precise. This created the need to merge the junctions of the lines that were close to each other. To avoid this, we used *centroid clustering* [48] between the points.



(a) Without clustering (b) With clustering

Figure 4.3: Point clustering to avoid triangles between lines

So, the system uses a *cluster manager* that groups nearby points and calculates a centroid for each cluster. When an intersection is detected, the new point is replaced by the cluster's centroid. This ensures that intersections are kept as single nodes shared by multiple strokes, giving the graph a consistent structure. Although the details of the clustering algorithm are discussed later on, it is important to note that this process prevents overlapping or redundant points but also avoids creating triangles between different strokes.

Constraints

There were several problems encountered while implementing this process that caused the application to not work as desired. That is why, several constraints were added in the clustering for the feature to work as expected. After several informal tests, we found that the optimal threshold for the number of points in each cluster is 5. Adittionally, we have to take into account that the first and the last points of a curve are needed to be able to be a cluster centroid so that other curves can intersect at the end of a stroke. Also, if a point is a junction, it has to stay the centroid of the cluster and not change under any circumstances so that every other line with points close to this one pass connects to the same point.

By enforcing these junction, clustering and beautifying constraints, the system accurately represents how strokes connect in an immersive environment, offering users a more desirable and consistent sketch.

4.3 Edit Drawing

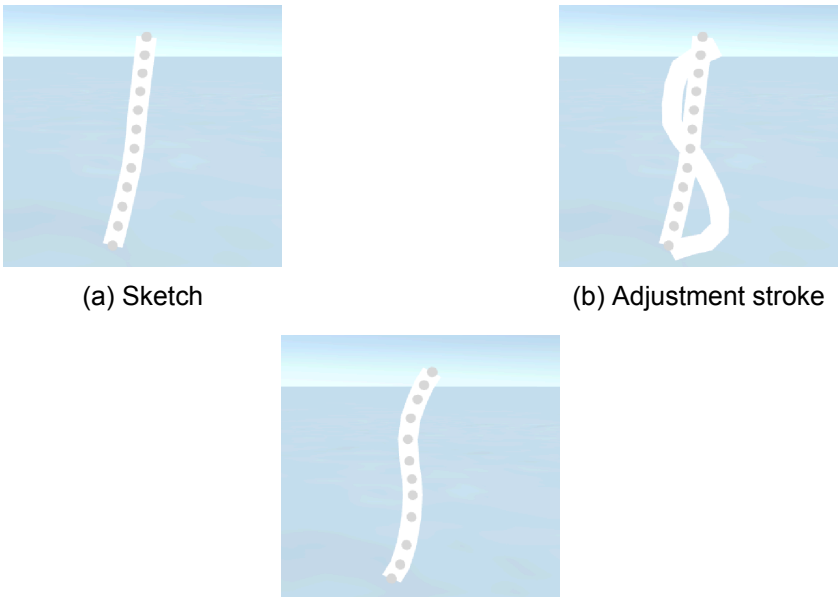
Several tools were included to allow users to modify their sketches by adjusting curves, mirroring strokes, moving points, changing the radius, undoing a line, previewing the mesh and clearing the sketch. The editing tools play a crucial role in the software because we prioritize the accurate shape of the curve since we are focusing on sketching the skeletons of objects. Our goal was to provide users with the ability to edit their strokes and refine them for greater precision.

4.3.1 Additional adjustment curves

The adjustment curves feature allows users to edit their strokes by drawing additional curves over existing ones to change or fix their shape. Instead of treating every new stroke as a separate new line, the system determines whether the new curve should be used as an *adjustment stroke* or a *new stroke* based on its similarity to the existing strokes. This enables users to refine their sketches without the need for manual selection tools.

To implement this functionality, we initially attempted to use line clusters to differentiate between adjustment strokes and new curves in the sketch. However, this approach made it difficult for the system to determine reliably whether a line or a certain percentage of its points belonged to an existing cluster.

After multiple iterations, we chose the KD-tree algorithm to detect the proximity between new and existing lines efficiently. By analyzing whether a significant percentage of the new line points were close to an existing one, we could classify it as an adjustment stroke. This method provided a more precise solution to distinguish between modifications and entirely new curves. This also gives us the flexibility to influence a part of the structure of the curve and not the whole stroke.



(a) Sketch

(b) Adjustment stroke

(c) Final result

Figure 4.4: Adjustment curves

Adjustment stroke detection using KD-Tree

When a user completes a stroke, the system determines whether it is a new stroke or an adjustment stroke by comparing it to previously drawn strokes. The similarity check is based on the spatial proximity of the stroke points to an existing stroke using a *KD-tree*

search for efficient comparisons between neighbors. If a significant portion of the stroke's points are very close to an existing stroke, the system classifies it as an adjustment stroke and refines the existing stroke instead of creating a new one.

To determine whether a stroke is an adjustment stroke, the following criteria must be met:

- The stroke must have a minimum percentage of its points (at least 90%) sufficiently close to an existing stroke.
- The spatial proximity threshold for two strokes to be considered similar is 0.05 units.

Stroke similarity
Given a new stroke represented as a sequence of points $P_{\text{new}} = \{P_1, P_2, \ldots, P_n\}$ and an existing stroke $P_{\text{original}} = \{Q_1, Q_2, \ldots, Q_m\}$, the similarity check is performed by computing the *nearest-neighbor distance*:

$$d_i = \|P_i - Q_j\| \tag{4.2}$$

where:

- d_i is the distance between a point P_i from the new stroke and its closest corresponding point Q_j in the existing stroke.
- The stroke is considered similar if at least 90% of the points satisfy:

$$d_i < 0.05 \tag{4.3}$$

If the similarity criteria are met, the new stroke is classified as an *adjustment stroke*, and the existing stroke is modified instead of adding a new one.

Stroke adjustment process
Once a stroke is classified as an adjustment stroke, it influences the existing stroke to create a new, adjusted version. The final stroke is computed using weighted interpolation between the original stroke and the adjustment stroke:

$$P_i^{\text{adjusted}} = (1 - \lambda) P_i^{\text{original}} + \lambda P_i^{\text{new}} \tag{4.4}$$

where:

- P_i^{original} is the corresponding point from the original stroke.
- P_i^{new} is the corresponding point from the adjustment stroke.
- λ ($0 < \lambda \leq 1$) is the interpolation factor set to 0.5, which ensures the creation of medium points.

Laplacian smoothing
To ensure smooth transitions and natural stroke refinement, the final stroke undergoes *Laplacian smoothing*, which reduces noise and irregularities in the same way it is performed at *Curve beautification* part, defined as in the equation 4.1. The Laplacian smoothing process is limited to *5 iterations*, as in the beautification process.

By integrating stroke similarity detection and weighted interpolation for stroke refinement, the system enables users to edit their sketches in an intuitive and easy way.

4.3.2 Moving a point in a stroke

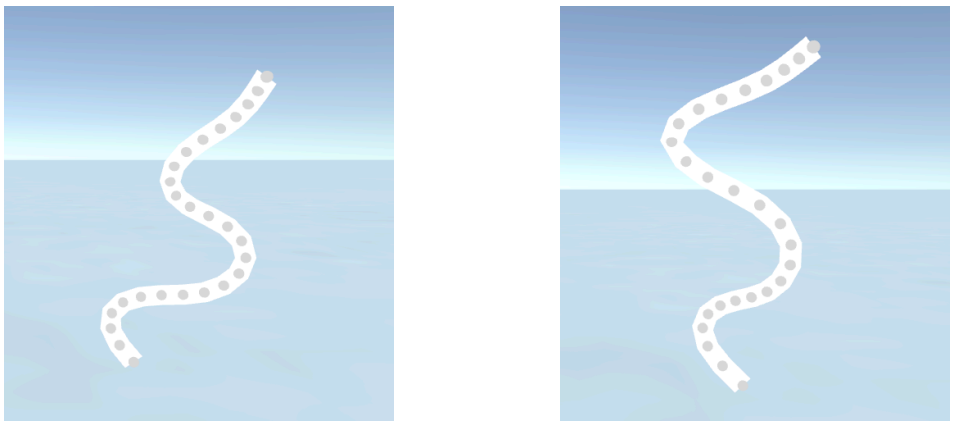
The ability to move points within a stroke allows users to edit their sketches without having to redraw entire lines. This feature enables precise curve adjustments while maintaining smooth transitions between points. The system continuously tracks the user's pointer, detects nearby points, and allows controlled movement while applying interpolation and smoothing techniques to maintain a natural stroke structure.

We explored multiple approaches to achieve the desired effect of smoothly dragging a point while ensuring its neighboring points followed in a natural, spring-like movement, maintaining the overall curve structure without introducing unwanted distortions. Our primary challenge was preserving the general structure of the curve while allowing it to move dynamically to the user's pointer movement.

Initially, we experimented with Laplace smoothing, Bézier curves, and splines. However, these methods recalculated the whole curve at every drag of the user, often resulting in unintended corners, which we wanted to avoid. We then tested various influence factors and hybrid approaches, but none fully met our requirements.

Ultimately, we developed a projection method where points were adjusted in response to the dragged point's movement. This approach had other challenges, particularly in determining the optimal influence distribution. To maintain the structure of the curve, we needed nearby points to be significantly affected while those farther away remained unchanged.

After weeks of research and testing, we arrived at a distance-based weighting function that effectively balances smoothness and responsiveness. This method ensures that the closest points to the dragged one are influenced the most, while the effect gradually diminishes with distance, preserving the overall shape and continuity of the curve.



(a) Sketch (b) Adjusted sketch

Figure 4.5: Moving points

Point selection

To move a point, the system first identifies which point the user is attempting to manipulate. This is done by defining a *selection radius* ϵ around the pointer. A point is considered selectable if its Euclidean distance from the pointer position satisfies:

$$d = ||P_{\text{pointer}} - P_i||, \quad \text{if } d \leq \epsilon, \text{ then } P_i \text{ is selectable} \tag{4.5}$$

where:

- P_{pointer} is the current position of the pointer,
- P_i is the position of a stroke point,
- d is the computed distance between them,
- ϵ is the selection threshold (set to 0.03).

If a point is detected within this threshold, it is highlighted to indicate selection availability. Once the user presses the selection button, the system assigns the nearest point as the active point and identifies the corresponding stroke.

Moving the selected point
Once a point is selected, its position is updated smoothly according to the user's pointer movement. To prevent abrupt position changes, interpolation is applied:

$$P_i^{\text{new}} = (1 - \lambda)P_i + \lambda P_{\text{pointer}} \tag{4.6}$$

where:

- P_i^{new} is the updated position of the point,
- P_i is its original position,
- P_{pointer} is the pointer position,
- λ is the interpolation factor ($0 < \lambda \leq 1$), which controls how smoothly the selected point moves.

Since moving a point affects the overall structure of the stroke, neighboring points must be adjusted accordingly to maintain a natural shape. The displacement ΔP is propagated to surrounding points using a distance-based weight function, ensuring that closer points adjust more than distant ones.

The influence weight w_j for a neighboring point P_j at distance d_j from the selected point is given by:

$$w_j = \begin{cases} 0.95, & \text{if } d_j = 1, \\ 1 - (0.05 d_j), & \text{if } d_j > 1, \end{cases} \tag{4.7}$$

where:

- $d_j = |j - i|$ is the absolute index distance from the selected point,
- w_j determines how much the neighboring point is affected,

Each affected point is updated based on:

$$P_j^{\text{new}} = P_j + w_j \Delta P \tag{4.8}$$

where:

- P_j^{new} is the new position of the neighboring point,
- P_j is its original position before movement,

- ΔP is movement direction vector scaled by user interaction,
- $w_j \Delta P$ applies a scaled movement based on the weight.

This ensures that closer points experience stronger movement, while those farther away retain their original positions with small modifications. This function affects up to 20 points forward and backward in the stroke (if they exist), with weights decreasing based on their distance from the selected point. Beyond that, the influence drops to zero. Once the user releases the point, the system finalizes the stroke modification by updating the data structures.

4.3.3 Mirroring

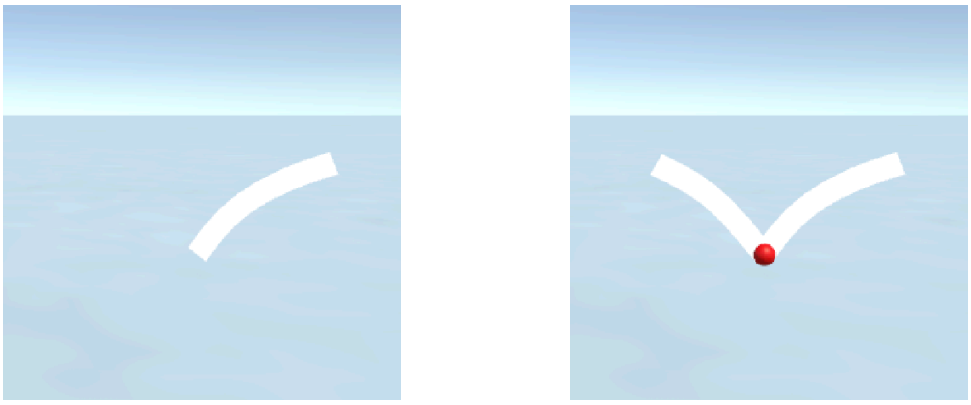
The mirroring functionality allows users to create a symmetrical copy of their last drawn stroke within the VR environment with the press of a button, enhancing usability for tasks that require symmetric designs. This feature ensures that the mirrored stroke maintains the same shape, width, color, and material properties as the original, while being positioned relative to a chosen axis.

To achieve the mirroring effect, the system first identifies the last drawn stroke within the sketch. The points from the original stroke are then reflected across a predefined mirror axis (the YZ plane, which mirrors across the X-axis), ensuring that the mirrored stroke maintains geometric consistency. The mirroring operation reflects each point P_i of the original stroke across the YZ plane. This transformation is defined as:

$$P_i^{\text{mirrored}} = (-x_i, y_i, z_i) \tag{4.9}$$

The variables are defined as: $P_i = (x_i, y_i, z_i)$ represents the position of the original point in 3D space. P_i^{mirrored} is the new position after reflection. The X-coordinate is inverted, while the Y and Z coordinates remain unchanged, ensuring symmetry across the YZ plane.

A small random offset is applied to the mirrored points to prevent exact duplication and ensure smoother intersections when merging strokes. The final mirrored stroke is then assigned to a new LineRenderer (a new line), preserving the visual properties of the original stroke while aligning it within the sketch structure.



(a) Sketch (b) Mirror

Figure 4.6: Mirroring

4.3.4 Adjust point radius

The adjust radius functionality allows users to modify the thickness of individual points in a stroke. It is one of the most important tools of the software since we can change the local thickness of the line and add volume to the generated 3D mesh. This feature enables users to change the stroke widths for more precise sketches. The functionality allows users to select a point and adjust its radius, using grip input and hand movement along the all axis.

