Insert here your thesis' task.

Czech Technical University in Prague
Faculty of Information Technology
Department of Software Engineering

Bachelor's thesis

Sculpting in Virtual Reality

Vojtěch Krs

Supervisor: Ing. Ondřej Jamriška

14th May 2014

Acknowledgements

I would like to thank my supervisor, Ing. Ondřej Jamriška, without whom this thesis would not be possible and whose guidance and enthusiasm were invaluable. I'd like to thank my family for the support they gave me throughout the process of writing and during my studies. My thanks also goes to my friends and schoolmates, many of whom share my amazement of new technology and particularly, virtual reality.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on 14th May 2014 . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Krs, Vojtěch. *Sculpting in Virtual Reality.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2014.

Abstrakt

Sculptování je populární metodou 3D modelování, která by mohla využít nových pokroků v oblasti virtuální reality. Tato práce se zaměřuje právě na implementaci sculptovacího systému ve virtuálním prostředí. Pro zobrazení virtuálního prostředí byly použity brýle pro virtuální realitu, Oculus Rift, přičemž interakce uživatele s tímto prostedím je umožněna pomocí Razer Hydra, ovladačů sledující pohyb a orientaci rukou. Samotné modelování využívá trojúhelníkové sítě zjemněné dle potřeby. Systém byl úspešně implementován a otestován. Dokazuje, že 3D modelování, zejména sculpting, ve virtuální realitě má smysl a navíc nabízí uživateli reálný dojem hloubky a velikosti spolu s intuitivním ovládáním.

Klíčová slova sculpting, virtuální realita, oculus rift, razer hydra, adaptivní trojúhelníková sít’

Abstract

Sculpting is a popular method of 3D modeling that could benefit from the recent advancements in the field of virtual reality. This thesis focuses on implementation of a sculpting system in a virtual reality environment. Oculus Rift head mounted display is used to immerse the user in this environment, while the interaction is facilitated by Razer Hydra, a 6-DOF hand tracking device. The sculpting employs an adaptively refined triangle mesh to represent the sculpted surface. The system was successfully implemented and tested, proving that sculpting in virtual reality is a viable option, offering the user a real sense of scale and depth in addition to an intuitive interaction.

Keywords sculpting, virtual reality, oculus rift, razer hydra, adaptive triangle mesh

Contents

List of Figures

List of Tables

Introduction

Motivation

The concept of virtual reality has been around for decades. What started as a science fiction has become more and more tangible thanks to the recent advances in technology. Head mounted displays (HMDs) are currently the best way to transport yourself to a different reality, along with special accessories such as haptic gloves, which communicate your hand gestures to a computer and can provide tactile feedback, or omnidirectional treadmills, which allow you to walk around virtual space without leaving your living room.

Until recently, HMDs were somewhat crude and very expensive, limiting their use only for special purposes such as military training. However, in 2012, Oculus Rift has been introduced. It is a HMD featuring low latency, wide field of view (FOV) and a relatively low price. Low latency and wide FOV are essential to a sense of immersion. If this sense is maximized, it can trick your brain to believe that you are actually in another reality.

The low price of Oculus Rift is a key for virtual reality to become mainstream. This headset is focused primarily on gaming, which has already proven to be a great driving force for technological advancement of graphic processing units (GPUs). However, the applications of virtual reality are hardly limited to the entertainment industry. Other uses include virtual reality therapy for treating phobias or posttraumatic stress disorder, medical training, architecture visualization and many others.

The application of virtual reality that I decided to focus on is 3D modeling, which is a process of creating three dimensional model of a physical object using a computer. The reason why I chose modeling, is that it can benefit tremendously from virtual reality. 3D artists today use 2D monitors with mouse or tablets for doing something that is inherently physical and three dimensional, therefore virtual reality seems like an ideal way to improve this process.

There are many approaches to modeling, but the one that is the closest to the physical act of modeling is sculpting. It is based on the same principle as real sculpting, thus its implementation in virtual reality could prove to be advantageous.

Head mounted display can provide only one part of the virtual reality, the image of an artificial world. However, what makes the experience even more compelling, is the interaction with this world. Traditional input methods, such as keyboard and mouse, aren't very suitable for virtual reality. I decided to use Razer Hydra instead, which are controllers that are able to track position and orientation of user's hands. This is essential for sculpting, because it allows the user to interact with the model in a fashion a sculpter would with a real sculpture.

Goals

The primary goal of my thesis is to design and implement a simple virtual reality sculpting system that will support the Oculus Rift HMD and will use Razer Hydra as the input controller. The primary features of the system will be as follows:

- An adaptively refined triangle mesh will be employed to represent the sculpted clay surface
- Four basic tools will be implemented, which will allow the user to:
  - build up the surface by adding clay
  - create dimples or holes by removing the clay
  - drag the surface
  - smooth the surface
- The system will be implemented in C++ using the OpenGL API

The secondary goals are to design the system is such a way that it is easy to use for novice users and performant enough to handle meshes with up to 100 thousand triangles. Besides the sculpting tools outlined above, the goal is to also implement a set of interaction tools that will allow the user to manipulate the model during sculpting in order to e.g., rotate it or move it around.

Chapter 1

Sculpting with Triangular Meshes

Imagine a sculptor creating a physical sculpture. What does he need before he starts to form even the most basic shape? Stone, hammer and chisel? Clay and hands? In general, a sculptor needs material and tools. Those are the two most essential things for sculpting. And this applies to both physical and 3D computer sculpting.

To be able to model an object in a computer, we need a way to represent it so the computer can understand it. The representation affects not only the way we save it in the computer memory, but consequently how we interact with it. The representation of the model can be compared to the sculptor's material and the program tools to physical tools.

1.1 Representation

The process of 3D modeling largely depends on the underlying representation of the object. Different representations may differ in their capabilities to represent various shapes, in the way we interact with them and also in their performance. The representations can be divided into two main categories, volume and surface, while both are viable for sculpting.

The volume representation describes the entire volume of an object by a collection of elements called voxels, which hold information about a particular region in space, usually color or material. This representation is more realistic, but can be hard for the computer to work with as the amount of data needed to represent even a simple object can be enormous. Nevertheless, it's commonly used in medical or engineering simulations.

On the other hand, the surface representation only describes the boundaries of an object. This can save a lot of memory space as well as computation power and is suitable for applications where information only about the surface

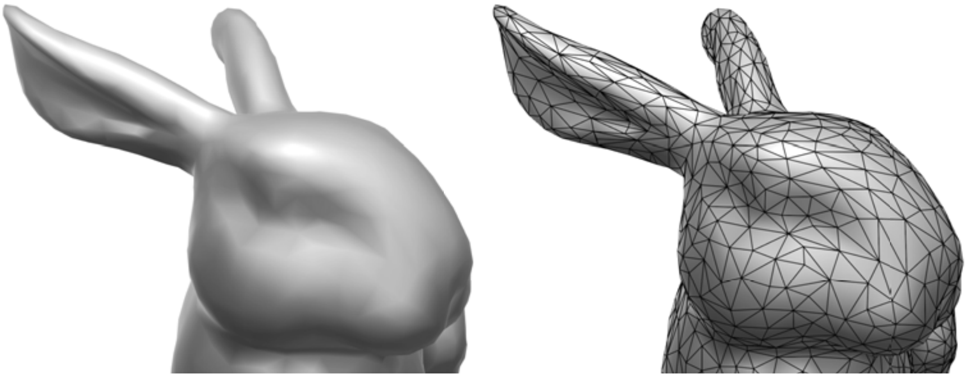


Figure 1.1: The surface of a bunny's head (left) as represented by a triangle mesh (right).

of an object is enough.

The most common and simplest way to describe a surface is to stitch it together from a geometrical primitive, while the most basic 3D planar primitive is the triangle. By connecting a number of triangles, we can approximate the shape and surface of an arbitrary object. The result is a graph, called *mesh*, containing the triangle's points, which are called *vertices*. These vertices are then connected by triangle's sides, called *edges*, to other vertices to finally form the individual triangles. Figure 1.1 illustrates such triangle mesh.

We can assemble the mesh also by using other polygons, such as quadrilaterals. However, triangles have the advantage of being natively supported by rendering hardware. This is because they are fairly easy to work with, particularly the math that is used is simple. This allows for fast processing and rendering.

1.2 Modeling and Sculpting

When dealing with meshes, the actual modeling process consists of manipulation of vertices, edges and faces. To shape a surface, transformations such as translation, rotation or scale are applied to these entities.

Commonly in 3D modeling, these transformations, along with other operations such as creation, removal or interconnection, are applied to individual vertices or a set of vertices defined by selection. This gives us full control over the mesh but it can be tedious to work with, especially when modeling a large and complicated mesh.