Point selection

The system detects the closest stroke point to the pointer, using a proximity selection method within a selection radius in the same way as in *Moving a point in a stroke* (section 4.3.2).

Upon the point selection, a range sphere visualization appears around p_{selected}, indicating the area of influence.

Radius adjustment

Once a point is selected, the radius is adjusted based on the change in the controller's world-space Z position. Although the system is designed to increase or decrease the radius based on forward and backward motion (along the Z-axis), movements along the X or Y axes can also influence the Z value. This is because the controller operates in 3D space, and its orientation with slight movements can cause shifts in the Z coordinate. As a result, the radius may change not only when the controller is pushed forward or pulled backward, but also when it is moved upward, downward, or side to side. The change in radius Δr is computed as:

$$\Delta r = z_t - z_{t-1} \tag{4.10}$$

where:

- Δr is the change in radius,
- z_t and z_{t-1} are the current and previous Z-coordinates of the controller,

To prevent unintentional modifications due to small hand movements, a threshold ϵ (0.0001) is applied:

$$|\Delta r| > \epsilon \Rightarrow \text{Apply radius change} \tag{4.11}$$

Weighting and propagation to adjacent points

To ensure smooth and natural stroke thickness, the radius adjustment is distributed across neighboring points based on predefined influence weights. This approach prevents abrupt transitions and maintains visual consistency in the sketch.

At first, we tried adjusting the radius by directly changing the start and end widths of each line segment using the LineRenderer's AnimationCurve. We calculated the segment positions and increased or decreased their width based on the user's input. However, this caused sudden jumps in thickness between segments, making the stroke look jagged and uneven. Since only the selected segment was updated, nearby points stayed the same, which resulted in sharp and unnatural transitions.

To improve this, we then tried a smoother method where the radius change followed a curve, like a bell shape (similar to a Gaussian). The selected point got the biggest change, while neighboring points were updated with smaller changes depending on their distance. This helped create smoother transitions between segments. However, it also introduced

a new problem, it distorted the overall shape of the stroke. Some parts became too thick or thin, especially if the stroke already had width variations. So even though the changes looked smooth, they no longer matched the original shape.

As a result of the above, we switched to a weighted influence system, where changes spread across multiple points, creating a more natural and visually pleasing effect. The influence of the radius change at a given neighboring point p_i is assigned based on its proximity to the selected point p_{selected}. The weighting function is defined as:

$$w_i = \begin{cases} 0.9, & \text{if } i = p_{\text{selected}} \\ 0.6, & \text{if } |i - p_{\text{selected}}| = 1 \\ 0.3, & \text{if } |i - p_{\text{selected}}| = 2 \\ 0.0, & \text{if } |i - p_{\text{selected}}| \geq 3 \end{cases} \tag{4.12}$$

where:

- w_i represents the weight assigned to point p_i,
- p_{selected} is the index of the selected point,
- i is the point being checked,
- The weights decrease progressively for points further from p_{selected}.

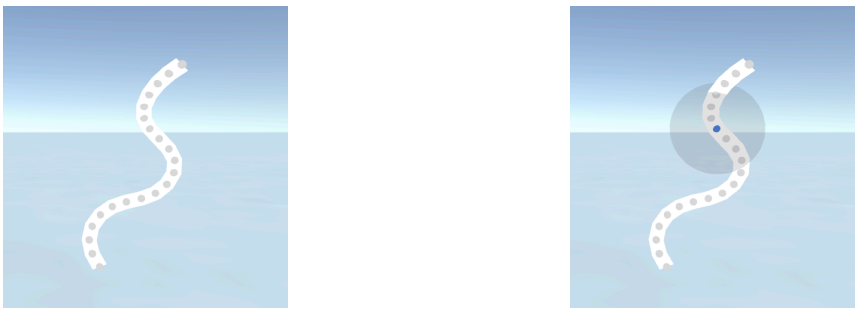
The radius update for each affected point is applied as:

$$r_i^{\text{new}} = r_i^{\text{prev}} + w_i \tag{4.13}$$

This ensures that the most significant change occurs at the selected point while adjacent points receive proportional adjustments based on their proximity. This method enables a visually smooth transition along the stroke, preserving both user intent and aesthetic consistency.

Undo functionality

The system tracks all radius modifications, allowing users to revert unintended changes via an Undo function. If the user triggers an undo command, the system retrieves the last modified line and resets its width to its default value.



(a) Initial skeleton

(b) Range sphere

(c) Adjusted radius

Figure 4.7: Adjust point radius

4.3.5 Additional features

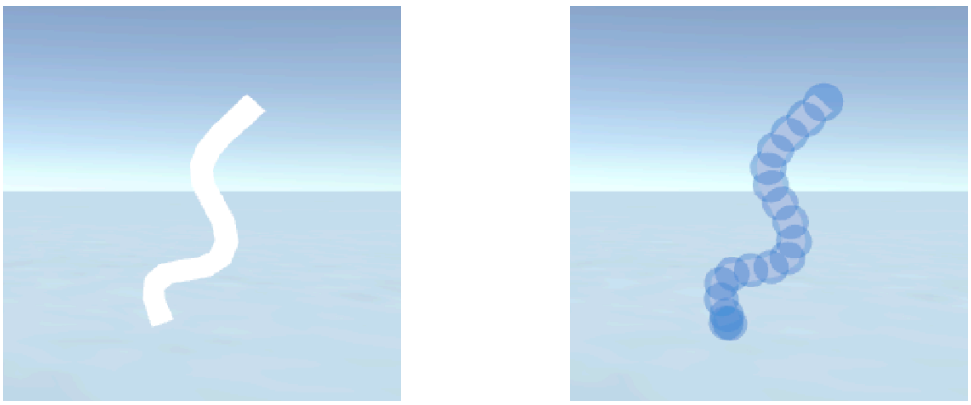
Several additional features have been implemented in the software, including sketch preview, undo, and clear drawing, to enhance the user experience and improve the sketching workflow.

Sketch preview with 3D Spheres

The preview functionality allows users to preview the sketch thickness without generating the final mesh. Similar to Saakes [49], a convenient way to preview a sketch is by rendering the strokes as a series of 3D spheres, each reflecting the local thickness of the line. This preview functionality gives users immediate feedback on any adjustments or movement of points along the sketch like change radius, move points or adjust lines.

In the initial implementation, we chose to recalculate the points of the line and render spheres at very small, precise intervals. This approach produced smooth, visually appealing results, making the sketch resemble a fully rendered mesh model. However, it had a significant drawback, because every time an edit was made (such as moving a point), the continuous re-rendering of spheres at every frame update caused severe lag, leading to a poor user experience and even dizziness in VR.

To address this, we refined our approach by rendering a smaller number of spheres, matching the number of points at the line. This adjustment preserves the overall structure and provides a clear preview of the sketch while maintaining real-time performance, ensuring a smooth and comfortable VR experience.



(a) Sketch (b) Preview with spheres

Figure 4.8: Sketch Preview

When preview mode is disabled, the preview container is removed, along with all the spheres. Thus, the preview can be toggled on and off at any point in the process. Each time the user modifies the sketch (for example moving a stroke's points or adjusting thickness), the preview container is regenerated to mirror these updates in real time.

Undo
The undo functionality allows users to revert specific actions, supporting only undoing sketched lines and radius changes, as these were identified as the most critical interactions for users from the informal tests. This feature enhances the flexibility of the sketching process, allowing users to refine their drawings without starting over. However, it does not support undoing every individual action performed during the sketching process.

When a user activates undo, the system first checks if the last action was a stroke width adjustment. If so, the system resets the stroke width instead of removing the stroke itself. This ensures that small modifications can be undone without affecting the entire drawing. If no width change is detected, the system then removes the most recent stroke from the sketch.

Clear drawing
The Clear Drawing functionality enables users to reset the entire sketch, allowing them to start over with a blank canvas. This feature ensures a quick and efficient way to restart the drawing process without manual deletion of individual elements. The user can clear their sketch from the menu.

4.4 3D mesh generation

The 3D mesh generation tool is one of the most essential features of the software, as it transforms sketch lines into cylindrical structures, adding volume to them and converting them into a 3D mesh. For our VR application, we developed a native library (plugin) for the Meta Quest 3 headset, built from the GEL library and compiled into .so (for Android) and .dylib (for macOS) formats. Unlike APIs that require network access, internet connectivity, or a persistent wired connection to a laptop, our approach ensures that all processing occurs entirely on the headset itself.

This decision was made primarily for performance and usability reasons. First, native libraries allow direct integration with Unity [50] while using the hardware capabilities of the Meta Quest 3. Unlike cloud-based or external APIs, which introduce latency and dependency on external services, our library operates locally, ensuring fast, real time

processing without additional communication. Secondly, native plugins give direct access to system resources and hardware acceleration, which helps with memory management and performance. This is especially useful for demanding tasks like mesh reconstruction.

As a result, for 3D mesh generation from a sketch feature, a plugin was created with the GEL library [38] (see Section 3.2.2). The user can convert the 3D sketch into a 3D mesh with the press of a button.

When we first developed the plugin, we noticed that meshes generated from the VR headset occasionally contained extra edges that did not appear in the sketch. After extensive testing, we determined that this was not due to a bug in the code but rather a difference in how the headset and the laptop processed the data. We confirmed this by observing that the same plugin produced different results when run on the laptop versus the headset using the same graph file. Since we were unable to pinpoint exactly how the headset introduced the extra edges, we implemented an additional preprocessing step before loading the object to remove incorrectly created large edges between points.

(a) Sketch (b) Mesh

Figure 4.9: 3D Mesh

The goal of this feature is to convert point-based sketches into 3D models by extracting their shape while preserving the topology. The first step is to generate a structured graph representation from the input point cloud and then convert the skeleton into a 3D mesh. To achieve this, the following steps are performed:

1. **Graph construction:** A graph is created from the input sketch, where each node represents a point in 3D space and edges define the connectivity between points.
2. **Mesh reconstruction:** The graph is converted into a 3D surface mesh using a reconstruction method (inverse skeletonization). The generated mesh is saved as an `.obj` file, allowing further use in modeling applications.
3. **Loading the 3D model into the scene:** The generated `.obj` file is imported into the scene with necessary adjustments and attached elements, ensuring proper visualization and interaction.
4. **Additional Features:** Wireframe rendering and model saving functionalities are incorporated to support topology inspection and further processing in external modeling applications.

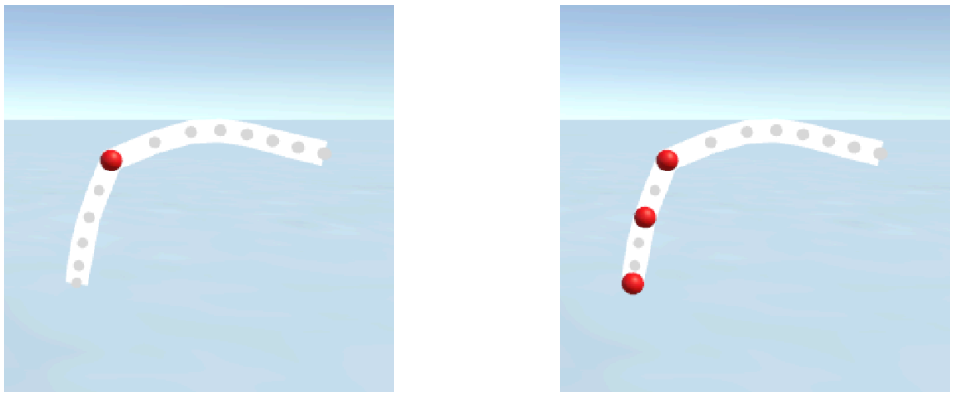
4.4.1 Graph construction

To convert a 3D sketch into a structured representation, we convert it to a graph. The graph consists of:

- **Nodes**: Represent individual points in the sketch, storing their 3D coordinates (x, y, z) and width (radius).
- **Edges**: Define the connections between nodes, ensuring that the sketched strokes are preserved as structured paths.

The process of constructing this graph follows several steps. First, the 3D positions and width of all points in the sketch are extracted. Each point is then assigned a unique index to maintain reference consistency. After defining the nodes, edges are established between consecutive points within the same stroke, ensuring that the sequence and structure of the sketched lines are preserved. Once both the nodes and edges are extracted, they are stored into a single graph file, which serves as the final representation of the sketched strokes.

In some cases, consecutive line points might create multiple junctions over a different line. Usually these junction point have distance 3 or smaller. In these case the software ignores these connections when there are smaller that 3 and the consecutive edges between the points already exist. There extra edges, are presumed by the application as user error (see figure 4.10). This function was found to be very time consuming for a large number of strokes (e.g. 50), when the user tried to generate the mesh and that is why it was deactivated for the user tests.



(a) Wrong edges between junctions (b) Wrong edges between junctions ignored

Figure 4.10: Edges constraint

4.4.2 Plugin integration

To integrate 3D mesh generation into the VR application, we developed a native C++ plugin based on the GEL framework [38]. Since the application is designed for Meta Quest 3, the plugin was compiled for Android, ensuring compatibility with the standalone VR headset. Additionally, for development and testing purposes, the plugin was also compiled for macOS to allow debugging before deploying to the VR device.

The plugin was developed by extending the *"SkeletonizePointCloud"* demo script that utilizes the *inverse skeletonization* technique (see Section 3.2.2) provided by the GEL library [38]. The original demo showcased how to process a set of 3D points into a graph

representation, extract its skeleton, and reconstruct a 3D mesh. The implementation was adapted and extended to meet the specific needs of our application.

To generate a 3D mesh from a point-based sketch, a graph-based approach is used. This method preserves the overall structure of the sketch while allowing for accurate reconstruction.

4.4.3 Loading the 3D model into the scene

The ability to load and display the 3D meshes dynamically is a crucial feature, enabling users to visualize their sketches as tangible objects. This functionality allows users to load the generated 3D model, apply materials, and interact with it within the virtual environment. The process involves reading an OBJ file, generating a mesh, applying materials and physics properties, and optimizing its placement within the scene.

Once loaded, the object is placed in the scene and remains fully interactive. Users can toggle between wireframe mode and the standard material view to inspect the structure of the generated model.

4.4.4 Saving the mesh

The Save Mesh functionality allows users to store their 3D sketches as an `.obj` file for future use in external modeling applications. This feature is essential for saving user's models, sketches, graph files and also metrics enabling further analysis and export to other applications. Users can save the object with the respective button on the Menu. Before saving, a new unique folder is created to organize and store the saved files.

4.5 User Experience

Several UI improvements were made to enhance the user experience and support structured user testing within the software. These changes helped make the system more user-friendly while ensuring all necessary elements for evaluation were in place.

To make drawing in VR more intuitive, we first used a virtual pen attached to the controller. However, after testing, we found that a pointer made drawing easier and more precise, so we replaced the pen with a pointer.

When a line is drawn, we visualize it along with any junctions that may be present. Additionally, if a point on the line is hovered, we display a visual indication, and if the user is adjusting the radius, the radius range is also shown. The same happens to any button available in the interface. Each interaction provides visual feedback, ensuring that users can see when they are hovering over or selecting a setting within the application.

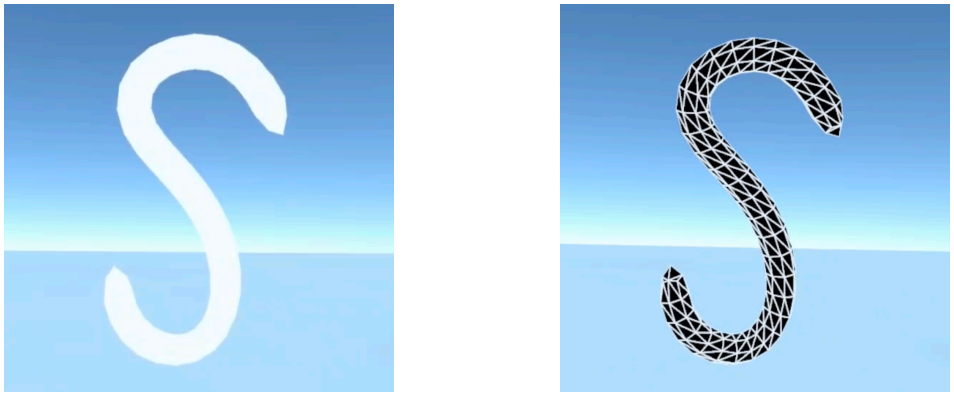
We also added a menu attached to the left controller, allowing users to select settings. The menu can be toggled on and off, and users can interact with it by clicking buttons or selecting them with the right controller. To improve feedback, we included a hover sound effect when users move over menu buttons.

To help users understand their sketches better, we visualized line points and junctions, making it clearer when two strokes are connected. We also added a hover effect when selecting points to make interactions easier. Although, we initially added a sound when hovering over points, we found it distracting for the user, so we removed it.

Finally, we ran informal tests to check how users interacted with the system, ensuring that the controls and movements felt natural. Our goal was to make the interaction as smooth and realistic as possible.

Wireframe

We also added a wireframe visualization feature to allow users to toggle between a solid and wireframe material of the generated 3D mesh. This feature is particularly useful for inspecting the mesh geometry, ensuring the desirable structure of the generated model.



(a) Standard material (b) Wireframe material

Figure 4.11: Wireframe

When activated, the wireframe rendering replaces the standard material of the 3D model with a wireframe shader, which highlights the underlying edges of the mesh without altering its geometry. The system ensures platform specific optimization by applying different materials based on whether the application is running on Android (Quest 3) or macOS (for testing).

User tests

For the user tests, helper labels (Appendix A) were added on the controller to explain the function of each button. Users could toggle these labels on and off as needed. A mini tutorial was also introduced, including short videos and step-by-step instructions for different features and test procedures. To improve readability, a zoom function was also added.

To compare different drawing methods, a separate test scene was created in Unity. This scene featured a Free-hand drawing mode without constraints or editing tools, allowing users to sketch naturally. This helped assess the effectiveness of the developed system compared to traditional Free-hand sketching.

Additionally, custom metrics were implemented to track how users interacted with each tool. This allowed statistical data to be collected for every participant, making it easier to analyze and compare results. These improvements helped ensure a smooth testing process and a detailed evaluation of the system's performance.

5 Implementation

5.1 User workflow

Below, we outline the fundamental user workflow and showcase the capabilities/features this software provides.

- **Curve drawing:** The user creates the sketch of the skeleton by drawing curves.
  - **Curve beautification:** The strokes are automatically adjusted to have more pleasing curves.
  - **Junctions:** Junctions are created automatically between stokes, if they are close enough, to create a continuous design.
- **Sketch editing:** Then they can edit the drawn strokes to make the sketch more precise and add volume to it.
  - **Draw adjustment curves:** The user can draw new strokes close to the ones that have already been drawn to adjust them.
  - **Mirroring:** When a stroke is drawn, it is possible to mirror it.
  - **Move points:** It is possible to choose a point and move it around to adjust the curve.
  - **Adjust radius:** The user is also able to change the radius of points by making them bigger or smaller.
  - **Preview:** At any time, after the first stroke has been created, it is possible to have a sketch preview with 3D Spheres and edit it.
  - **Undo:** It is also possible to undo a curve or the radius adjustments.
  - **Clear drawing:** If the drawing is not pleasing, the user can clear it and start again from scratch.
- **3D model creation:** When the users have finished the design, they can create the 3D mesh from their drawing.
  - **Mesh creation:** Object creation from sketch with a native plugin developed for VR (Android - Meta Quest 3).
  - **Wireframe:** The user can also view the model with a wireframe material.
  - **Move object:** It is possible to move and rotate the object.
  - **Save 3D object:** When the object is created, the user can save the object and he can use it in any software to edit the model. It also saves the initial sketch, the graph file and the metrics.
  - **Clear mesh:** If the result is not pleasing, it is possible to clear the object, edit it and create it again.

5.2 Immersive drawing

5.2.1 Hand-drawing

Within the immersive VR environment, users can freely draw strokes in 3D space to create sketches of skeletons of objects. This drawing functionality is achieved by attaching a virtual pointer (represented as a small 3D model) to the user's hand controller and continuously tracking its position. When the trigger button on the controller is pressed, the system records the pointer's trajectory as a new stroke.

Specifically, each stroke is stored as a collection of points, captured by a Unity `LineRenderer` component. The system checks, in every frame, whether the user's trigger button is held down. If so, new points are added to the stroke as long as the pointer's current position changes significantly from the last recorded point. Once the user releases the button, the stroke is considered complete and the system finalizes the generated `LineRenderer`. During this process, the thickness of the stroke is set by the setting of the variables `startWidth` and `endWidth` parameter in the `LineRenderer`, ensuring that strokes reflect the width of the pointer (`0.01`). Also, each stroke is recorded in a dedicated container (a separate `GameObject`), the user's sketches can be easily manipulated, saved or extended.

5.2.2 Automatic stroke adjustment process

When drawing multiple strokes in VR, it is desirable to identify and merge points from different strokes, that are very close to each other (intersections or junctions). For this process, the KD-Tree algorithm was implemented with multiple constraints to support the desired result.

KD-Tree construction: To efficiently find nearby points, all previously drawn positions are stored in a global list (`allPoints`). Whenever a new stroke is completed, the system either constructs or updates a *KD-Tree* using these points. A KD-Tree is a data structure that allows quick nearest neighbor queries. If no points exist yet, the KD-Tree is left uninitialized.

Threshold-Based intersection: Each new stroke is processed point by point. For each new line point, the system performs a nearest neighbor query in the KD-Tree. If the distance between the new point and its nearest neighbor is below a predefined threshold, the user's new point is considered an *intersection* that should be merged with the existing geometry to create a junction. This threshold (`0.02`), acts as a constraint that dictates whether two points are close enough to be unified.

Clustering and point merging: In addition to merging coincident points, the system uses a *cluster manager* that groups nearby points and calculates a centroid for each cluster. When an intersection is detected, the new point is replaced by the centroid of the cluster. This ensures that junction points are kept as single nodes shared by multiple strokes, giving the graph a consistent structure.

Laplacian smoothing: After merging intersecting points, the system applies a Laplacian smoothing step to the newly created stroke. Points classified as intersection/junction nodes (the ”fixed points”) are excluded from smoothing to preserve their exact positions. By slightly adjusting the position of non-fixed points, the stroke gains a more natural shape while retaining the key structural constraints (junctions, radius, etc.). This smoothing is controlled by parameters such as the smoothing factor (*e.g.*, `0.3`) and the number of iterations (5).

5.2.3 KD-Tree implementation

A key component of the junction detection process is the *KD-Tree* class, which organizes all previously drawn points for rapid nearest neighbor lookups. By structuring the data in a way that halves the search space along different coordinate axes at each level, the KD-Tree greatly accelerates queries such as detecting whether a newly drawn point intersects with existing strokes.

The primary `KDTree` class stores a reference to the root node and uses methods for building, searching, and inserting points:

- Build tree**:** When the KD-Tree is created, it takes a list of points and splits them into smaller parts by picking a different axis (X, Y, or Z) at each level. The point in the middle becomes the current node, and the remaining points are split into two groups, one for the left subtree and one for the right.
- Nearest neighbor search: The method `FindNearestNeighbor` starts at the root and goes down the tree by comparing the query point with the current node along the active axis. It follows the side where the point might be closer, but also checks the other side if it could contain an even better match.
- Insertion: The `Insert` method recursively navigates through the tree just like in a search and adds the new point once it reaches an empty spot. This allows the tree to be updated dynamically while the user draws, without having to rebuild the entire structure.

Whenever the user draws a new stroke:

1. If no KD-Tree exists (no points were previously saved), a fresh KD-Tree is built from the initial set of stroke points.
2. If the KD-Tree already exists, each new point is inserted into the tree. By doing so, the search structure remains valid as the user continues to draw.

When the system needs to determine if a newly placed point intersects with an existing stroke, it queries the KD-Tree for the *nearest neighbor* and compares their distance against a small threshold. If the distance is smaller than this threshold, the system treats the points as a point that already exists or intersecting and merges them into a single entity.

5.2.4 Clustering and point merging

To maintain the 3D sketch representation, the system uses a clustering mechanism that merges close points into shared junctions. This prevents triangles from being created between the lines that should all intersect at the same point. The clustering process dynamically updates during sketching, adapting to new strokes while preserving spatial consistency.

Clustering algorithm

When a new stroke is drawn, each of its points is processed to determine whether it belongs to an existing cluster or if a new cluster should be formed. This is achieved using a clustering threshold r_c (`0.02`), which defines the maximum allowable distance for points to be grouped. Extensive testing was done to adjust this parameter, ensuring a balance between structural consistency and user desirability.

A new point P_i is compared against existing cluster centroids. If the distance to the nearest centroid is within the clustering radius r_c, the point is assigned to that cluster, and the

centroid is recomputed. The clustering algorithm sorts points within each cluster, selecting a representative point (the median) to act as the centroid:

$$C = \text{median}(P_1, P_2, \ldots, P_N) \tag{5.1}$$

If no suitable cluster is found within r_c, a new cluster is created, initializing the centroid with the newly drawn point.

Constraints
When an intersection occurs, the system ensures that the centroid of the cluster remains fixed to preserve the structural integrity of the sketch. If a new point is determined to be an intersection, it replaces the cluster centroid:

$$C = P_{\text{intersection}} \tag{5.2}$$

This prevents the centroid from shifting after an intersection has been identified, ensuring that junctions remain stable. If a newly drawn stroke contains points that overlap with an existing cluster, the system replaces those points with the corresponding cluster centroid. This process prevents the creation of duplicate intersection points and maintains a single unified junction for multiple strokes. This approach ensures that centroids remain close to the most densely populated area of the cluster, minimizing distortions.

5.2.5 Laplacian smoothing

To enhance the quality of the sketched strokes, the system applies *Laplacian smoothing* to reduce abrupt variations in stroke curvature. The smoothing algorithm iteratively refines each stroke while preserving fixed points, such as intersections and cluster centroids.

Given a set of stroke points P_i, Laplacian smoothing adjusts each non-fixed point based on its neighboring points (see equation 4.1). Intersection points (fixed points) are excluded from the smoothing process to maintain the structure. These points are stored in a fixed point list, ensuring that they remain unchanged.

5.3 Edit Drawing

5.3.1 Additional adjustment curves

The adjustment curve implementation enables users to edit their sketches *by redrawing over existing strokes*. This feature is implemented by detecting similar strokes, modifying existing strokes when necessary, and applying *Laplacian smoothing* to ensure smooth results. The implementation consists of three main steps:

1. Detecting similar strokes using a KD-tree.
2. Blending strokes using weighted interpolation.
3. Refining strokes with Laplacian smoothing.
4. Updating the sketch with the adjusted stroke.

Each step ensures that users can refine their sketches dynamically without requiring the selection of different tools.

Detecting similar strokes using a KD-Tree
To determine whether a newly drawn stroke should be classified as an *adjustment stroke*, the system checks its similarity to existing strokes using a *KD-tree nearest neighbor search*. The function `checkSimilarity()` efficiently compares a new stroke to previously drawn strokes.

- KD-tree construction: The system builds a *KD-tree* from the points of previously drawn strokes.
- Nearest-Neighbor search: Each point in the new stroke is checked against the KD-tree to find the closest corresponding point by computing the equation 4.2.
- Proximity threshold: If at least `90%` of the points are within `0.05 units`, the stroke is classified as an *adjustment stroke*.

Blending strokes using weighted interpolation
If a new stroke is classified as an *adjustment stroke*, it is blended with the existing stroke to create a modified version. The functions `CreateMediumLine()` and `CalculateMediumLine()` are responsible for computing the intermediate stroke using point interpolation with the equation 4.4.

- Point interpolation: Each point from the original stroke is influenced by the corresponding point from the adjustment stroke.
- Fixed weighting factor: The interpolation factor `0.5` ensures a new medium stroke, without affecting the junction points.

Refining Strokes with Laplacian Smoothing
After blending the strokes, *Laplacian smoothing* is applied to refine the stroke and remove unnecessary noise. The function `ApplyLaplacianSmoothing()` iteratively smooths the points while preserving important details the same way it is implemented at the beautification process. Each point is updated based on the average of its neighbor points, with a smoothing coefficient of `0.2` and it runs for `5 iterations`. Additionally, each point is evaluated to determine if it is a junction point, ensuring that junctions remain unchanged during the process.

Updating the sketch with the adjusted stroke
Once the modified stroke is created, the system updates the sketch data structures to ensure that future strokes are processed correctly. The function `AddLineToDataStructures()` adds the new adjusted line's points to the datastructures (clusters and KD-Tree).

5.3.2 Move point in stroke
The ability to modify strokes by moving individual points allows users to edit their sketches without redrawing entire strokes. The system ensures smooth adjustments by projecting movement to nearby points. The implementation consists of three main steps:

1. Detecting and selecting a stroke point.
2. Updating the selected point based on pointer movement.
3. Adjusting neighboring points using projection.
4. Updating the sketch with modified stroke.

Each step ensures that the stroke remains visually continuous while allowing for precise modifications.

Detecting and selecting a stroke point
A stroke point is selected by detecting the closest point within a predefined selection radius ϵ of the user's pointer. The function `FindClosestPoint()` determines the nearest point using Euclidean distance:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2} \tag{5.3}$$

where:

- d is the distance between the pointer position P_{pointer} and the stroke point P_i.
- If $d \leq \epsilon$ (where $\epsilon = 0.03$), the point is selectable.