In the case of sculpting, we give up the full control over the mesh in favour of fast and easy modeling. Instead of moving individual entities, tools like pulling, pushing and smoothing of the are applied on the mesh as if it would be made of clay. The sculpting process usually starts with a simple model,

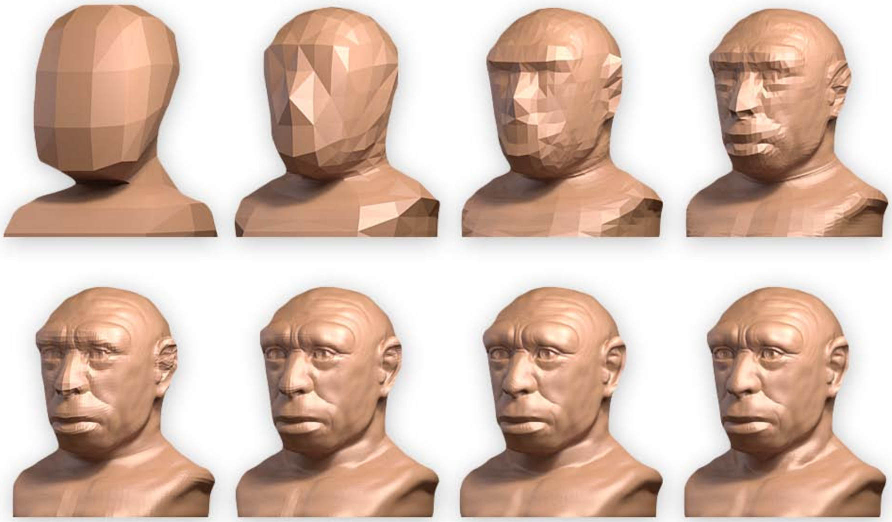


Figure 1.2: Sculpting process iterations: the artist starts with a basic mesh and gradually adds details (courtesy of Cícero Moraes [1]).

such as sphere, on which the tools are applied one after another to add detail and sculpt it into the desired shape. Figure 1.2 shows snapshots of this process.

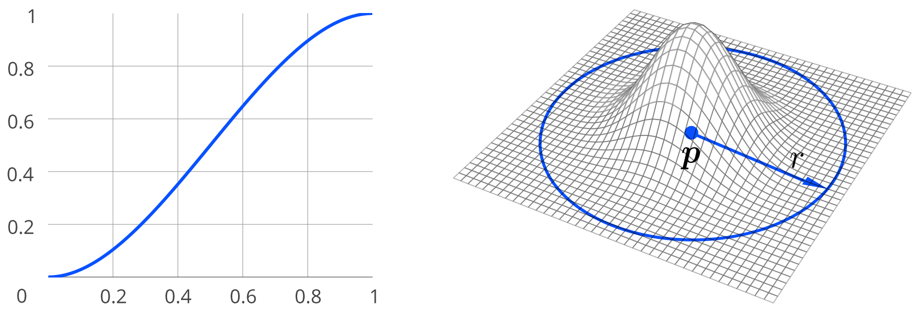
1.3 Sculpting Tools

Sculpting tools are usually applied to a group of vertices, in what's called an area of interest. This area is usually spherical, while its size is determined by a radius parameter. However, the position of the area has to be computed depending on user input. This is done by sending a ray, typically from a mouse cursor, and finding *point of intersection* (POI) of the ray with the surface of the mesh. The tool is then applied in the direct vicinity of this point.

Another attribute of a tool is strength. It dictates how much should the tool be applied. It can be either uniform over the whole area of interest or it can vary. The variations in strength are then computed by a *weight function.*

The weight function computes the *weight coefficient* of a point, i.e., how much of the strength should be applied at a given vertex, depending on its distance from the POI. To achieve a brush-like behaviour, when the center of the area is transformed the most, while the strength diminishes with distance from the center, a *smoothstep* [2] function (equation 1.1, figure 1.3a) is used.

$$\mathit{smoothstep}(t) = 6t^5 \quad 15t^4 + 10t^3 \tag{1.1}$$



(a) Smoothstep function

(b) Weight function w

Figure 1.3: Functions used for vertex weight calculation

1.3.1 Common principles

When a tool is applied, a new position is calculated for every vertex inside the area of interest. Every vertex needs two additional pieces of information before we can calculate its transformed position: a local vertex normal and a weight coefficient.

The local vertex normal $\boldsymbol{n}$ is a vector that is perpendicular to the surface near given vertex. We compute it by averaging surface normals of all triangles that contain given vertex [3]. The surface normal of a triangle is given by the cross product of its two edges. Because the mesh shape is changing with every application of a tool, these normals have to be recalculated after every application.

To calculate the weight coefficient we use the aforementioned weight function w. Figure 1.3b shows the shape of the function, which is also the shape of the surface after applying pull tool, for instance. Equation 1.2 describes the weight function, where $\boldsymbol{v}$ is the vertex in question, $\boldsymbol{p}$ the point of intersection and r the radius.

$$w(\boldsymbol{v}) = 1 \quad \mathit{smoothstep}\left(\frac{\|\boldsymbol{v} \quad \boldsymbol{p}\|}{r}\right) \tag{1.2}$$

We can either apply tools continuously, every frame, or gradually, when the tool moves. If the tool is applied every frame, at a rate of several frames per second, we need the tool application to be independent of this rate. To achieve this, we have to incorporate time into the calculation of transformed vertex position by introducing another variable, the elapsed time Δt.

Therefore, when determining the new vertex position, the final rate $\rho(\boldsymbol{v})$ at which a transformation of the tool is applied can be calculated as a product of above variables, as seen in equation 1.3, where s denotes the global strength of the tool.

$$\rho(\boldsymbol{v}) = w(\boldsymbol{v})s\Delta t \tag{1.3}$$

1.3.2 Pull Tool

Pull tool is the simplest tool, as during its application the vertices are simply moved in the direction of their local normal vector. This causes vertices of a flat surface to move in the same direction and vertices of a curved surface to move away from each other. Volume described by such curved surface appears to inflate when this tool is applied as seen in figure 1.4. The way get the transformed position of a vertex is illustrated by equation 1.4.

$$\boldsymbol{v}' = \boldsymbol{v} + \rho(\boldsymbol{v})\boldsymbol{n}; \tag{1.4}$$

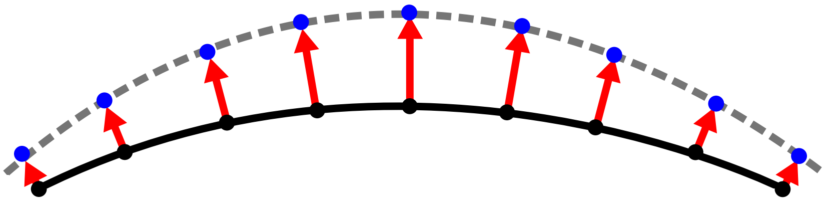


Figure 1.4: Pull tool, red arrows show the vertex normals along which are the vertices transformed to their new position, shown in blue.

1.3.3 Flatten Tool

Sometimes a flat surface is required instead of a curved one, to do this we can use a flatten tool. Its objective is to transform the group of vertices in such a way, that each and every vertex rests on a common 3D plane with the rest of the group, while this common plane is fitted to the average position and average normal vector of vertices in the group, as illustrated by figure 1.5.

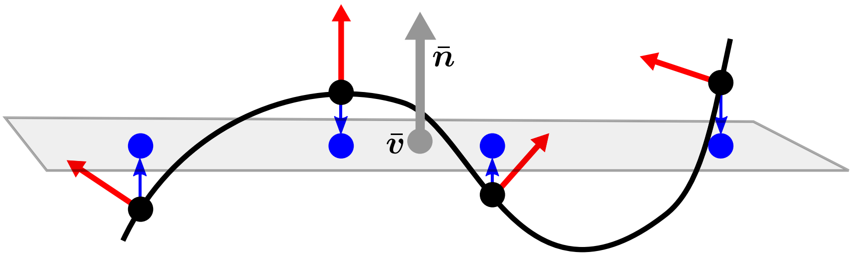


Figure 1.5: Flatten tool, red arrows illustrate vertex normals while the blue arrows show the transformation direction for each vertex to its new position, visualized also in blue.

First we compute the average position $\bar{\boldsymbol{v}}$ and average normal vector $\bar{\boldsymbol{n}}$ of vertices inside the area of interest. After that we form a plane equation using the average values. We can easily calculate the missing constant component d, as illustrated in equation 1.5.

$$\begin{aligned} \bar{\boldsymbol{v}} \cdot \bar{\boldsymbol{n}} + d &= 0 \\ d &= \quad \bar{\boldsymbol{v}} \cdot \bar{\boldsymbol{n}} \end{aligned} \tag{1.5}$$

Now for each vertex we calculate the distance l between the given vertex and the plane. The sign of resulting number shows on which side of the plane the vertex sits on. The transformed position of a vertex can then be computed as described in equation 1.6. Where $\boldsymbol{n}$ denotes the local normal of given vertex and $\rho(\boldsymbol{v})$ is the rate. We subtract the second member of the equation, instead of adding it, to ensure the direction in which the vertex moves points towards the plane.

$$\begin{aligned} l &= \bar{\boldsymbol{n}} \cdot \boldsymbol{v} + d; \\ \boldsymbol{v}' &= \boldsymbol{v} \quad l\rho(\boldsymbol{v})\boldsymbol{n}; \end{aligned} \tag{1.6}$$

1.3.4 Buildup tool

Buildup tool is the most common way to shape a surface. It simulates a brush stroke that adds a layer of material on top of an existing surface. To achieve this, we employ the very same principle we used with flatten tool. However, before forming the plane equation, we move the average vector along the average normal as described by equation 1.7, where s denotes the strength of the tool.

$$\bar{\boldsymbol{v}}' = \bar{\boldsymbol{v}} + s\bar{\boldsymbol{n}} \tag{1.7}$$

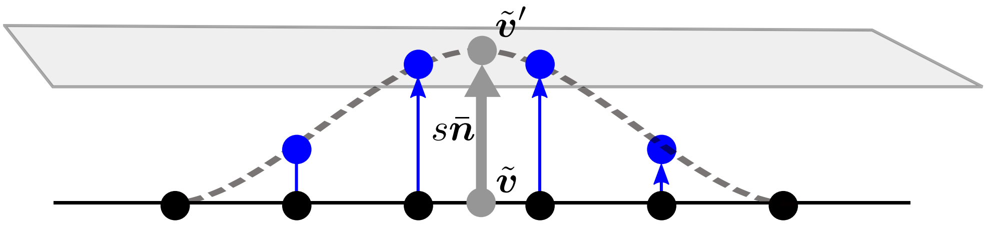


Figure 1.6: Buildup tool illustration that shows vertices being moved towards the imaginary plane at rate depending on their weight.