Once a valid point is found, the function `TrySelectPoint()` assigns it as the active point and retrieves the corresponding stroke.

- Point selection: The system checks all stroke points within the selection radius.
- Highlighting: A visual cue is applied to indicate the selected point.
- Controller input: Selection is confirmed when the user presses the interaction button.

Updating the selected point based on pointer movement
When a point is selected, its position is updated to match the movement of the user's pointer. To ensure smooth transitions, interpolation is applied that is shown in equation 4.6. The function `MoveSelectedPoint()` ensures that the stroke point follows the pointer in a controlled manner without abrupt jumps. The position of the point is adjusted continuously based on the pointer.

Adjusting neighboring points using projection
To maintain stroke continuity, the displacement of the selected point is applied to surrounding points using distance-weighted projection. The function `ApplyProjectionMovement()` calculates the influence of movement on each neighboring point based on its distance. The weight function used to determine the influence is shown in equation 4.7. The new position of each neighboring point is computed as shown in equation 4.8, while junction points remain unchanged.

Updating the sketch with modified stroke
Once the point is moved, the system updates the sketch by processing the modified stroke and removing outdated data. The function `UpdateAfterEdit()` performs the following steps:

1. Removing the previous stroke: The original stroke is removed from the KD-tree and clusters. The modified stroke is reprocessed for fast searching and point groupings are adjusted based on modifications.
2. Adding the modified stroke: The modified stroke is inserted into the KD-tree for future junction detection, and the clustering system updates the stroke structure.
3. Updating visualization: If the sketch is in preview mode, it is refreshed to reflect the changes.

5.3.3 Mirror a line

For the implementation of mirroring, the system begins by detecting the last drawn stroke within the sketch container. If a stroke is available, its corresponding line data is extracted, and the mirroring process is applied.

To generate the mirrored stroke, the algorithm reflects each point P_i of the original stroke across the YZ plane using the transformation described in equation 4.9.

This operation ensures that the stroke is properly reflected without altering its original structure. Additionally, to avoid exact duplication, a small random offset is introduced to each mirrored point:

$$P_i^{\text{final}} = P_i^{\text{mirrored}} + \tag{5.4}$$

where is a random deviation vector ensuring slight differences, which helps prevent as possible unintended intersections when strokes are merged.

After computing the mirrored points, a new `LineRenderer` instance is created to store and display the mirrored stroke. The system transfers key visual attributes from the original stroke, including stroke width and its variation along the line, material properties, and stroke color.

Furthermore, the mirrored stroke is repositioned to align with the original stroke's endpoint. This is achieved by computing an offset vector between the original stroke's starting position and the mirrored stroke's new starting position:

$$\Delta_{\text{offset}} = P_0^{\text{original}} - P_0^{\text{mirrored}} \tag{5.5}$$

This offset is applied to all mirrored points, ensuring correct placement relative to the original stroke. The final mirrored stroke is tagged differently within the scene to prevent additional mirroring of the same stroke.

5.3.4 Point radius adjustment

The Adjust Radius functionality was implemented using Unity's XR Input System, using the grip input from the right controller to dynamically modify stroke thickness. The implementation consists of four primary steps: point selection, controller-based radius adjustment, smooth propagation, and undo functionality.

Point selection and hovering

To enable intuitive selection, the system implements a point detection algorithm that identifies the closest point within a defined selection radius. The `FindClosestPoint()` function determines the nearest valid point from the user's pointer position. When a point is hovered, its material is temporarily changed to a highlighted color for visual feedback.

Radius modification

Once the closest point is identified, the system listens for a grip button press to confirm selection. The selected point is locked to ensure controlled modifications. Additionally, the system determines a symmetric range of neighboring points to be affected by the radius adjustment:

$$R_{\text{start}} = \max(P_{\text{selected}} - 3, 0), \quad R_{\text{end}} = \min(P_{\text{selected}} + 3, \text{Line Length}) \tag{5.6}$$

where:

- R_{start} and R_{end} define the range of influence.
- P_{selected} is the index of the selected point.
- The adjustment extends up to three neighboring points for smooth transitions.

To adjust the radius, the system tracks movement along the Z axis of the right-hand controller. If the controller is moved forward, the segment thickness decreases, and if moved backward, the thickness increases. To create a smooth transition, the radius change is applied gradually to surrounding points using a weighted influence model. The system assigns different weights based on proximity to the selected point (as described in Section 4.3.4).

Visualizing the adjustment range

To provide visual feedback, the system generates a range sphere around the selected point. This helps users understand the area being affected. The range sphere is dynamically updated based on user interactions and disappears once the adjustment is complete.

5.3.5 Additional tools

Sketch preview with 3D spheres

This preview functionality gives users immediate feedback on any adjustments or movement of points along the sketch. When the preview mode is activated, the system creates a dedicated *preview container* for the 3D spheres corresponding to every vertex in the sketched lines.

For each line in the sketch, the algorithm retrieves the individual points of each line (via `LineRenderer.positionCount` and `LineRenderer.GetPositions()`). A sphere is created at each point, sized according to the stroke's width at that position. Specifically, the width is determined by the line's `widthCurve`.

Undo functionality

The Undo functionality enables users to revert the last drawn stroke. The implementation ensures that either the most recently sketched line or the last modified stroke width can be undone. For more implementation details, see Appendix D.

Clear drawing

The Clear Data functionality is designed to reset the sketching environment by removing all previously drawn strokes, points, junctions and resetting the data structures. For more information, see Appendix D.

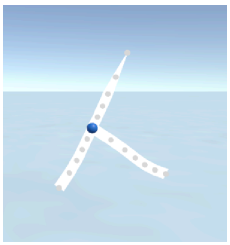
5.4 Mesh generation from drawing

The final stage in the user workflow involves converting the sketched lines into a 3D model 5.1. When the user finishes drawing, he can create the mesh by pressing a single button, while triggering the *"Inverse skeletonization"* [36] process.

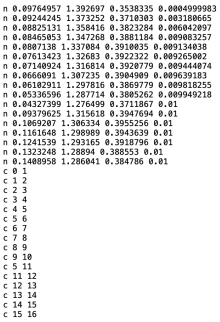
The system executes multiple steps to produce and load the resulting object:

- **Graph file creation:** A graph file (point cloud) is created by the lines the user draw, the junction between the lines and the radii of each point and saved s an input file.
- **Plugin integration:** The graph file created is used as input into the native plugin which processes the data and exports the final `.obj` model.
- **Object loading:** The newly generated `.obj` file is then imported into the scene, where various adjustments and interactions become available to the user.

In the following sections, we examine the creation and operation of the native plugin, detailing how it uses the graph file and converts it into a rendered mesh.



(a) Sketch



(b) Graph file

(c) 3D object

Figure 5.1: Mesh creation

5.4.1 Graph file

The first step in the workflow is converting user-drawn strokes into a graph-based representation Listing 5.1. As the user sketches lines (represented by `LineRenderer` components in Unity), each point along the stroke is saved as a node, complete with its 3D coordinates and a radius that reflects the stroke's width at that point. When consecutive points in the same stroke are processed, they form edges in the graph, indicated by pairs of node indices. If a specific point is shared by multiple strokes, this point naturally creates a junction (an intersection) in the graph, meaning multiple edges are connected with one node.

Through this process, we generate a graph file Listing 5.1 with nodes (containing 3D position and radius) and edges (representing connectivity between consecutive or intersecting stroke points). Below is an example illustrating the general structure of the resulting file.

- Each line beginning with `n` describes a node's position (x, y, z) and radius.
- The lines beginning with `c` define an edge between two node indices.

```
n 0.3107527 0.8971981 0.3048117 0.01953936
n 0.2994019 0.8936143 0.3076782 0.01854726
n 0.28791 0.8927209 0.3103042 0.0161815
n 0.2758117 0.894496 0.3130808 0.01335785
n 0.2616969 0.8994396 0.3170415 0.01099209
c 0 1
c 1 2
c 2 3
c 3 4
```

Listing 5.1: Simple Graph file

These files are then used as input files in the developed plugin that creates the 3D object. By capturing position, connectivity and radius information, we preserve both the geometry and thickness of sketched lines for further processing in the plugin.

5.4.2 Mesh generation from sketch using a C++ plugin

To convert the drawn sketch into a 3D mesh, a custom C++ plugin was developed using the GEL Library [38]. This plugin processes the point cloud representation of the sketch, extracts a skeleton graph, and reconstructs a 3D mesh using the skeleton to mesh pipeline.

For the compilation of the C++ plugin across different platforms, *CMake* was used as the primary build system. CMake provides a platform-independent approach to managing dependencies, generating build files, and linking external libraries. The CMake configuration specifies the required source files, compiler settings, and dependencies.

The processing workflow consists of the following steps:

1. **Graph construction:** The system first loads the set of points extracted from the user's sketch and represents it as a graph structure, where each point is treated as a node, and edges define their connectivity.
2. **Skeletonization:** The graph is then processed to extract a skeleton representation of the shape, simplifying the point cloud into a structured form.
3. **Radii filtering:** A median filter is applied to smooth the node radii and ensure the radii change for each point.
4. **Mesh reconstruction:** Finally, the processed skeleton is converted into a 3D mesh using the FEQ method (see Section 3.2.2), by converting the skeleton into a polygonal structure.

The mesh processing is implemented in a C++ function, exposed as a shared library for integration with Unity. The function takes an input file containing the point cloud representation (graph file) and generates an output mesh in `.obj` format.

5.4.3 Android NDK integration

To enable mesh generation on Android, the C++ plugin was compiled as a shared library using the Android NDK [51]. This allows the function to be called from Unity's to call the C++ function through a native plugin, enabling efficient execution on ARM-based mobile devices, from C# scripts, making it possible to process sketches and generate meshes directly on the VR headset. The script was based on the demo *SkeletonizePointCloud* script provided by the Library. For more information about the *Application.mk* and *Android.mk* files see Appendix D.

Once the shared library is compiled, it is placed inside the Unity project's `Plugins/Android` folder, where it is automatically included in the Android build. The mesh generation process is triggered by calling the Android plugin through a C# script.

This integration enables real time mesh generation directly on Android VR devices (Quest 3), without requiring external servers or desktop processing. For details on how the GEL and PyGEL libraries were compiled into a shared plugin for Android, see Appendix D.

5.4.4 Load object file into the scene

To implement the OBJ file importing and loading, the system utilizes the *Dummiesman OBJ Loader* [1] to parse and render OBJ files directly in Unity. The system supports loading

[1] https://assetstore.unity.com/packages/tools/modeling/runtime-obj-importer-49547

OBJ files, ensuring compatibility across different platforms, including macOS and Android (Meta Quest 3) for testing purposes.

This process extracts the mesh data and constructs a corresponding 3D object within Unity. Upon import, the system assigns a *MeshFilter* to store the mesh structure, allowing further modifications such as transformation adjustments and material application.

To maintain the position of the 3D model within the VR environment, the system centers the model pivot and places it at a predefined position in front of the user. This is necessary because many OBJ files have arbitrary pivot positions that may cause misalignment when placed in a VR environment. The mesh is first analyzed to determine its bounding box, and the transformation is adjusted so that the object's center aligns with the scene's origin.

Once the object is loaded, a material is applied for visualization. Additionally, a wireframe rendering mode is provided, allowing users to toggle between solid rendering and a wireframe view for better structural inspection.

Physics and interaction components
To enable interaction within the VR environment, the loaded model is assigned:

- A BoxCollider, allowing for physical interactions and collisions.
- A Rigidbody set to kinematic mode, ensuring that the object remains stable while being interactable.
- An XRGrabInteractable component, enabling users to grab and manipulate the object.

5.5 VR application - User interface

5.5.1 Frameworks and Technologies

The development of the VR software involved the utilization of several key frameworks and technologies. Unity 3D [50] and several packages were used that can be found at Appendix C. The primary frameworks used were XR Interaction Toolkit and Oculus XR Plugin provided by Unity. The software was created for Meta Quest 3 headset and controllers.

5.5.2 User interaction

The user can interact with the software through the controllers, the menu and the tutorial provided. As shown in Appendix A, the controller mappings provide an overview of the button functions. Editing features are triggered by a button on the controllers, the extra features are visible on the menu of the software and triggered by buttons 5.1.

5.5.3 User interface

For many UI elements of the app the Unity VR template [52] was used, such as buttons, menu panel, sounds, controller labels and visualization.

There is also a menu as shown in Figure A.2 that is attached to the left controller and can be toggled ON/OFF. We can interact with it by pushing a button with the right controller or by choosing it with the respected button. At the menu there is also a log box, where is shown everything that is happening on the app. Additionally there are helper labels (see Appendix A) on the different buttons used at the controllers to help user get familiar with them. These helper labels can also be toggled ON/OFF.

There is a tutorial panel that has different tabs for each different feature of the software but also with information about the test. Each tab provides instructions and a video with the feature.

Controller	Trigger Actions
Left Controller	Menu ON / OFF Helper labels ON / OFF
Right Controller	Draw Undo Move line points Change radius Mirror a line UI click / interaction
Menu	Clear sketch Clear mesh Generate mesh Preview sketch View wireframe Save mesh View Log

Table 5.1: Meta Quest 3 Controller Triggers

For the test User Tests, a new scene was created in Unity with a basic free-hand drawing technique without any constraints and editing tools except undo. There was also added a simpler Menu (see Appendix A) with only save sketch and move to the new drawing technique.

5.5.4 Wireframe

The wireframe effect is achieved through a shader provided by Unity, the *VR/SpatialMapping/Wireframe*. When `isWireframe` is enabled, the material is switched to either a standard wireframe material or a VR-optimized wireframe variant (`VRWireframe`), depending on the platform. This simple toggle workflow helps users to assess the structure of the model to visualize the geometry of a 3D object in VR.

6 Tests and Results

6.1 Early informal tests

Two early informal test iterations were conducted to evaluate the initial functionality of the system and identify usability issues before introducing more advanced features. These iterations focused on improving interaction design and interface clarity.

1. Early test iteration

The first phase of testing was performed before implementing any editing tools. The primary objective was to refine the basic interaction mechanics and ensure smooth functionality. The following improvements were made:

- Undo functionality was introduced, allowing users to reverse unintended actions, enhancing flexibility and control.
- The pen tool (that was attached to the controller) was replaced with a pointer, making the selection and interaction more intuitive.
- Bugs related to junction detection were identified and fixed, improving the accuracy of stroke intersections.