1.3.5 Smooth Tool

Occasionally we need to fix some previous sculpting operation that went wrong or just simplify the surface. Smooth tool is the ideal solution, as it smooths out the surface and transforms vertices so they have even spacing.

$$\tilde{\boldsymbol{v}} = \frac{1}{|\mathcal{N}(\boldsymbol{v})|} \sum_{\boldsymbol{v_n} \in \mathcal{N}(\boldsymbol{v})} \boldsymbol{v_n} \tag{1.8}$$

$$\boldsymbol{v}' = (1 - \rho(\boldsymbol{v}))\boldsymbol{v} + \rho(\boldsymbol{v})\tilde{\boldsymbol{v}} \tag{1.9}$$

The actual smoothing is done by moving every vertex in the area of interest into the centre of its neighbouring vertices, as described by equation 1.8, where $\mathcal{N}(\boldsymbol{v})$ is the set of neighbouring vertices and $\tilde{\boldsymbol{v}}$ is the target position. To apply the tool smoothly, a linear interpolation is used. This is described in equation 1.9, where the rate $\rho(\boldsymbol{v})$ acts as the coefficient.

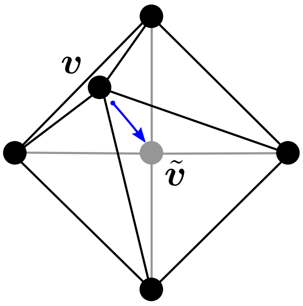


Figure 1.7: Smooth tool moves the vertex $\boldsymbol{v}$ towards $\tilde{\boldsymbol{v}}$, the center of its neighbours.

1.3.6 Drag Tool

Drag tool is quite different from the previous tools. The basic principle is simple: an area of interest is selected and the tool is moved in arbitrary direction while the selected vertices follow the movement. To put it even more simply, the vertices are grabbed and dragged through 3D space.

To calculate the transformed position we need to know how the tool moved from last frame. If we remember the tool's position, we can calculate the offset $\boldsymbol{\Delta c}$. The target position $\tilde{\boldsymbol{v}}$ is then given by the addition of offset to the vertex, as seen in equation 1.10.

$$\begin{aligned} \boldsymbol{\Delta c} &= \boldsymbol{c_{last}} - \boldsymbol{c} \\ \tilde{\boldsymbol{v}} &= \boldsymbol{v} + \boldsymbol{\Delta c} \end{aligned} \tag{1.10}$$

The actual new position is then calculated using linear interpolation, similarly to smooth tool. However, there is no need for elapsed time or strength in the calculation, as the rate of transformation depends on the tool offset.

$$\boldsymbol{v}' = (1 \quad w(\boldsymbol{v}))\boldsymbol{v} + w(\boldsymbol{v})\tilde{\boldsymbol{v}} \tag{1.11}$$

Therefore, only the vertex weight acts as the linear coefficient, as seen in equation 1.11. This is done to maintain the bulge shape, pictured in figure 1.3b. Note, that unlike in the previous tools, there is no need for a local normal vector, as the direction in which the vertex moves completely depends on the tool offset.

1.4 Adaptive Remeshing

Sculpting suffers from several drawbacks. The most important is the progressive loss of detail while the model is being sculpted. The surface expands and triangles become stretched, moreover, in order to model fine details the model must be subdivided to a desired level of detail. To ensure this level of detail and to avoid triangle stretching, we need to employ adaptive remeshing [4]. The difference when it's used or not is illustrated by figure 1.8.

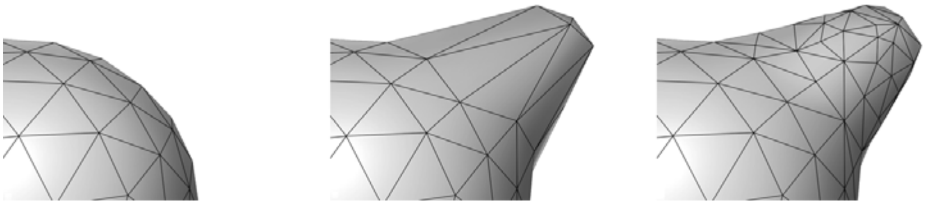


Figure 1.8: On a initial object (left) a sculpting tool is applied (center), which stretches the triangles. Remeshing restructures the mesh (right).

Adaptive remeshing is a process of restructuring a mesh in regard to its changing geometrical shape. To ensure high quality triangular mesh we need to keep the following goals in mind:

- Mesh should be dense enough to be able to represent fine details of the mesh.
- Aspect ratio of all triangles should be close to 1.
- Size of the triangles should be as similar as possible

1.4.1 Remeshing process overview

The remeshing scheme used is similar to [5, 6]. First we split all edges that are longer than a split threshold, then we collapse all edges that are shorter than the collapse threshold. This causes the mesh to keep only edges of a length between the thresholds. While splitting or collapsing edges, new edges are created and existing edges are modified. These edges may meet the condition of being longer or shorter than above thresholds. To ensure they are handled too, both the splitting and collapsing phases run repeatedly until there are no edges that meet the condition.

Moreover, the edges are first sorted by their length before each splitting or collapsing phase. This causes the resulting mesh to have better quality. The details of this process are further explained in the section 3.3.

1.4.2 Edge Split

The split edge operation is performed on long edges. It splits the edge and adds a new vertex. This vertex can be placed either at the edge midpoint or at a position calculated in such way that it retains the curvature of the surface. One of the ways to achieve this is to use the butterfly subdivision scheme, which will be described later in section 1.4.4.

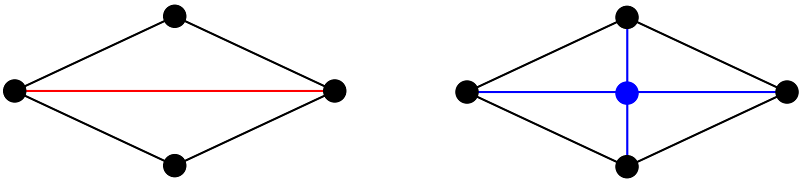


Figure 1.9: Edge split: the long edge (red) is split, subdividing its two adjacent triangles into four (blue)

1.4.3 Edge Collapse

When the edge is shorter than a given threshold, we perform the edge collapse operation. This is done by replacing the edge with a single vertex. We can use the butterfly subdivision scheme again to determine the position of this new vertex.

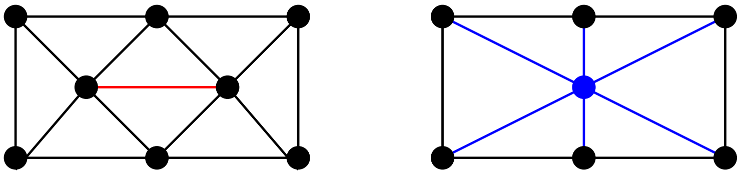


Figure 1.10: Edge collapse: the short edge (red) is collapsed into a new single vertex (blue)

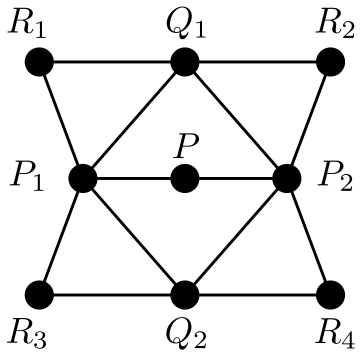


Figure 1.11: Butterfly subdivision scheme: the position of the new point P is calculated from the neighbouring points.

1.4.4 Butterfly Scheme

The new position of vertices after splitting or collapsing edge can be calculated by the butterfly subdivision scheme [7]. Its principle and implementation are very simple and the results are satisfactory for our purposes. Its main purpose is to preserve the curvature of the surface after remeshing. The new vertex position P is determined as:

$$P = \frac{1}{16}(8(P_1 + P_2) + 2(Q_1 + Q_2) - (R_1 + R_2 + R_3 + R_4)) \tag{1.12}$$

Where P_1 and P_2 are end points of edge in question, Q_1 and Q_2 are vertices of adjacent triangles of given edge that are not end points of the edge, and $R_1 ... R_4$ are vertices of adjacent triangles to the triangles incident on the edge. The full scheme is illustrated by figure 1.11.

Chapter 2

Sculpting in Virtual Reality

The difference between the current sculpting applications and sculpting in virtual reality is not only in the way we display the sculpted object and its environment, but mainly in the way how the user interacts with it using input devices. Both used peripherals, the Oculus Rift and the Razer Hydra, have their specifics, which will be described in this chapter. Furthermore, the form of the developed application will be discussed, along with its controls.

2.1 Display

Virtual reality is currently undergoing a revolution, which was started by Palmer Luckey in 2012 when he introduced his prototype of virtual reality head-mounted display called the Oculus Rift. Since then, it gained huge popularity among gamers and technology enthusiasts alike, and after a successful Kickstarter campaign a developer kit was released to public.

A second developer kit is currently scheduled for summer 2014, with consumer version coming in the near future. The company founded by Palmer Luckey, the Oculus VR, has since been heavily invested in and was recently acquired by Facebook. Other VR headsets started to emerge recently to join the virtual reality revolution, such as Sony's project Morpheus or True Player Gear's Totem.

2.1.1 Oculus Rift Overview

Oculus rift is head-mounted display that features a LCD panel with a resolution of 1280 x 800 pixels, a lens for each eye and three sensors: gyroscope, magnetometer and accelerometer. Head-mounted displays have been around for some time, but what is special about Oculus Rift is its low latency head tracking and very wide field of view.

The integration of Oculus Rift into an application consists of two parts. One is to read the orientation data from the headset, the second is to render

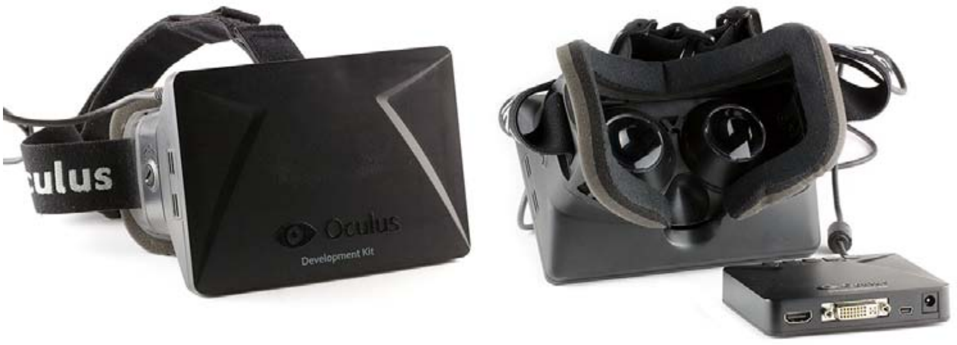


Figure 2.1: Oculus Rift developer kit (courtesy of Sebastian Stabinger [8])

the 3D scene in a very specific way, while the time between the data input and image output must be as short as possible.

The latency is crucial for virtual reality. If the latency is higher than 20 ms, the sense of immersion is broken and motion sickness can occur [9, page 32]. Even lower tresholds were suggested, such as 15 or 7 ms [10].

2.1.2 LCD Display

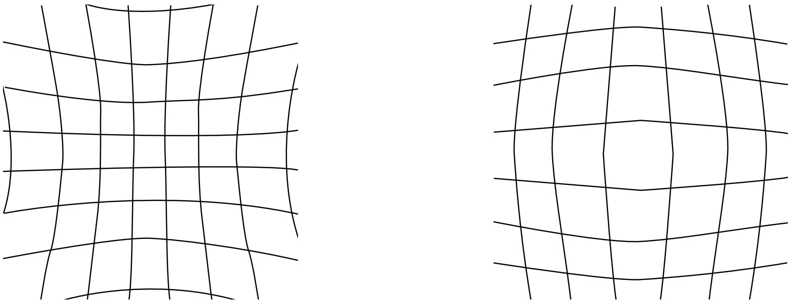
The display has relatively low resolution, even more so as the 720p display is divided by the lenses into two halves, one for each eye. That means that the effective resolution is only 640x800 per eye. Current developer kit also suffers from the screen door effect, i.e., that space between individual pixels is visible, because of the low pixel density of the LCD display. Nevertheless, the state of art is progressing rapidly. The second developer kit is supposed to have resolution of 1920 x 1280 pixels, which eliminates the screen door effect.

2.1.3 Stereoscopic 3D

As mentioned above, Oculus Rift uses single display to show images for both eyes. The image is rendered separately for each eye from a slightly different viewing position to simulate the offset of human eyes. When these two images are combined in the brain, they give the user the perception of 3D depth. As in real world, this perception only applies to objects that are relatively close to the viewer.

2.1.4 Lenses

The lenses serve a single purpose, to widen the field of view. However, the use of lenses causes a number of problems. The two most important are distortion and chromatic aberration.



(a) Pincushion distortion (b) Barrel distortion

Figure 2.2: Lens distortion

The lenses magnify the image of the display but they introduce a pincushion distortion (see figure 2.2a) at the same time. This has to be countered by the rendering software. By rendering the image with the opposite distortion, a barrel distortion (see figure 2.2b), one can cancel out the pincushion distortion, resulting in correct image in the eye of the viewer. [9, page 26]

Because the refractive index of the lenses varies for different wavelengths, the effect known as chromatic aberration occurs. This optical artifact is more visible when we look further from the center of the lens. It causes colored fringes to appear around borders of objects. This is also fixed by the rendering software. [9, page 40]

2.1.5 Sensors

The three sensors inside Oculus Rift, the gyrosope, the magnetometer and the accelerometer, come together in what's called *sensor fusion* to determine the orientation of the head. The Oculus Rift firmware uses predictive head tracking algorithm that allows it to estimate the head orientation upto 50 ms into future, which considerably lowers the latency.

Over a long period of time an error can accumulate in the sensors. This is called the *drift error*. Errors in pitch and roll are automatically corrected by accelerometer that estimates the gravity vector. Yaw drift errors are reduced by magnetometer data, however the device has to be first calibrated, as the correction is done by comparing current data to data of pre-measured points in space [9, page 19].

2.2 Controls

Using traditional input methods, like keyboard and mouse, while using a vritual reality headset can be troublesome. The user cannot see the keyboard and has to interact with it from memory. Therefore gamepads became popular input device for virtual reality games, as they have small amount of buttons and are easy to use. However, while being fairly well suited for games, they fall short on other applications of virtual reality, such as this sculpting application.

2.2.1 Razer Hydra Overview

Luckily, other input devices are available. One of them being Razer Hydra, which consists of two controllers with 6-DOF tracking and a base unit. The 6-DOF refers to six degrees of freedom, specifically, freedom of movement along three axes and rotation along three axes. Razer Hydra also features several buttons and joystick on each controller in layout similar to classical gamepad. The two controllers simulate the movement and orientation of hands, allowing for much more realistic interaction with the virtual world.

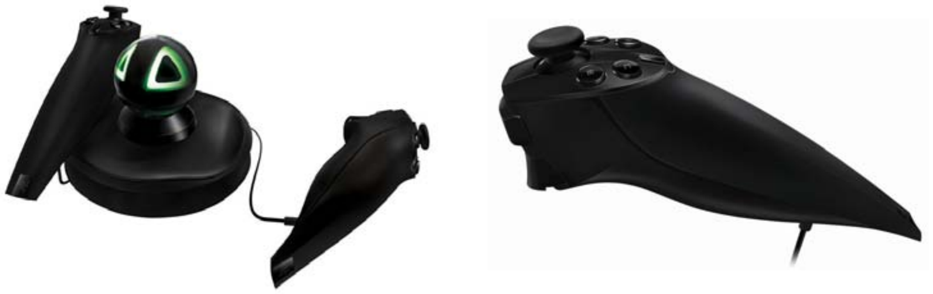


Figure 2.3: Razer Hydra (courtesy of Razer Inc. [11])

The two controllers are connected by a wire to a base, which is connected to PC. The base tracks the absolute position and orientation of the controllers using weak magnetic field with a precision of 1 mm and 1° [12].

2.3 Putting It Together

When modeling an object, we usually don't care about its surroundings since we are focused solely on the object. However, in virtual reality, it would be disturbing to float in empty space with just the object in front of us. The same applies to the input, as we do not see the physical controllers with a headset on, we need them to be in virtual world also.

Therefore, when the user first starts the application, he is put in a virtual environment with a simple floor and a sphere in front him. After a short calibration he can then proceed to sculpt and manipulate the sphere with the controllers that are displayed in front of him. The figure 2.4 shows the screenshot how the application looks like after it's been started. The actual image being rendered for Oculus Rift can be seen in 2.5

The 3D scanned model of Razer Hydra, which was used to visualize the controllers in the virtual world, was kindly provided by Zoltán Erdőkövy [13].