2. Early test iteration

In the second iteration, changes were made based on observations from the initial testing phase, focusing on usability and interface improvements. Key modifications included:

- Redesigning the menu system, introducing a single button for creating meshes (instead of two buttons for creating and viewing the mesh), simplifying the workflow and also including clearing of the mesh before creating a new one.
- Moving the menu panel to the left controller and attaching it to it.
- Enhancing the UI for line points, as the previous hover effect was not sufficiently noticeable, leading to interaction challenges.
- Removing the sound when hovering over points as it was found annoying by the users instead of useful.
- Adding a mini tutorial, providing users with guidance on system features and interaction methods to reduce the learning curve.

These informal tests were crucial in iterating on the design, ensuring that fundamental interactions were intuitive, and addressing early usability challenges before proceeding to more formal user evaluations.

6.2 Experienced user evaluation

As part of the evaluation, we will present a series of sketches and corresponding 3D meshes created by an experienced user. This evaluation aims to assess and showcase the effectiveness of the system in supporting 3D sketching.

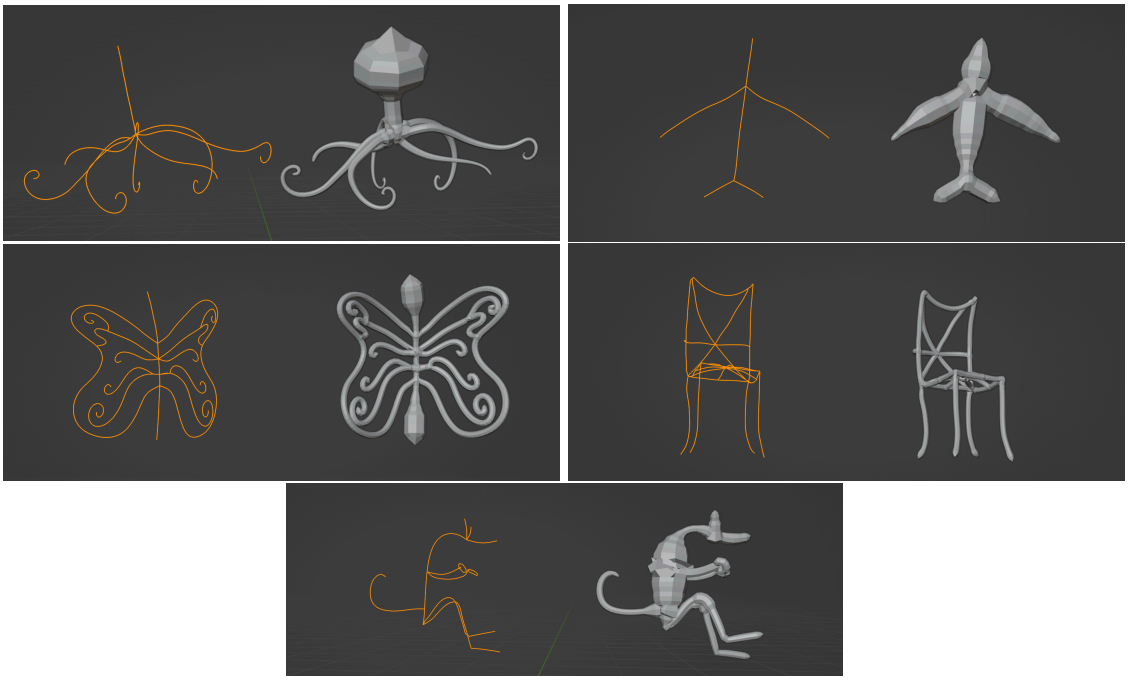


Figure 6.1: 3D sketches/meshes created by an experienced user

6.3 Formal user study

6.3.1 Test plan

Below we analyze the formal user test plan.

- **Immersive hand-drawn sketching in VR**: We will assess how intuitive and efficient it is for users to draw 3D strokes in VR. Participants will complete freehand sketches and use assistive features such as automatic stroke connection and mirroring. Metrics will be collected that will include: task completion time, stokes drawn and user satisfaction.
- **Editing and refining drawings**: We will evaluate the effectiveness of editing tools such as moving points, adjusting curve radius, and mirroring. Participants will have to draw an initial sketch and then modify it using editing tools. Metrics will be collected that will include: number of times feature was used and user preference.
- **User experience evaluation**: We will assess the ease of learning and usability of the VR modeling system. Participants will complete tasks while their performance is tracked over multiple trials. We will collect feedback and observations will be analyzed.
- **System performance**: The efficiency of the system will be assessed by measuring processing time, frame rates (FPS) and CPU/GPU load when creating and editing 3D sketches.

6.3.2 Test steps

The following steps outline the structured procedure for evaluating the VR 3D modeling system:

- **Step 1: Comparison of our technique with Free-hand Drawing** Participants will first create a sketch of their own in *Free-Hand Drawing mode*, and then recreate the same sketch in the new prototype *Skeleton Design mode* software. This step is designed to compare drawing ease, accuracy and efficiency between the two systems.
  - They first create a sketch with Free-Hand drawing mode.
  - They then replicate the same sketch with our new mode system. They can also edit the sketch with the tools provided.
  - Then they will save their sketch along with some Metrics.
- **Step 2: Drawing in the new skeleton design mode and editing/refining Drawings** Participants will be given one new reference sketch and asked to recreate it using the new skeleton design mode. Participants will modify their sketch using the editing tools available in the software. This step assesses the effectiveness of tools such as moving points, adjusting curve radius, and mirroring. This will evaluate how effective the editing tools are.
  - Participants are given one 2D image as a reference (an octopus).
  - They will sketch the image in the VR software, using skeleton design mode and assistive features.
  - Participants will edit the sketch they created. They will use tools such as move, mirror, adjust point radius and curve adjustment etc.
  - Then they will save their drawing, mesh and metrics that contain the number of corrections made (number of strokes, undo, medium lines, point adjustments, radius adjustments), as well as task completion time.
- **Step 3: Assessment - Questionnaire** A usability evaluation will be conducted to assess the learning curve of the VR system and feature usefulness.
  - Participants will answer a post-experiment questionnaire (see Appendix B).
  - Metrics such as task completion time and tools used will be analyzed to determine learning progress.
- **Step 4: System Performance Benchmarking** The final step involves measuring the performance of the system by analyzing processing time, FPS, and CPU/GPU load. We can monitor the performance with Meta Quest Developer Hub (MQDH).
  - Performance variations will be recorded to ensure system efficiency.

6.4 Result analysis

6.4.1 Questionnaire analysis

We present the results of the user study that was conducted to evaluate the *VR skeleton stroke based drawing system for 3D model creation*. The data was collected through a questionnaire (Appendix B), and the results are analyzed both qualitatively and quantitatively. The questionnaire was divided into four sections: participant background, com-

parison between Free-Hand Sketch and Skeleton Design mode, drawing experience in Skeleton Design mode, and overall user satisfaction.

Participant background
The system was tested with 10 users, both male and female participants. Four users had some prior experience with modeling software, while the remaining six users had no previous experience and had never used modeling software before. Also, *none* of the participants had previously used a VR headset or application.

Comparison between Free-Hand drawing and Skeleton design
The results demonstrate that the Skeleton Design mode was generally slightly preferred over the Free-Hand Sketching mode in of ease of use, accuracy, and efficiency. Participants rated both modes on a 1-5 scale. The detailed numerical results are presented in Table 6.1.

Mode	Average ease of use	Average accuracy	Average efficiency
Free-hand drawing	3.4	2.2	3.0
Skeleton design	4.1	3.6	3.7

Table 6.1: Sketching in different modes

According to the results, 70% of users found the Skeleton Design mode easier to use compared to the Free-Hand Sketch mode, though the difference was not substantial. The Skeleton Design mode received an average ease of use rating of 4.1, whereas the Free-Hand mode was rated slightly lower at 3.4. More importantly, accuracy was significantly improved in the Skeleton Design mode, which achieved an average accuracy rating of 3.6, compared to 2.2 for Free-Hand sketching.

Additionally, efficiency was also higher in the Skeleton Design mode, with an average rating of 3.7, compared to 3.0 for Free-Hand sketching. This indicates that users not only found it easier to create sketches with the Skeleton Design approach but were also able to complete them more quickly.

The differences in ease of use, accuracy, and efficiency are also visualized in the box plots (see Figures 6.2, 6.3 and 6.4). These box plots demonstrate that the Skeleton Design mode had better results than Free-Hand drawing in general.

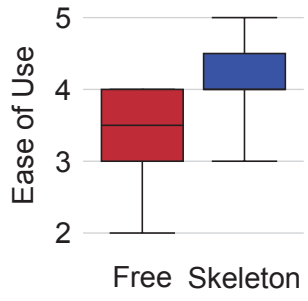


Figure 6.2: Ease of Use

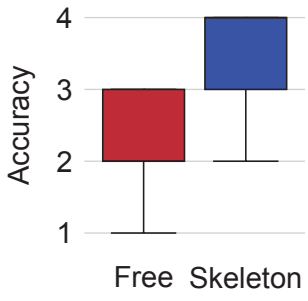


Figure 6.3: Accuracy

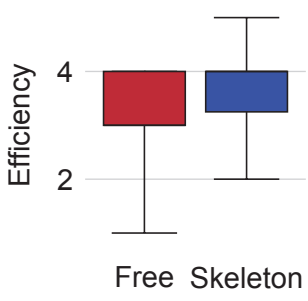


Figure 6.4: Efficiency

Below, we present the sketches and meshes created by users in the Skeleton Design mode. Participants were given the freedom to draw any structure they wanted, with examples such as trees suggested to guide them.

Figure 6.5: User sketches and meshes

Drawing experience in Skeleton Design

The Skeleton Design mode was evaluated based on its intuitiveness and the ease of editing sketches. As shown in Table 6.2, the intuitiveness of the sketching tools received an average rating of 4.0, indicating that users generally found the system easy to understand and use. This shows that the tools for drawing and editing strokes did not require significant learning effort.

Regarding the easiness of editing and refining sketches, users rated the experience at 3.5 on average. This includes all the tools provided by the software. While users appreciated the ability to modify their sketches, some found certain editing operations to be slightly complex.

Aspect	Average rating
Intuitiveness of sketching tools	4.0
Ease of editing and refining sketches	3.5

Table 6.2: User experience ratings in Skeleton Design mode

The users were then asked to choose one tool as the most useful and one as the least useful (see Figure 6.6). Based on their feedback, the most useful tool was the Move Points feature, which was selected as the most useful by 40% of participants. The second most appreciated features were Generating the Mesh and Changing the Radius, both of which were chosen by 20% of users. These results suggest that users found the ability to manipulate the stroke width and convert sketches into structured 3D models particularly beneficial. Drawing multiple strokes to influence the curve and Mirroring, were also chosen as most useful by 10% of users. Preview, and Wireframe, were not selected by anyone.

The bar plot (see Figure 6.6) visualizes these results, demonstrating the distribution of user preferences across the available tools. The data indicate that interactive sketch modification features, such as moving points and adjusting radius, played a crucial role in improving the user experience.

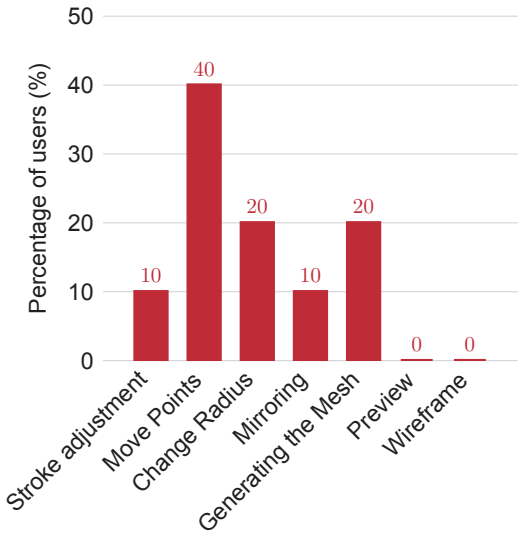


Figure 6.6: Most useful features

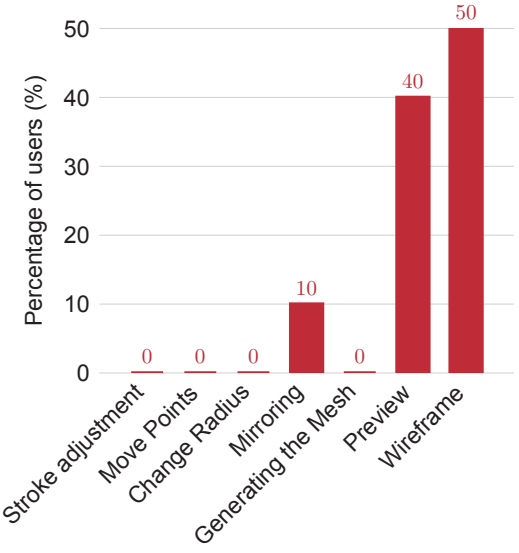


Figure 6.7: Least useful features

According to the results, the Wireframe mode was rated as the least useful feature by 50% of users, followed by the Preview function, which 40% of users found unhelpful or confusing (see Figure 6.7). The Wireframe and the Preview tools were seen as redundant by many participants. Users reported that the system's existing visualization methods were already sufficient for evaluating their sketches, making the preview tool unnecessary. Mirroring tool was also considered less useful, with 10% of users rating it as confusing or unnecessary.

The bar plot (see Figure 6.7) visualizes these results. The data indicate that preview and wireframe tools were not useful to users, especially to novice users. Future improvements should focus on changing these tools or the way the user interacts with them.

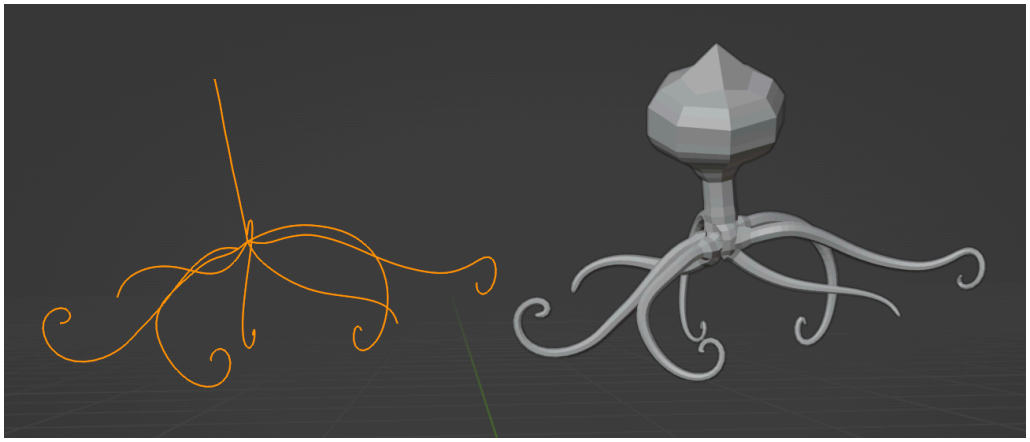


Figure 6.8: Reference image

Below, we present the sketches and meshes created by users based on a reference model. Participants were provided with an octopus image (see Figure 6.8), generated using the software, and were asked to create an octopus with the Skeleton Design system.