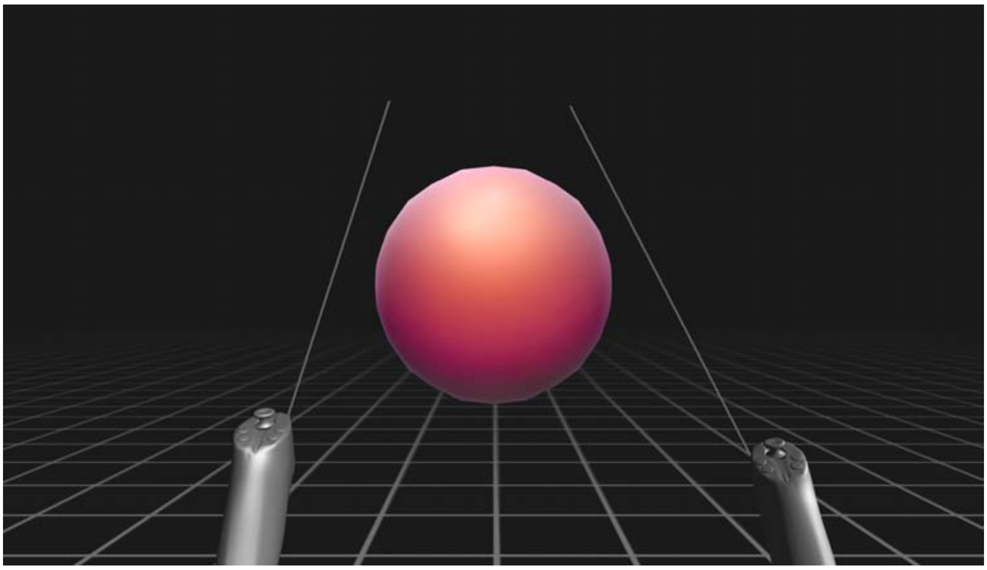


Figure 2.4: Screenshot of the application right after it's been started and calibrated.

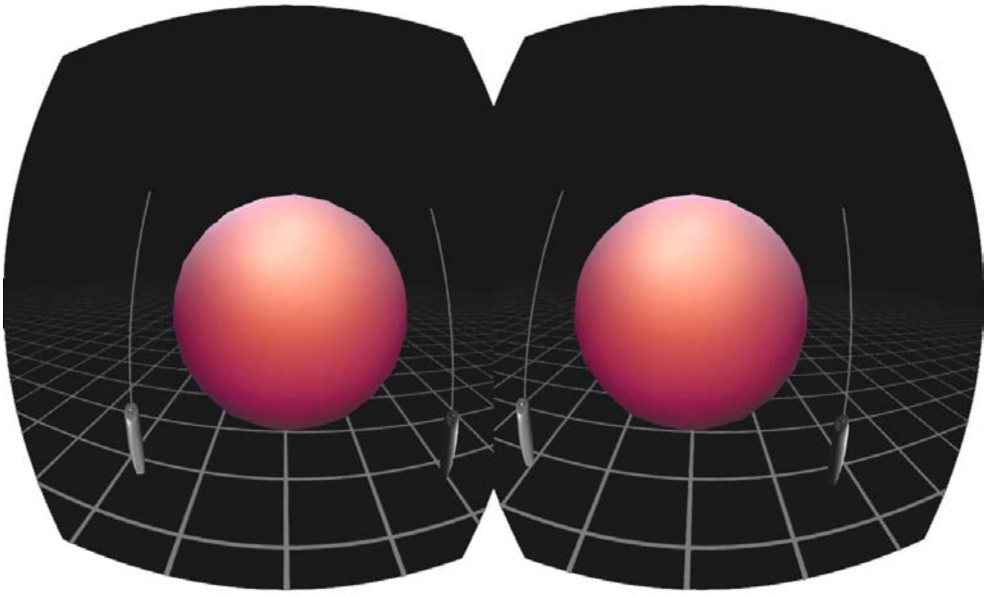


Figure 2.5: Image that is being sent to Oculus Rift: a barrel distortion is applied on each side to correct distortion introduced by the lenses. Each side corresponds to one eye and the view is shifted accordingly.

2.3.1 Sculpting

The actual sculpting is done by aiming the right controller on the object and pressing right *trigger button*. In default mode, the tool is applied only when the controller moves, or more precisely, when the intersection point on the object moves by a certain distance.

By pressing the right *bumper button*, the user can change from default mode to continuous, or alternatively called airbrush, mode. That causes the tool to be applied constantly, even if there is no movement of the controller.

Right start button toggles the inversion mode, which causes the draw and pull tool to transform vertices in opposite direction, and *left button 1* toggles the symmetry mode, which causes the tool to be applied symmetrically on the other side of the object

Using the *right joystick*, the user can change the radius of the tool on the x-axis and strength of the tool on the y-axis.

The individual tools can be switched using numbered buttons on both controllers.

Drag tool, unlike other tools, reacts to relative position or orientation change of the controller to the point of intersection. By moving and rotating the controller, the user can drag the surface arbitrarily. Thanks to adaptive remeshing, the stretched surface will be remeshed and will stay in place as illustrated by figure 2.6.

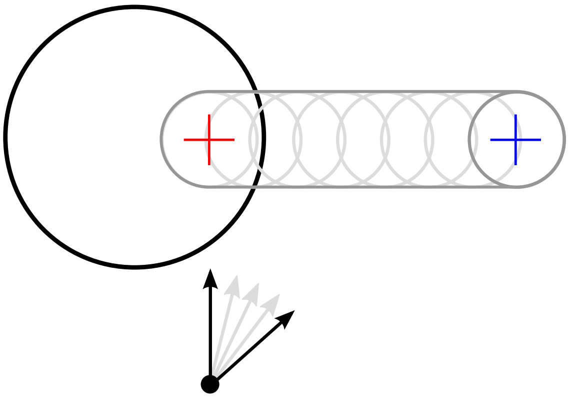


Figure 2.6: Drag tool over time. Red cross shows the initial point of intersection, the blue cross shows the point of intersection after dragging the mesh using just the rotation of the controller.

2.3.2 Manipulation

The user is also able to manipulate the object in several ways. By holding the *left bumper* the object will rotate along its own axis. If the button is released while the controller has enough velocity, the object will continue to rotate. The rotation will slow down and stop shortly after that. By holding the *left bumper* and *right bumper* at same time while increasing or decreasing the distance between the two controllers, the user is able to scale the object.

To move the object or rotate it in other directions, the user can use pivoting. Pivoting is enabled by pressing the *left trigger*. While in pivoting mode, the left controller will act as a pivot and the object will move with it and rotate around it. By pushing the *left joystick* on y-axis, the user can bring the object closer or farther from the pivot. After disengaging the pivoting mode, the object will stay at its current position.

While manipulating the object in any of above ways, the user is able to continue sculpting. This is particularly useful when pivoting or rotation is enabled.

2.3.3 Movement

Apart from moving the object itself, the user can walk around the virtual environment using the *left joystick*. While standing or walking, the user can look around by taking advantage of Oculus Rift 360 head tracking. The direction of movement is primarily in the direction where the *body* is turned. However, when moving forward/backward, the direction is slightly adjusted to consider where the user is looking.

2.3.4 Controls Overview

Table 2.3 shows the basic program controls. However, there are other special and conditioned controls as seen in table 2.4.

Control	**Action**
Joystick X	Forward/backward
Joystick Y	Strafe left/right
Trigger	Enable pivoting
Bumper	Rotate object
Button 1	Symmetry toggle
Button 2	Drag tool
Button 3	Save mesh
Button 4	Wireframe toggle
Start button	Reset mesh

Table 2.1: Left controller

Control	**Action**
Joystick X	Strength +/−
Joystick Y	Radius +/−
Trigger	Apply current tool
Bumper	Airbrush toggle
Button 1	Draw tool
Button 2	Pull tool
Button 3	Smooth tool
Button 4	Flatten tool
Start button	Invert tool

Table 2.2: Right controller

Table 2.3: Controls

Table 2.4: Special controls

Control	**Condition**	**Action**
Right Bumper	Left Bumper is pressed	Scaling mode
Right Joystick X	No intersection point	Turning Left/Right
Left Joystick Y	Pivoting enabled	Zoom object

2.3.5 Calibration

Calibration has to be performed before the user can start sculpting. One reason to do so is the ambiguity of position tracking solution of Razer Hydra which reports two positions: real position and mirrored position across the base unit [12]. To determine the real position, the user must go through a calibration process, which consists of following steps:

1. Point controllers to the base
2. Press trigger on left controller
3. Press trigger on right controller

Another step in calibration is:

4. Put controllers to your shoulders and press both triggers

The last step is used to estimate the distance of the user to the base and also to estimate the position of the users head in relation to the base and controllers.

The first three steps of the calibration are performed by Razer Hydra's API, the last step is custom.

Chapter **3**

Implementation

The following chapter focuses on implementation details, particularly, it describes both the details of the sculpting process and the integration of Oculus Rift and Razer Hydra. It follows and extends the principles described in both previous chapters.

The sculpting implementation consists of three parts: the mesh data structure, i.e., how the mesh is represented, the remeshing process and the implementation details of individual tools. Oculus Rift integration section describes how the application has to be designed to be able to render a 3D scene for a virtual reality headset and the Razer Hydra integration section shows how to implement a position tracking device that is able to interact with virtual clay.

3.1 Overview

3.1.1 Platform and Libraries

The C++ language was used to develop this application, while the primary target platform was Windows. Extending to other platforms is made possible thanks to the choice of open source multiplatform libraries, however, support of other systems is not available out of the box at this time.

The graphics API of choice was OpenGL version 3.2+, which was used for 3D rendering. GLM was library employed as the primary mathematical library, mainly because it is designed to work well with OpenGL and provides tools for 3D math. To create and manage the window and OpenGL context, the GLFW library was chosen, as it is the most modern available library of this type.

3.1.2 Main program loop

Algorithm 1 illustrates the high-level structure of the application. After the initialization, the application stays in what's called a *main loop*, consisting of

Algorithm 1: High-level program structure

```
begin Initialization
    Initialize Oculus Rift
    Create window & OpenGL context
    Initialize Razer Hydra
    Load shaders, meshes and textures
end
repeat
    Update
    Draw
until termination requested;
Free resources;
```

Algorithm 2: Update

```
begin Read data from Oculus Rift
    Read orientation
end
begin Read data from Razer Hydra
    Read position
    Read orientation
    Read buttons pressed
end
Process input
Find point of intersection
if sculpting AND point of intersection exists then
    Apply current tool on the mesh
    Remesh
    Recalculate mesh normals
    Send the modified mesh data to GPU
end
```

updating and drawing, until the user gives the signal to close the application.

Most of the application time is spent in the Update procedure, illustrated by algorithm 2, that runs every frame. First, it reads all available data from both the Oculus Rift and the Razer Hydra and processes them. Then, it finds the point of intersection and applies the current tool if sculpting is enabled. After every tool application, remeshing process is performed and the mesh normals are recalculated. Finally, the modified mesh is sent to the GPU.

3.2 Mesh Data Structure

The remeshing process heavily utilizes two operations, edge split and collapse. Therefore, the mesh data structure had to be designed in such a way, that these operations are as efficient as possible. Specifically, finding incident edges, adjacent triangles and neighbouring vertices had to be as fast as possible. Moreover, addition and removal of these edges, triangles and vertices is *vertices*

Vertex	Coords
A	$(1, 1, 1)$
B	$(1, -1, -1)$
C	$(-1, 1, -1)$
D	$(-1, -1, 1)$

triangles

Triangle	Verts
a	(B,C,D)
b	(A,C,D)
c	(A,B,D)
d	(A,B,C)

edges

Edge	Verts	Triangles
i	(A,B)	(b,d)
j	(C,D)	(a,b)
k	(A,D)	(b,c)
l	(A,B)	(c,d)
m	(B,C)	(a,d)
n	(B,D)	(a,c)

vertexEdges

Vertex	Edges		
A	i	k	l
B	l	m	n
C	i	j	m
D	k	j	n

vertexTriangles

Vertex	Triangles		
A	b	c	d
B	a	c	d
C	a	b	d
D	a	b	c

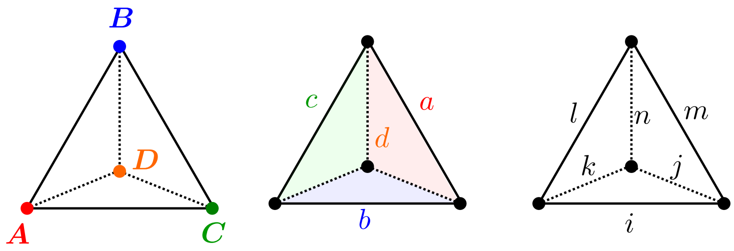


Figure 3.1: Mesh data structure breakdown for a tetrahedron. Indices are denoted by letters instead of integers.

performed in both edge split and edge collapse, thus these operations had to be taken in account also.

3.2.1 Overview

The mesh data structure includes three primary arrays of data: *vertices*, *edges* and *triangles*. It also includes two auxiliary arrays of arrays: *vertexEdges* and *vertexTriangles*. One additional array, *normals*, is also included. All the mentioned arrays are implemented using the STL vector container. The data structure breakdown is shown in figure 3.1.

Vertices array contains simple 3-tuples of floats, representing vertex coordinates. The two extra arrays, *vertexEdges* and *vertexTriangles*, keep the information about to what edges and triangles given vertex belongs. *Normals* is an array of simple 3-tuples of floats that store the surface normal vector for each vertex. They are recalculated after every change to the mesh using method discussed earlier in section 1.3.1. Triangles keep three integer indices of their vertices, referring to their position in the *vertices* array. *Edges* array contains edges, structures of two pairs: one pair of vertex indices and one pair of adjacent triangle's indices. Non-manifold edges are therefore not supported.

3.2.2 Adding entities

Adding vertices or edges is straightforward. It is only matter of adding given entity at the end of the respective array and adding the interconnection information. For example, in case of an edge, we first insert the edge at the end of the *edges* array and then, for both vertices, we add it to *vertexEdges* subarrays denoted by the vertex indices, so each vertex knows which edge it belongs to.

Adding a whole triangle is slightly more complicated. Apart from adding the new triangle to *triangle* array and connecting it to its vertices it also has to go through all its edges and either create new or modify existing ones.

3.2.3 Removing entities

Because we remove individual entities fairly often, it must be done effectively. However, the vertices, edges and triangles are stored in an one dimensional array, thus simply deleting the entity and moving the rest of the array is out of question.

A different approach to deleting was chosen and it is very similar to how the deleting works in a hash table data structure. The complexity of the operation is constant, however the data have to be consolidated from time to time (in this case after every remesh).

Basically, a tombstone is created in place of the entity when it's deleted. This is done differently for each entity. The edge is considered removed when its end vertices are equal. The triangle is considered removed when at least two of its indices are equal. However, vertices aren't being deleted, they just stop being referenced if there is neither a valid edge nor triangle containing them.

Algorithm 3: Process of splitting all edges longer than a threshold

```
Input: threshold
repeat
    edgesToSplit = empty array
    for each e in edges do
        if $length^2(e) > threshold^2$ then
            add e to edgesToSplit
        end
    end
    sortedEdges = sortByLength(edgesToSplit, descending)
    for each e in sortedEdges do
        if $length^2(e) > threshold^2$ then
            split(e)
        end
    end
until no split occured
```

3.3 Remeshing Process

The remeshing process is performed after every change made to the mesh. The process consists of a split phase, collapse phase and consolidation.

3.3.1 Split & Collapse Overview

Both split and collapse phases of above process run repeatedly on sorted arrays of edges to achieve equal triangle size and triangle aspect ratio close to 1. The split phase is illustrated by algorithm 3.

The collapse phase is almost identical to algorithm 3. However, the comparing condition is *lesser than* and the sorting function is *ascending.*

The edge length is checked twice in each iteration, as some of the edges that are to be split/collapsed will change in length during the phase and they might not meet the condition again. The vector length is not fully computed as $||\boldsymbol{v_0} - \boldsymbol{v_1}||$, but as $||\boldsymbol{v_0} - \boldsymbol{v_1}||^2$ to avoid the costly square root function. It is then compared to the threshold length, which is squared. This is possible because squaring preserves the order relation.

3.3.2 Edge Split Algorithm

The split edge operation is divided in four steps as illustrated by figure 3.2.

1. Remove edge (upper left)
2. Remove adjacent triangles t_a and t_b (upper right)
3. Add new vertex $\boldsymbol{v_c}$ (lower left)
4. Add triangles $(\boldsymbol{v_c}, \boldsymbol{v_a}, \boldsymbol{v_0})$, $(\boldsymbol{v_c}, \boldsymbol{v_1}, \boldsymbol{v_a})$, $(\boldsymbol{v_c}, \boldsymbol{v_0}, \boldsymbol{v_b})$ and $(\boldsymbol{v_c}, \boldsymbol{v_b}, \boldsymbol{v_1})$ (lower right)

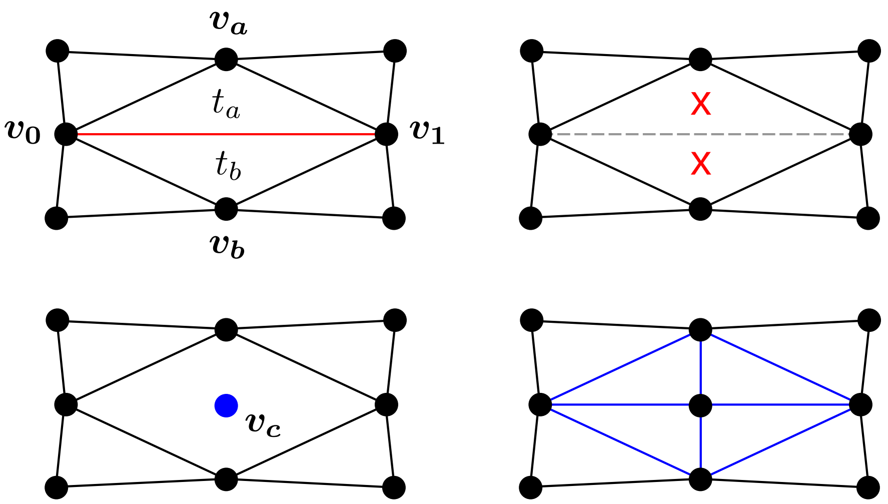


Figure 3.2: Edge split algorithm

3.3.3 Edge Collapse Algorithm

Collapsing an edge is slightly more complicated than splitting, mainly because more neighbouring vertices, edges and triangles are involved. The collapse operation is performed in 7 steps, which are illustrated by figure 3.3.