Figure 6.9: User sketches and meshes

User satisfaction

Participants provided feedback regarding the ease of learning and enjoyment while using the system. Based on the user responses, the new VR system was generally well received in terms of ease of learning and enjoyment (see Figures 6.10 and 6.11). The ease of learning received an average score of 4.2, with most users rating it 4 or higher, indicating that the system was intuitive and required little time to grasp. However, a small variation in responses suggests that a few participants needed some time to get used to it (see Table 6.3).

Time required	Percentage of users
Immediately	30%
After a few minutes	40%
After multiple attempts	30%
Still not comfortable	0%

Table 6.3: Time to feel comfortable with the new system

When asked about their overall enjoyment of the system, 100% of users enjoyed using the application. the system scored exceptionally high, with an average rating of 4.9. The majority of participants rated their experience as 5.0, suggesting a highly engaging and satisfying interaction with the VR application. Additionally, all users (100%) agreed that the user is beginner-friendly and none of the users experienced any lag while using the system.

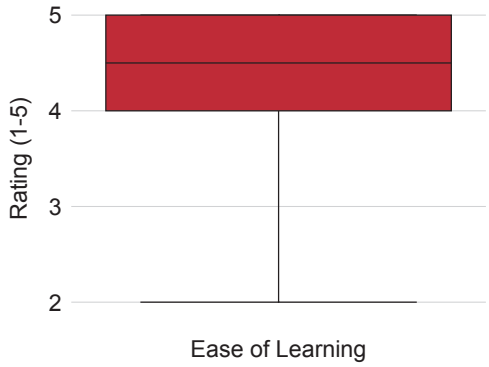


Figure 6.10: Ease of learning

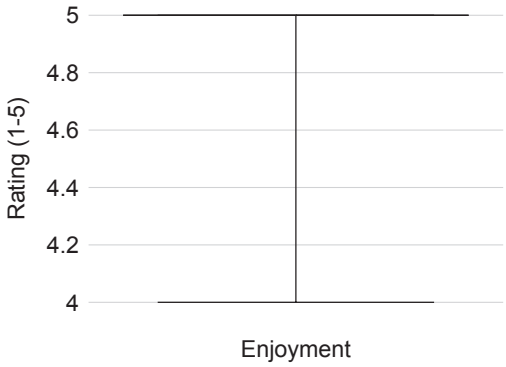


Figure 6.11: Enjoyment

Participants were asked to give feedback on what they liked most about the application, what they found challenging, and which features they would like to see in future iterations. The responses highlighted several key points.

Many users appreciated the ability to refine sketches after drawing. Most of the users described the application as easy to learn and use, with novice users appreciating the instructional video explaining the controls. *Move points* tool was particularly well received, as it allowed adjustments without having to redraw entire strokes. The *mirroring* feature was found useful, for symmetrical designs like the octopus. Several participants enjoyed the ability to generate a 3D mesh from their sketches. The automatic connection of strokes was highlighted as a major improvement over Free-Hand drawing, making sketches more structured.

Despite the overall positive feedback, some challenges were noted. The *mirroring* feature did not work as expected for some users, making it difficult to control their strokes. The *undo* function was found to be limited, as users expected it to reverse all types of modifications, including small changes. Some users experienced difficulties when trying to draw small strokes, as they would sometimes unintentionally modify nearby existing lines because of the *adjust a curve* tool. The generated mesh was not smooth in some areas while changing the radius, which some users found problematic. A few participants noted that they wanted more control over deleting individual strokes rather than relying solely on the undo feature.

Participants also suggested improvements for future versions of the application:

- Allow users to select and delete individual strokes instead of relying solely on undo.
- Improve the *mirroring* tool by allowing users to define the mirroring plane for better control.
- Provide an option to control the stroke width while drawing.
- Enhance the *change radius* tool by adding an adjustable range setting.
- Introduce different drawing methods, such as the ability to create straight lines.
- Add an option to be able to apply different colors to specific sketch parts.
- Implement a feature that allows the users to import reference images into the VR environment to guide their sketches.

6.4.2 Metrics analysis

Every time a user saved a sketch at the test process, a *Metrics file* was also saved that had several tool elements with a number of times used next to each element, such as Sketch Time, Sketched lines drawn, Undo lines drawn, Medium lines drawn, Preview times button pressed, Moved a point, Changed Radius, Generated Mesh, etc.

Analysis comparison between the two modes

To compare the two sketching modes, we analyze the first sketch of the users. The longer sketching time observed in Skeleton Design Mode (see Table 6.4) is expected, as users could edit their sketches with the available tools.

Metric	Free-Hand Mode (Avg.)	Skeleton Design Mode (Avg.)
Sketch Time (s)	276.1	367.6
Sketched Lines Drawn	75.2	63.2
Time per Stroke (s)	4.31	8.92
Undo Lines Drawn	34.2	27.1
Medium Lines Drawn	-	18.5

Table 6.4: Comparison of sketching metrics

The Skeleton Design mode took significantly longer (367.6 seconds) on average compared to the Free-Hand mode (276.1 seconds). This increase in time suggests that the structured approach in Skeleton Design mode requires more precision but also because it offers multiple editing tools that allow users to refine their sketches without needing to restart. Additionally, the Free-Hand mode allows for a rapid drawing process, making it best for initial ideation and exploration.

Users sketched an average of 75.2 lines in Free-Hand Mode, whereas 63.2 lines were drawn in Skeleton Design mode because users refined their sketches through repeated strokes in the first mode. In contrast, in Skeleton Design mode there was a lower number of strokes, that also some of them were adjusting strokes and not new strokes of the sketch. The average time per stroke in Skeleton Design mode (8.92 seconds) was more than double that of Free-Hand mode (4.31 seconds).

The number of undo operations was also higher in Free-Hand mode (34.2) compared to Skeleton Design mode (27.1). This indicates that Free-Hand sketching is more prone to frequent deletion, likely due to the lack of editing tools, compared to the Skeleton Design mode.

The analysis reveals a difference between efficiency and precision in the two modes. This comparison suggests that Free-Hand mode may be better for rapid ideation, while Skeleton Design mode is more effective for detailed and structured design.

Analysis of Octopus sketch

The Octopus sketch task required users to use multiple editing tools, such as adjusting the radius and moving points. The results show that users took an average of 338.1 seconds to complete the sketch, which is expected given the complexity of the task and the frequent use of editing tools (see Table 6.5).

Compared to previous sketching tasks, the number of sketched lines was significantly lower (25.5 lines on average), reflecting the reference image. Instead of continuously drawing new strokes, users spent more time refining their existing shapes, leading to an average time per stroke of 13.80 seconds, the highest observed across all sketching tasks. This suggests that users prioritized precision over speed, carefully modifying individual components rather than sketching new lines rapidly.

The frequent use of editing tools is shown in the results. Users moved points an average of 11.7 times. The radius was changed an average of 51.4 times, indicating that adjusting stroke thickness was a very important part of achieving the correct shape. Other key interactions included mesh generation (2.0 times per user) and clearing the mesh (2.9 times per user), suggesting that users evaluated their design and made corrections before finalizing it. The *see preview* feature was used 0.5 times, *see wireframe* was used 1.3 times and mirroring was used 1.1 times on average further supporting their answers to the questionnaire that they didn't find these tools very useful.

Metric	Average
Sketch Time (s)	338.1
Sketched Lines Drawn	25.5
Time per Stroke (s)	13.80
Undo Lines Drawn	5.9
Medium Lines Drawn	7.0
See Preview	0.5
Moved a Point	11.7
Changed Radius	51.4
Generated Mesh	2.0
See Wireframe	1.3
Cleared Mesh	2.9
Mirroring Used	1.1

Table 6.5: Metrics for the Octopus sketch

Overall, the analysis suggests that the octopus sketch task required a more methodical approach. The emphasis on modifying existing strokes rather than drawing new ones led to a significantly longer time per stroke and the use of editing features such as changing the radius and moving points.

6.4.3 System performance benchmarking

For the *system performance benchmarking*, we used the *Meta Quest Developer Hub* [53] along with the Meta Quest 3 headset. The goal was to analyze how the system behaves under different conditions. The performance metrics analyzed are:

- Frames Per Second (FPS): To measure the rendering speed and smoothness of the VR application.
- CPU load: To evaluate the computational demands of our system with the Meta Quest 3.
- GPU load: To assess the impact on the graphical processing unit during real-time rendering.

We collected performance measurements under total stroke count. The real time results below include up to 60 new strokes that were used from all the tools included in the application except the mesh generation plugin. Over 40 curves were intersecting at different points.

Figure 6.12: Frame Rate

The FPS results, shown in Figure 6.12, show that the application maintained a stable frame rate of approximately 73 FPS throughout the benchmarking process, with only minor changes. This confirms that the system has a stable real-time performance under typical workloads. There were some drops below 60 FPS when we had 50 drawn strokes, while enabling the preview mode and trying to edit the sketch by moving the point or changing the radius. During these actions, the preview spheres had to be rendered constantly for 50 stokes while the sketch was changing and that explains the drop in the frame rate.

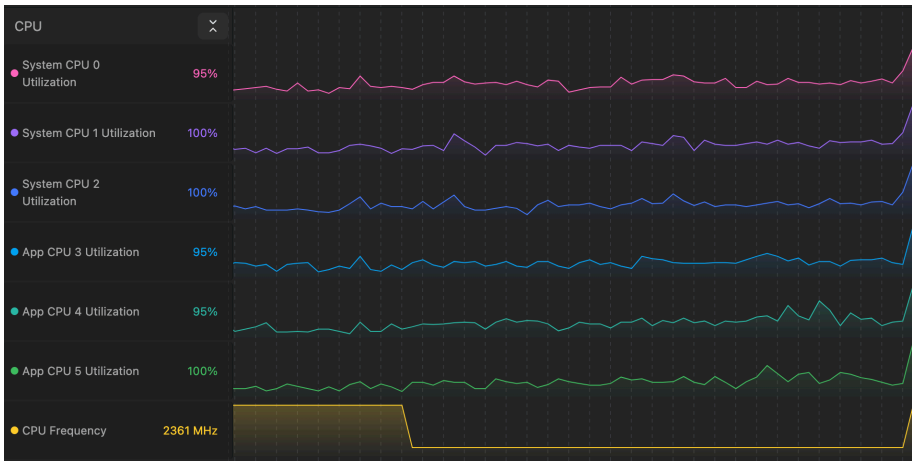


Figure 6.13: CPU Load

The CPU results, depicted in Figure 6.13, indicate that multiple CPU cores are utilized, with some reaching up to 100% utilization. This suggests that the application makes effective use of multithreading, but it also highlights the need for potential optimizations to balance CPU load distribution.

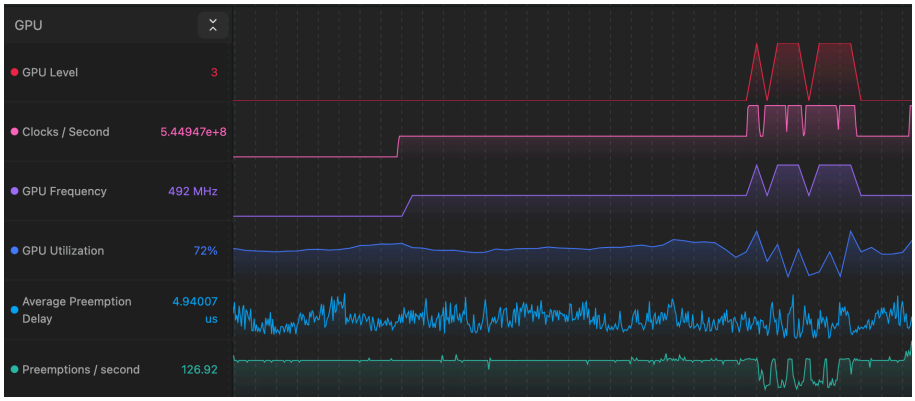


Figure 6.14: GPU Load

As seen in Figure 6.14, the GPU utilization averaged around 72%, with a GPU frequency of 492 MHz. This level of utilization indicates that the system efficiently distributes rendering workloads, ensuring real-time performance without overloading the GPU. As seen in FPS results, there is an increase at the GPU load again when using the preview tool enabled for over 50 strokes while trying to edit the sketch.

Performance considerations

The benchmarking results demonstrate that the system efficiently handles CPU and GPU workloads while maintaining stable frame rates. However, performance challenges arose when enabling the preview tool, as it rendered twice the number of points while the user attempted to edit the sketch. These issues only became noticeable when working with a large number of strokes, particularly in scenarios involving 50-60 strokes. Despite this, the system consistently maintained real time performance, as shown by the recorded data.

7 Discussion

The results of this study provide valuable insights into the user experience and effectiveness of the *VR skeleton stroke based drawing system for 3D model creation*. The main objectives (see Section 1.3) were to assess the system's intuitiveness and efficiency, measure task performance, evaluate mesh generation accuracy, gather user feedback, and analyze system performance. The results provide strong evidence that the system fulfills its goals effectively, while also showing both the advantages and areas for improvement in the system.

Intuitiveness and efficiency
One of the primary goals was to evaluate how intuitive and efficient it is for users to draw 3D curves in a VR environment. The results indicate that users, including those with no prior VR or 3D modeling experience, found the system easy to understand and enjoyable to use. The Skeleton Design mode, which included editing and assistive features, received higher ratings in ease of use (with an average score 4.1) compared to Free-Hand drawing (with an average score 3.4). Additionally, the average ease of learning score was 4.2, and all users reported enjoying the application. This shows that the interface and tools were accessible and learnable, even for novice users. Although users spent more time in Skeleton Design mode, the increased time was associated with editing, which suggests that users were taking advantage of the tools. These results confirm that the system is intuitive and user-friendly for novice users, achieving the first objective.

Task efficiency and user interaction
Regarding task efficiency, Skeleton Design mode allowed users to complete more accurate and refined sketches, though with a longer time per stroke. The Free-Hand mode supported rapid ideation, while Skeleton Design offered control and flexibility, enabling users to revise their sketches without starting over. Interestingly, users expressed that they felt they worked faster in Skeleton Design mode, but the metrics showed that they actually spent more time per stroke compared to Free-Hand mode. This inconsistency may be associated to how users perceived the different workflows. Skeleton Design may have felt more structured and efficient due to its editing tools, even though it required more time to use.

Sketching metrics showed a decrease in total strokes and undo actions in Skeleton Design mode, indicating that users were drawing fewer but higher quality strokes. The metrics also revealed a significant use of editing tools, supporting the idea that users were actively engaging with the features of the system to improve their sketches.