1. Remove edge (upper left)
2. Remove adjacent triangles t_a and t_b (upper right)
3. Remove edges $(\boldsymbol{v_0}, \boldsymbol{v_a})$, $(\boldsymbol{v_1}, \boldsymbol{v_a})$, $(\boldsymbol{v_0}, \boldsymbol{v_b})$ and $(\boldsymbol{v_1}, \boldsymbol{v_b})$ (upper right)
4. Add new vertex $\boldsymbol{v_c}$ (lower left)
5. Add new edges $(\boldsymbol{v_c}, \boldsymbol{v_a})$ and $(\boldsymbol{v_c}, \boldsymbol{v_b})$ (lower left)
6. Replace $\boldsymbol{v_0}$ and $\boldsymbol{v_1}$ with $\boldsymbol{v_c}$ in triangles adjacent to $\boldsymbol{v_0}$ and $\boldsymbol{v_1}$, respectively (lower right)
7. Replace $\boldsymbol{v_0}$ and $\boldsymbol{v_1}$ with $\boldsymbol{v_c}$ in edges connected to $\boldsymbol{v_0}$ and $\boldsymbol{v_1}$, respectively (lower right)

The vertices $\boldsymbol{v_0}$ and $\boldsymbol{v_1}$ are not explicitly removed, they just stop being referenced, as explained in section 3.2.3, and are deleted in the consolidation phase later.

Because more neighbouring vertices are involved in the edge collapsing than in edge splitting, we cannot recklessly follow the above algorithm for every edge. Some special cases must by observed and handled. These special

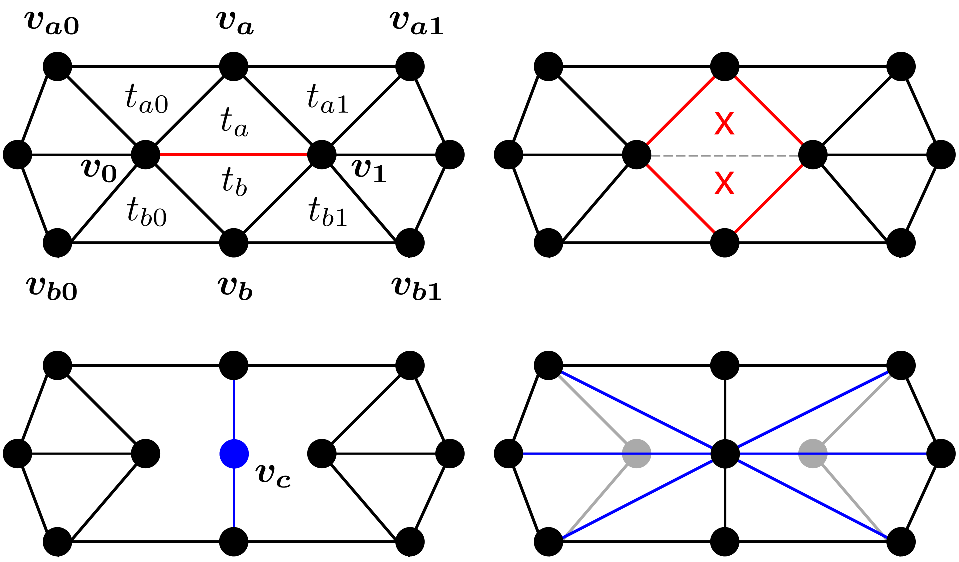


Figure 3.3: Edge collapse algorithm

cases are a specific configuration of the vertices that would cause problems if they weren't dealt with.

One of these special cases is when $v_{a0} = v_{a1}$ or $v_{b0} = v_{b1}$. If not handled, it would lead to a non-manifold edge, which would break the data structure.

Another special case is when v_0 and v_1 shares an edge with a vertex that is connected to v_1 or v_0, respectively, and that vertex isn't v_a nor v_b. This can't be seen properly in 2D, but considering the figure 3.3, such connection would go *behind* the displayed triangle mesh. Multiple problems arise when this case is not observed, such as invalid interconnection information which can lead to visual artifacts on rendered mesh.

I found, that instead of trying to handle above special cases by creating a branch of code that would do the edge collapse separately, *canceling* the operation is far easier way to solve this problem. The repeated process of collapsing ensures that all short edges are eventually collapsed, even if these cases are skipped. Also, if the ideal edge length is set very high, the mesh will degenerate into a tetrahedron.

3.3.4 Consolidation

The consolidation step is necessary because of the way how removed vertices, edges or triangles are handled, which is illustrated in section 3.2.3. Basically, after each remeshing, number of vertices become unreferenced and many edges and triangles are made obsolete. The consolidation rebuilds the mesh data structure so only relevant entities are present.

Algorithm 4: Consolidation

```
/* Copy old entities                                                  */
oldTriangles = triangles
oldVertices = vertices

/* Clear all mesh arrays                                              */
clearArrays()
isReferenced = empty array
newIndex = empty array

for i=0 to sizeof(vertices) do
    add false to isReferenced
    add -1 to newIndex
end
for each t in oldTriangles do
    if wasRemoved(t) then continue
    for each $v_i$ in t do                  /* For each vertex index $v_i$ in t */
        if isReferenced[$v_i$] == true then continue
        /* addVertex re-adds the vertex and returns its new index.     */
        newIndex[$v_i$] = addVertex(vertices[$v_i$])
        isReferenced[$v_i$] = true
    end
    addTriangle(newIndex[t[0]], newIndex[t[1]], newIndex[t[2]])
end
```

The consolidation is done by re-adding existing non-deleted triangles into an empty mesh data structure. Also, only referenced vertices are re-added and these vertices need to be re-indexed. This is illustrated by algorithm 4.

3.4 Sculpting Tools

When applying a tool, all vertices are checked whether they fall into the area of interest, which is determined by the tool radius and point of intersection, and then a transformation is applied with a rate depending on their distance to the point of intersection and strength of the tool.

3.4.1 Tool design

Inheritance and polymorphism has been utilized in design of the tools as they share a common interface. The base class *MeshTool* includes common attributes like radius, strength, etc. and also an abstract function that applies given tool. This *apply* function is then overridden in every tool with its own implementation. A class diagram that shows the relationships, attributes and methods of all tools is shown in figure 3.4.

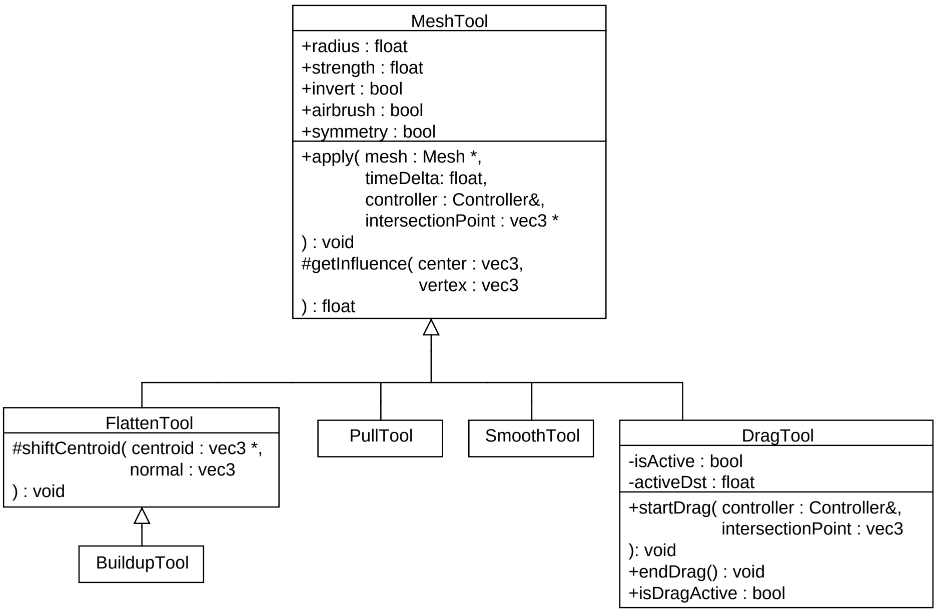


Figure 3.4: Tool class diagram

3.4.2 Pull Tool

The pull tool is the simplest of the implemented tools. The actual process is straightforward; it iterates over all vertices and transforms them according to equation 1.4.

3.4.3 Flatten Tool

Algorithm 5 illustrates how the flatten tool transform vertices. First, all affected vertices are saved in arrays, then the imaginary plane is computed, and finally all affected vertices are transformed depending on their distance to that plane. The vertices are saved to avoid iterating over all vertices again.

3.4.4 Buildup Tool

As discussed earlier, the buildup tool is only a variation of the flatten tool. Thanks to this, only small part of the code has to be changed, specifically, when calculating the imaginary plane, the average position is modified as described in equation 3.1, where s denotes the tool strength.

$$\boldsymbol{c} = \boldsymbol{c} + s\boldsymbol{n} \tag{3.1}$$

Algorithm 5: Flatten tool

```
$verts$ = empty array
$\boldsymbol{c} = (0, 0, 0)$
$\boldsymbol{n} = (0, 0, 0)$
for $v_i$ = 0 to sizeof(vertices) do
    $\boldsymbol{v}$ = vertices[$v_i$]
    if $\rho(\boldsymbol{v}) > 0$ then
        /* Add vertex to centroid */
        $\boldsymbol{c} = \boldsymbol{c} + \boldsymbol{v}$
        $\boldsymbol{n} = \boldsymbol{n} + normals[v_i]$
        add $v_i$ to $verts$
    end
end
/* Calculate average vertex position and normal */
$\boldsymbol{c} = \boldsymbol{c}$ / sizeof($verts$)
$\boldsymbol{n} = \boldsymbol{n}$ / sizeof($verts$)
/* Construct plane */
$d = -\boldsymbol{c} \cdot \boldsymbol{n}$
for $i$ = 0 to sizeof(verts) do
    $v_i$ = verts[i]
    $\boldsymbol{v}$ = vertices[$v_i$]
    $l = -(\boldsymbol{v} \cdot \boldsymbol{n} + d)$
    vertices[$v_i$] = $\boldsymbol{v} + l\rho(\boldsymbol{v})\boldsymbol{n}$ ;
end
```

3.4.5 Smooth Tool

Smooth tool is the only tool that makes use of the interconnection between the vertices as it needs all neighbours of a given vertex. As seen in algorithm 6, it iterates over all vertices and for each vertex calculates a centroid of its neighbours and then moves the vertex to this centroid at a rate determined by tool strength and weight of the vertex and elapsed time.