Mesh accuracy and quality
A key objective was to evaluate the effectiveness of the system in transforming skeleton sketches into accurate and structured 3D meshes. Visual analysis of user meshes confirms that the system successfully converted hand-drawn strokes into structured 3D models. Users enjoyed the ability to visualize and generate meshes directly in the headset, and no significant complaints were raised about mesh inaccuracies beyond some comments on stroke thickness. Although mesh smoothness was occasionally affected by the radius changes, the overall quality of the generated meshes was considered good, fulfilling the third objective.

User satisfaction and preferences
The fourth objective focused on assessing user satisfaction and preferences. All partic-

ipants reported enjoying the system, with an average enjoyment score of 4.9. The tools most appreciated were the point moving and radius adjustment features, which allowed for flexible sketch refinement. On the other hand, tools such as preview and wireframe were perceived as less useful and sometimes confusing.

Feedback from users highlighted the strengths of the system in ease of use and editing capabilities while also identifying areas that could be improved in future versions. Suggestions included better undo functionality, customizable mirroring, and improved mesh smoothness.

System performance
The final objective was to evaluate the performance of the system. Benchmarking on the Meta Quest 3 headset showed that the application maintained an average of 73 FPS in normal conditions. CPU usage reached 100% on certain cores, and GPU usage averaged 72%, indicating that the system efficiently utilized available resources.

However, performance drops were observed during mesh editing and when using the preview feature with a high number of strokes. These drops were caused by the system rendering preview spheres for each stroke, placing a heavier load on performance. Although these issues did not significantly disrupt usability, they point out opportunities for optimization. This confirms that the system works well in real time, but future improvements should focus on making it run more smoothly when there are many strokes.

7.1 Future work

While this thesis presents a new approach to immersive 3D modeling using hand-drawn strokes, there are several areas for future improvement and expansion. Based on user feedback and system evaluation, further development can enhance both the intuitiveness and functionality of the system.

One potential addition could be doodling and annotation features. This would allow users to make quick sketches or annotations within the VR environment using simple drawing tools, which could improve usability for conceptual modeling. Additionally, expanding the system's tools and input system, such as enabling radius adjustment with the left hand or allowing users to dynamically modify curve influence while moving a point could enhance precision and ease of use.

Another improvement involves optimizing and refining existing features based on user feedback. Users should be able to select and delete individual strokes instead of relying solely on the undo function. The mirroring tool could be improved by allowing users to define a custom mirroring plane rather than relying on predefined symmetry. Additionally, providing users with more control over stroke width while drawing would enhance flexibility in sketching. To further support the modeling process, the system could integrate reference images into the VR environment. This would allow users to import background images as guides while sketching. Another aspect of future work could be leveraging more functions from external libraries, such as GEL [19], to refine the mesh generation process, improving the output quality.

Beyond technical improvements, a broader user study could provide deeper insights into the system's usability. Evaluating the system across a wider range of user skill levels (from novice to expert designers) would offer valuable data on how well the tool adapts to different users' needs.

8 Conclusion

This thesis explored a VR based skeleton stroke modeling approach that allows users, particularly novices, to create 3D models through intuitive freehand sketching. The system was developed to provide a user friendly workflow where hand drawn strokes are converted into structured 3D meshes, integrating various editing tools to refine the curves to help user draw in 6DOF. Through user evaluations, sketching metrics analysis, and system performance benchmarking, the study assessed the effectiveness and usability of the proposed system.

The findings demonstrate that the Skeleton Design mode provided higher accuracy and refinement capabilities compared to Free-Hand Sketching. Users particularly valued having editing tools such as Move Points and Change Radius, which allowed for more precise modifications. However, some features, such as Preview and Wireframe, were underutilized, suggesting areas for improvement in future iterations. Performance tests showed that the system maintained real-time responsiveness.

Overall, this study highlights the potential of VR as an intuitive and immersive tool for 3D modeling. The system provides a balance between immersive mid-air sketching and structured editing. While there are areas for refinement, this work serves as a step toward more accessible and interactive VR design tools.

Bibliography

[1] Peng Li. “Research on Visual Art Design Method Based on Virtual Reality”. In: *International Journal of Gaming and Computer-Mediated Simulations* 13 (Jan. 2021), pp. 16–25. DOI: 10.4018/IJGCMS.2021040102.

[2] Blender Foundation. *Blender*. https://www.blender.org. Accessed: 2025-03-20. 2023.

[3] Autodesk Inc. *What is CAD? Learn about Computer-Aided Design*. Accessed: 2024-03-20. 2023. URL: https://www.autodesk.com/solutions/cad-software.

[4] Chen Chen et al. “Investigating Input Modality and Task Geometry on Precision-first 3D Drawing in Virtual Reality”. In: *2022 IEEE International Symposium on Mixed and Augmented Reality (ISMAR)*. IEEE, Oct. 2022, pp. 384–393. DOI: 10.1109/ismar55827.2022.00054. URL: http://dx.doi.org/10.1109/ISMAR55827.2022.00054.

[5] *Gravity Sketch*. 2019. URL: https://www.gravitysketch.com/.

[6] Alfred Oti and Nathan Crilly. “Immersive 3D sketching tools: Implications for visual thinking and communication”. In: *Computers Graphics* 94 (2021), pp. 111–123. ISSN: 0097-8493. DOI: https://doi.org/10.1016/j.cag.2020.10.007. URL: https://www.sciencedirect.com/science/article/pii/S0097849320301667.

[7] M. D. Barrera Machuca et al. “Multiplanes: Assisted Freehand VR Drawing”. In: *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology (UIST ’17)*. ACM, 2017, pp. 1–3. DOI: 10.1145/3131785.3131794.

[8] J. M. Cohen et al. “An Interface for Sketching 3D Curves”. In: *Proceedings of the 1999 Symposium on Interactive 3D Graphics*. 1999, pp. 17–21.

[9] Y. Gingold, T. Igarashi, and D. Zorin. “Structured Annotations for 2D-to-3D Modeling”. In: *ACM SIGGRAPH Asia 2009 Papers*. 2009, pp. 1–9.

[10] Joon Hyub Lee et al. “RobotSketch: A Real-Time Live Showcase of Superfast Design of Legged Robots”. In: *SIGGRAPH Asia 2024 Real-Time Live!* SA Real-Time Live! ’24. Tokyo, Japan: Association for Computing Machinery, 2024. ISBN: 9798400711398. DOI: 10.1145/3681757.3697055. URL: https://doi.org/10.1145/3681757.3697055.

[11] Rahul Arora et al. “SymbiosisSketch: Combining 2D & 3D Sketching for Designing Detailed 3D Objects in Situ”. In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. CHI ’18. Montreal QC, Canada: Association for Computing Machinery, 2018, pp. 1–15. ISBN: 9781450356206. DOI: 10.1145/3173574.3173759. URL: https://doi.org/10.1145/3173574.3173759.

[12] Tianrun Chen et al. “Deep3DSketch+: Rapid 3D Modeling from Single Free-Hand Sketches”. In: Mar. 2023, pp. 16–28. ISBN: 978-3-031-27817-4. DOI: 10.1007/978-3-031-27818-1_2.

[13] S.-H. Zhang, Y.-C. Guo, and Q.-W. Gu. “Sketch2model: View-aware 3d modeling from single free-hand sketches”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2021, pp. 6012–6021.

[14] Enrique Rosales, Jafet Rodriguez, and ALLA SHEFFER. “SurfaceBrush: from virtual reality drawings to manifold surfaces”. In: *ACM Transactions on Graphics* 38.4 (July 2019), pp. 1–15. ISSN: 1557-7368. DOI: 10.1145/3306346.3322970. URL: http://dx.doi.org/10.1145/3306346.3322970.

[15] Rahul Arora et al. “Experimental Evaluation of Sketching on Surfaces in VR”. In: *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI ’17)*. New York, NY, USA: ACM, 2017, pp. 5643–5654. DOI: 10.1145/3025453.3025474.

[16] Mayra Donaji Barrera Machuca, Wolfgang Stuerzlinger, and Paul Asente. “The Effect of Spatial Ability on Immersive 3D Drawing”. In: *Proceedings of the ACM Conference on Creativity and Cognition (C&C ’19)*. New York, NY, USA: ACM, 2019, pp. 173–186. DOI: 10.1145/3325480.3325485.

[17] Yu-Hsin Tung and Chun-Yen Chang. “How three-dimensional sketching environments affect spatial thinking: A functional magnetic resonance imaging study of virtual reality”. In: *PLOS ONE* 19.3 (Mar. 2024), pp. 1–19. DOI: 10.1371/journal.pone.0294451. URL: https://doi.org/10.1371/journal.pone.0294451.

[18] D. Seth, S. Purohit, and V. Bhakar. “Applications of CAD/CAM in Modern Engineering”. In: *International Journal of Computer Science* 10.2 (2020), pp. 55–67. DOI: 10.1007/s10207-020-00562.

[19] Andreas Bærentzen et al. *Guide to computational geometry processing. Foundations, algorithms, and methods*. Jan. 2012. ISBN: 978-1-4471-4074-0. DOI: 10.1007/978-1-4471-4075-7.

[20] *Tilt Brush*. 2021. URL: https://www.meta.com/en-gb/experiences/tilt-brush/2322529091093901/.

[21] *OpenBrush*. 2021. URL: https://openbrush.app/.

[22] Facebook. *Oculus Quill*. Accessed: 2025-01-28. 2016. URL: https://www.oculus.com/experiences/rift/1118609381580656/.

[23] Richard Rodriguez et al. “An Artists’ Perspectives on Natural Interactions for Virtual Reality 3D Sketching”. In: *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*. CHI ’24. Honolulu, HI, USA: Association for Computing Machinery, 2024. ISBN: 9798400703300. DOI: 10.1145/3613904.3642758. URL: https://doi.org/10.1145/3613904.3642758.

[24] Mayra D. Barrera Machuca, Wolfgang Stuerzlinger, and Paul Asente. “Smart3DGuides: Making Unconstrained Immersive 3D Drawing More Accurate”. In: *Proceedings of the 25th ACM Symposium on Virtual Reality Software and Technology*. VRST ’19. Parramatta, NSW, Australia: Association for Computing Machinery, 2019. ISBN: 9781450370011. DOI: 10.1145/3359996.3364254. URL: https://doi.org/10.1145/3359996.3364254.

[25] Rahul Arora and Karan Singh. “Mid-Air Drawing of Curves on 3D Surfaces in Virtual Reality”. In: *ACM Trans. Graph.* 40.3 (July 2021). ISSN: 0730-0301. DOI: 10.1145/3459090. URL: https://doi.org/10.1145/3459090.

[26] Emilie Yu et al. “CASSIE: Curve and Surface Sketching in Immersive Environments”. In: *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. CHI ’21. Yokohama, Japan: Association for Computing Machinery, 2021. ISBN: 9781450380966. DOI: 10.1145/3411764.3445158. URL: https://doi.org/10.1145/3411764.3445158.

[27] Ling Luo et al. “3D VR Sketch Guided 3D Shape Prototyping and Exploration”. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2023, pp. 9267–9276.

[28] Tianrun Chen et al. “Reality3DSketch: Rapid 3D Modeling of Objects From Single Freehand Sketches”. In: *Trans. Multi.* 26 (Jan. 2024), pp. 4859–4870. ISSN: 1520-9210. DOI: 10.1109/TMM.2023.3327533. URL: https://doi.org/10.1109/TMM.2023.3327533.

[29] Fatemeh Abbasinejad, Pushkar Joshi, and Nina Amenta. “Surface patches from unorganized space curves”. In: *Computer Graphics Forum* 30 (2011).

[30] Mikhail Bessmeltsev et al. “Design-driven quadrangulation of closed 3D curves”. In: *ACM Transactions on Graphics (Proc. SIGGRAPH Asia)* 31.6 (2012).

[31] Bardia Sadri and Karan Singh. “Flow-complex-based shape reconstruction from 3D curves”. In: *ACM Transactions on Graphics (TOG)* 33.2 (2014).
[32] Matthew Berger et al. “A survey of surface reconstruction from point clouds”. In: *Computer Graphics Forum* 36 (2017).
[33] Emilie Yu et al. “Piecewise-smooth surface fitting onto unstructured 3D sketches”. In: *ACM Trans. Graph.* 41.4 (July 2022). ISSN: 0730-0301. DOI: 10.1145/3528223.3530100. URL: https://doi.org/10.1145/3528223.3530100.
[34] Jules Bloomenthal and Brian Wyvill. “Interactive Techniques for Implicit Modeling”. In: *Computer Graphics (ACM)* 24 (Nov. 2000). DOI: 10.1145/91394.91427.
[35] C. Zanni et al. “Scale-Invariant Integral Surfaces”. In: *Computer Graphics Forum* 32.8 (2013), pp. 219–232.
[36] J. Bærentzen, Marek Misztal, and K. Wełnicka. “Converting skeletal structures to quad dominant meshes”. In: *Computers Graphics* 36 (Aug. 2012), pp. 555–561. DOI: 10.1016/j.cag.2012.03.016.
[37] Karran Pandey, J. Andreas Bærentzen, and Karan Singh. “Face Extrusion Quad Meshes”. In: *ACM SIGGRAPH 2022 Conference Proceedings*. SIGGRAPH ’22. Vancouver, BC, Canada: Association for Computing Machinery, 2022. ISBN: 9781450393379. DOI: 10.1145/3528233.3530754. URL: https://doi.org/10.1145/3528233.3530754.
[38] J. Andreas Bærentzen. *GEL library*. 2018. URL: https://github.com/janba/GEL.
[39] Jason Jerald. *The VR Book: Human-Centered Design for Virtual Reality*. Association for Computing Machinery and Morgan & Claypool, 2015. ISBN: 9781970001129.
[40] Mayra Donaji Barrera Machuca et al. “Toward More Comprehensive Evaluations of 3D Immersive Sketching, Drawing, and Painting”. In: *IEEE Transactions on Visualization and Computer Graphics* 30.8 (2024), pp. 4648–4664. DOI: 10.1109/TVCG.2023.3276291.
[41] Qian Zou et al. “Stylus and Gesture Asymmetric Interaction for Fast and Precise Sketching in Virtual Reality”. In: *International Journal of Human–Computer Interaction* 40.23 (2023), pp. 8124–8141. DOI: 10.1080/10447318.2023.2278294. URL: https://doi.org/10.1080/10447318.2023.2278294.
[42] Taneli Nyyssönen et al. “A Comparison of One- and Two-Handed Gesture User Interfaces in Virtual Reality—A Task-Based Approach”. In: *Multimodal Technologies and Interaction* 8.2 (2024), p. 10. DOI: 10.3390/mti8020010. URL: https://www.mdpi.com/2414-4088/8/2/10.
[43] Nimrod Kramer. *VR Interaction Design: Best Practices & Principles*. Accessed: February 3, 2025. 2024. URL: https://daily.dev/blog/vr-interaction-design-best-practices-and-principles.
[44] Leap Motion. *VR Design Best Practices*. Accessed: February 3, 2025. 2016. URL: https://medium.com/@LeapMotion/vr-design-best-practices-bb889c2dc70.
[45] Timoni West. *UX Pointers for VR Design*. Accessed: February 3, 2025. 2016. URL: https://medium.com/@timoni/ux-pointers-for-vr-design-dd52b718e19.
[46] Mathieu Desbrun et al. “Implicit Fairing of Irregular Meshes using Diffusion and Curvature Flow”. In: *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 1999)*. ACM Press/Addison-Wesley Publishing Co., 1999, pp. 317–324. DOI: 10.1145/311535.311576.
[47] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. “An Algorithm for Finding Best Matches in Logarithmic Expected Time”. In: *ACM Trans. Math. Softw.* 3.3 (Sept. 1977), pp. 209–226. ISSN: 0098-3500. DOI: 10.1145/355744.355745. URL: https://doi.org/10.1145/355744.355745.
[48] Santosh Kumar Uppada. “Centroid Based Clustering Algorithms-A Clarion Study”. In: 2014. URL: https://api.semanticscholar.org/CorpusID:1649872.

[49] Daniel Saakes. “Sketching in Virtual Reality for Rapid and Situated Idea Generation”. In: *International Design Congress*. 2015, pp. 1–4.
[50] Unity Technologies. *Unity Real-Time Development Platform*. https://unity.com. Accessed: 2025-03-20. 2023.
[51] Google. *Android NDK*. 2023. URL: https://developer.android.com/studio/projects/install-ndk.
[52] Unity. *VR multiplayer template*. 2024. URL: https://docs.unity3d.com/Packages/com.unity.template.vr-multiplayer@2.0/manual/index.html.
[53] Meta. *Meta Quest Developer Hub for Mac*. Accessed: March 5, 2025. 2025. URL: https://developers.meta.com/horizon/downloads/package/oculus-developer-hub-mac/.

A Controllers and Menu

This appendix presents the controller mappings for both the right and left controllers of the Meta Quest 3 headset as well as the Menus created.

The labels shown in the images below correspond to what users viewed within the software. Additionally, users had the option to toggle these labels on or off using the designated button, as indicated. The right hand is considered the dominant hand in this mapping.

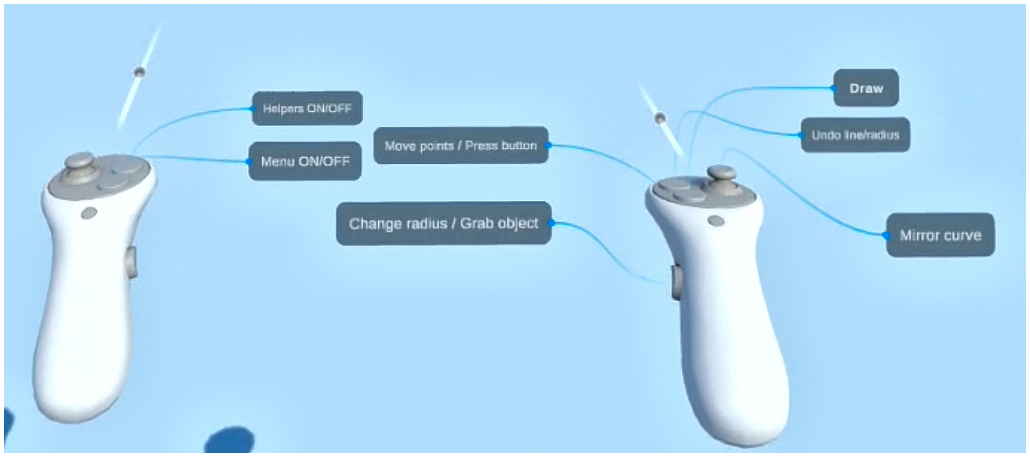
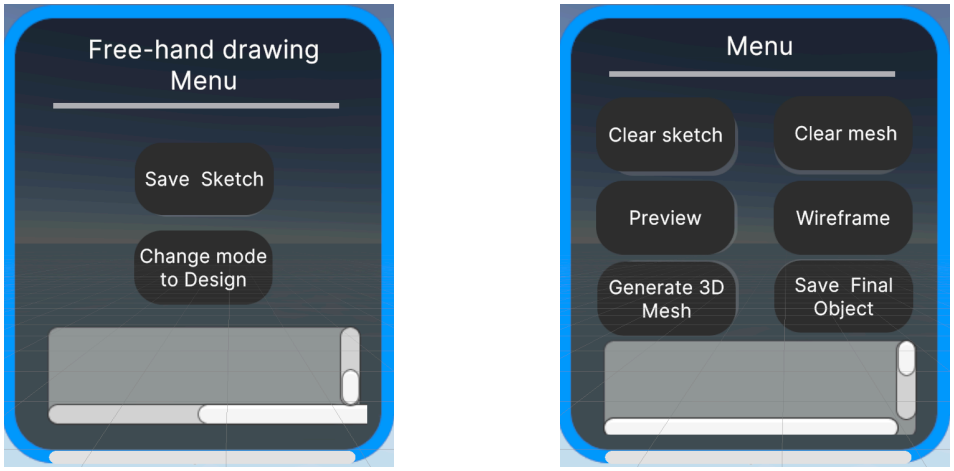


Figure A.1: Controllers

Below are the two menus created for the system's two different modes.



(a) Free-Hand drawing menu

(b) Skeleton design menu

Figure A.2: Menu

B Questionnaire

This appendix contains the questionnaire used to evaluate the VR skeleton stroke based drawing system for 3D model creation and it was created and distributed with Google Forms.

VR Skeleton Stroke Based Drawing System Questionnaire

B.1 General questions

1. Write the first letters of your name (for cross-checking reference).
2. Have you ever used any design or modeling software before? (Yes/No)
3. Have you ever used any Virtual Reality application before? (Yes/No)
4. Do you have any experience in 3D design or modeling? (Yes/No)

B.2 Comparison with free-Hand sketch

1. Which application did you find easier to use?
    - Free-Hand Sketch
    - Skeleton Design App/mode
2. How easy was it to create a sketch in Free-Hand drawing mode?
    - Very difficult 1 – 2 – 3 – 4 – 5 Very easy
3. How easy was it to create a sketch in the Skeleton Design App/mode?
    - Very difficult 1 – 2 – 3 – 4 – 5 Very easy
4. How accurate do you feel your sketch was in Free-Hand sketching mode?
    - Not accurate 1 – 2 – 3 – 4 – 5 Very accurate
5. How accurate do you feel your sketch was in the Skeleton Design App/mode?
    - Not accurate 1 – 2 – 3 – 4 – 5 Very accurate
6. How efficient (fast) was the sketching process in Free-Hand sketching mode?
    - Very slow 1 – 2 – 3 – 4 – 5 Very fast
7. How efficient (fast) was the sketching process in Skeleton Design App/mode?
    - Very slow 1 – 2 – 3 – 4 – 5 Very fast

B.3 Drawing experience

1. How intuitive were the sketching tools in the new design system?
    - Not intuitive at all 1 – 2 – 3 – 4 – 5 Very intuitive
2. Which tool/feature did you find the most useful?
    - Drawing multiple strokes to influence the curve
    - Move points

- Change radius
- Mirroring
- Generating the mesh
- Preview
- Wireframe

3. Which feature was the least helpful or confusing?
4. How easy was it to edit/refine your sketch using the tools (move, mirror, adjust point radius, curve adjustment)?
    - Very difficult 1 – 2 – 3 – 4 – 5 Very easy
5. How easy was it to learn the new VR system?
    - Very difficult 1 – 2 – 3 – 4 – 5 Very easy

B.4 User satisfaction

1. How quickly did you feel comfortable using the system?
    - Immediately
    - After a few minutes
    - After multiple attempts
    - Still not comfortable
2. Did you enjoy using it? (Yes/No)
3. Do you think the system is beginner-friendly? (Yes/No)
4. Did you notice any lag or frame rate drops while drawing?
    - No noticeable lag
    - Occasional lag
    - Frequent lag
    - Severe lag

B.5 Open feedback

1. What did you like most about the application?
2. What did you like least about the application?
3. Is there another feature you would like the application to have? (Yes/No)
    - If yes, which one?
4. Anything else you would like to add?

C Software

Unity version: 2022.3.44f1
Visual Studio version: 2022

Packages used in Unity:

XR Plugin Management 4.5.0
XR Interaction Toolkit 3.0.3
Oculus XR Plugin 4.2.0
OpenXR Plugin 1.12.1
TextMeshPro version: 3.0.6

D Additional implementation details

This appendix has additional implementation details for some features of the system.

D.1 Additional features details

D.1.1 Undo functionality

When the user triggers the undo action, the system determines whether a stroke width adjustment was the last change. If a width modification was detected, the system resets the width of the last modified stroke instead of removing a line. This prioritization ensures that minor adjustments can be reverted before an entire stroke is undone.

If no width modification is found, the system removes the most recently drawn stroke from the sketch container. This process involves:

- Identifying the last drawn `LineRenderer` in the active sketch container.
- Removing the corresponding points from internal data structures, including the KD-tree, point clusters, and intersection tracking system.
- Destroying the line object from the scene.
- Updating the spatial data structures to reflect the removal.

If the removed stroke contained intersection points shared with other strokes, the system verifies whether these intersections are still valid. If an intersection point is no longer referenced by any remaining strokes, it is also removed from the visualization and intersection tracking.

Once a stroke is removed, the system updates:

- The *KD-tree*, ensuring efficient nearest-neighbor searches for future strokes.
- The *point clusters*, dynamically adjusting their structure after a stroke removal.
- The global *allPoints* list, which maintains a record of all remaining points in the scene.

If the undo operation results in an empty scene, the system resets the KD-tree and reinitializes the clustering system.

D.1.2 Clear drawing

The implementation consists of two main steps:

1. Deactivating or removing existing sketch elements/objects (sketched lines, preview strokes, points, and junctions)
2. Creating a new sketch container for new Strokes and resetting data structures (list of stored points, KD-tree and clustering structures)

Deactivating or removing existing sketch elements

To clear the sketch, all game objects associated with the previous drawing session are identified and removed. This includes:

- *Sketched strokes*, which are stored as objects tagged with `"LineSegment"`.

- *Preview spheres*, which allow users to visualize their drawing before confirming.
- *Visualized points and junctions*, which serve as interaction elements.

The function `ClearAllData()` is responsible for iterating through these objects and either disabling or deleting them. All objects tagged as `"LineSegment"` are immediately destroyed, while preview objects tagged as `"PreviewContainer"` are deactivated to avoid unnecessary memory reallocation.

Creating a new sketch container for new strokes and resetting data structures

Once previous elements have been cleared and internal states reset, a new container is created for storing upcoming strokes. To prevent any interference from previous sketches, internal counters and data structures are reset. This ensures that no residual data affects new drawings.

A new container is instantiated dynamically and assigned a unique name (`"LinesContainer_x"`), where `X` represents an incrementing identifier. Additionally, the system:

- Initializes an empty list of points for tracking strokes.
- Resets the KD-tree to ensure accurate spatial queries.
- Initializes clustering structures to optimize point grouping.

This functionality is implemented in the `CreateNewLineContainer()` function. It ensures that each new sketch starts with a dedicated storage structure, preventing conflicts with previous drawings.

D.2 Android NDK integration

The Android NDK configuration is defined from the file *Application.mk* as follows:

- Enabled using `-fexceptions` for error handling in the C++ plugin.
- The plugin is compiled with `c++_shared` for compatibility with Unity.
- Targeting the ARM 64-bit architecture (`arm64-v8a`)for Android VR devices such as Meta Quest 3.
- The build targets Android API Level 21 (`android-21`), ensuring compatibility with modern VR platforms.

The C++ plugin requires linking against several dependencies, including:

- GEL Library (`libGEL.a`): Provides essential geometry processing functionalities.
- PyGEL Library (`libPyGEL.so`): Extends the GEL library for additional mesh manipulation.
- Android System Libraries: The plugin links against `llog`, `landroid`, and `lGLESv3` to support logging, system operations, and OpenGL rendering.

To facilitate integration with Unity, the generated shared library exports a function `ProcessPointCloud`, which can be called using Unity's DLL import mechanism. The function takes input and output file paths as parameters, allowing Unity to provide the point cloud and receive the processed 3D mesh.

The build script uses the *Android.mk* file to configure the compilation and linking settings. It includes:

- **Prebuilt libraries:** The GEL and PyGEL libraries are linked as external dependencies.
- **Source file inclusion:** The main processing script is specified in the build configuration.
- **Compiler flags:** The build process enables C++ exception handling.

D.3 Building GEL and PyGEL Plugin with CMake

To compile the GEL and PyGEL libraries into a shared plugin for Android, the build process was managed using *CMake* along with the *Android NDK toolchain*. This approach allows integration of the native C++ library with Unity while ensuring compatibility with ARM-based Android devices.

CMake build process

The compilation process consists of the following steps:

1. **Configure the Android toolchain:** CMake is executed with the Android NDK toolchain file, which ensures the build system is correctly set up for cross-compilation targeting ARM architectures.
2. **Define the target ABI and platform:** The compilation specifies `arm64-v8a` as the target architecture and `android-21` as the minimum API level for compatibility.
3. **Build and Install:** CMake is instructed to build the necessary targets and install the compiled shared libraries.

The following command is used to configure and build the project:

```
cmake .. -DCMAKE_TOOLCHAIN_FILE=/Applications/AndroidNDK.app/Contents/
        NDK/build/cmake/android.toolchain.cmake
         -DANDROID_ABI="arm64-v8a"
         -DANDROID_PLATFORM=android-21
cmake --build . --target install
```

CMake Option	Description
`-DCMAKE_TOOLCHAIN_FILE`	Specifies the location of the Android NDK toolchain file, which configures the cross-compilation environment.
`-DANDROID_ABI="arm64-v8a"`	Targets the `arm64-v8a` architecture for compatibility with modern Android devices such as Meta Quest 3.
`-DANDROID_PLATFORM=android-21`	Sets the minimum API level to `android-21` to maintain compatibility with Android systems.
`cmake --build . --target install`	Executes the build process and installs the compiled libraries in the designated output directory.

Table D.1: Build options

After the build completes, the generated `libPyGEL.so` and `libGEL.a` files are placed in the appropriate Unity `Plugins/Android` folder, allowing Unity to load and interact with the compiled GEL and PyGEL functionalities.

By compiling GEL and PyGEL into a shared library, Unity gains access to geometric processing capabilities, enabling real-time skeletonization of point clouds within the VR environment.

Technical
University of
Denmark

Richard Petersens Plads, Building 324
2800 Kgs. Lyngby
Tlf. 4525 1700

www.compute.dtu.dk