Note that the new positions of vertices are stored in a separate array, which replaces the old array after all vertices have been processed. This is because all the new positions have to be calculated from vertices in the state of a mesh that has not yet been modified.

3.4.6 Drag Tool

The drag tool is slightly different from other tools as it also needs the information when and how the dragging started and when it ended. Therefore it implements two more functions, *startDrag* and *endDrag*. Apart from that, it also uses the information about the position and orientation of the controller in previous frame. Specifically, the ray coming out of the controller. The ray calculation method is described further in this chapter, in section 3.5.4.

The *startDrag* function computes the initial distance between controller and the point of intersection. This distance is then used in the entire process of dragging, as can be seen in algorithm 7, where it acts as the d input variable.

Algorithm 6: Smooth tool

```
$newVertices$ = empty array
for $v_i = 0$ to sizeof(vertices) do
    $\boldsymbol{v} = vertices[v_i]$
    /* Number of neighbours */
    k = sizeof($vertexEdges[v_i]$)
    $\boldsymbol{c} = (0, 0, 0)$
    for $i=0$ to $k$ do
        $e_i = vertexEdges[v_i][\mathrm{i}]$
        $e = edges[e_i]$
        /* Find the vertex index of the edge $e$ other than $v_i$ */
        $v_0 = e$.vertices[0]
        if $e$.vertices[0] == $v_i$ then $v_0 = e$.vertices[1]
        /* Add the vertex to the centroid */
        $\boldsymbol{c} = \boldsymbol{c} + vertices[v_0]$
    end
    /* Calculate target position */
    $\tilde{\boldsymbol{v}} = \boldsymbol{c}$ / k
    /* Calculate new position */
    $\boldsymbol{v}' = (1 - \rho(\boldsymbol{v}))\boldsymbol{v} + \rho(\boldsymbol{v})\tilde{\boldsymbol{v}}$
    add $\boldsymbol{v}'$ to $newVertices$
end
$vertices = newVertices$
```

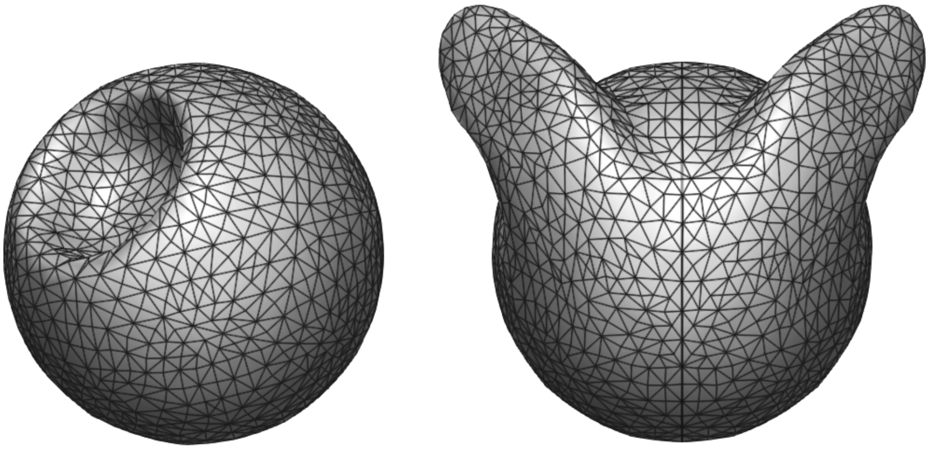
Algorithm 7: Drag tool

```
Input: $d$, controller
$r$ = controller.currentRay
$u$ = controller.lastRay
/* Calculate offset $\boldsymbol{o}$ */
$\boldsymbol{a} = \boldsymbol{r_o} + d * \boldsymbol{r_d}$
$\boldsymbol{b} = \boldsymbol{u_o} + d * \boldsymbol{u_d}$
$\boldsymbol{o}$ = -($\boldsymbol{b}$ - $\boldsymbol{a}$)
for $v_i = 0$ to sizeof(vertices) do
    $\boldsymbol{v} = vertices[v_i]$
    $vertices[v_i] = (1 - \rho(\boldsymbol{v}))\boldsymbol{v} + \rho(\boldsymbol{v})(\boldsymbol{v} + \boldsymbol{o})$
end
```

3.4.7 Tool Modes

In *default mode* the tool is applied only when the point of intersection (POI) moves by a certain amount, so called *minimal step*, in contrast to the *airbrush mode*, where it is applied always. The exact distance is computed as the the distance of POI where the tool was last applied, which is, however, transformed by the last tool transformation, and the current POI. If the last POI wasn't transformed, the tool would continue to be applied even though the controller didn't move. The rate at which the tool is applied is not affected by the elapsed time.

In the *airbrush mode* the tool is applied every frame, regardless of controller movement. However, the rate at which the tool is applied is influenced by



(a) Inversion (b) Symmetry

Figure 3.5: Tool modes

the elapsed time Δt. Both *default* and *airbrush mode* don't apply to the drag tool, as it works on a different principle.

The *inversion mode* causes the tools to transform the vertices in the opposite direction, therefore creating a dimple or a hole. Figure 3.5a shows how the mesh looks like after the application of an inverted tool. *Inversion* applies only to buildup and pull tool.

The last, and perhaps the most useful tool mode is the *symmetry mode*. If it's enabled, the tool is applied around the POI and also around the POI that is mirrored along the yz plane, as is illustrated by figure 3.5b. If a vertex that is to be transformed resides in both the standard and mirrored area, the transformation has to be modified accordingly. The implementation varies slightly for every tool, specifically when the area of interest and the mirrored area aren't disjoint.

Let f be the weight depending on standard POI and let g be the weight depending on mirrored POI. Then the final weight of a vertex is computed as follows:

- Pull tool : $weight = (f + g)/2$
- Smooth tool : $weight = max(f, g)$

In case of buildup tool, the transformations are applied in the same way on the mirrored area, however the imaginary plane is computed separately for this area and vertices are transformed according to this separate plane.

The drag tool symmetry computation proceeds as usual if the vertex is only in the standard area. If not, specifically, if $g > f$, then the vertex weight is set to g and the offset $\boldsymbol{o}$ is flipped along the yz plane. That is done by changing the sign of its x coordinate.

3.5 Integration of Razer Hydra

The Razer Hydra is commonly used for games and as it pre-dates the Oculus Rift, it was commonly used for games, which developed for keyboard and mouse, and introduced so called *gestures*, which translated simple movements of the controllers to various game commands. However, since the Oculus release, it has been heavily used in various virtual reality demos. The *gestures* are not so important in virtual reality, as the main reason why these controllers are used is to simulate real hands and track their position and orientation.

3.5.1 Integration

The interface for Razer Hydra, called Sixense API, is a set of C functions that allow reading data from the device. It also features number of utilities that simplify the integration.

One of these utilities is the *ControllerManager*, which facilitates part of the calibration process. It takes care of determining which hemisphere surrounding the controller base is the correct one, as the controllers report both correct and mirrored position across the base. The integration of this utility is fairly simple, as it only needs to receive the controller data every frame and in response it returns a message which should be displayed to the user.

However, before the application can receive any data, it has to initialize the API. This is done by calling the *sixenseInit()* function. After that controller data are available. Upon closing the application *sixenseExit()* has to be called to free allocated resources.

The data is read by the *sixenseGetAllNewestData()* function that returns all available data for both controllers. This includes position, orientation, button states but also the hardware index of the controller or whether the controller is docked at the base or not.

3.5.2 Custom Calibration

For the purposes of this application, a custom calibration and initialization had to be added. The first important step is to determine which controller is the right and which is the left. This is done simply by comparing their coordinates. The coordinates also serve for calculation of distance of controllers to the base. This distance is then used to position the mesh in an ideal place and also to position the virtual head, in other words, the camera, in correct place after the calibration. It is also used to modify the filtering parameters.

3.5.3 Filtering

The filtering parameters control the filtering, in other words smoothing, of the controller position. If the filtering would be too weak, the position wouldn't be steady and would flicker, if too strong, the position would be smooth, but would lag behind considerably. The default parameters proved to be too weak, so the following values were chosen for ideal smoothing:

- $nearRange = 0.4 * d$
- $nearValue = 0.945$
- $farRange = 1.4 * d$
- $farValue = 0.985$

Where d is the measured distance from controller to base. The $nearRange$ and $farRange$ determine the range where the filtering rate should be interpolated between $nearValue$ nad $farValue$. For ranges smaller or bigger than $nearRange$ or $farRange$, the $nearValue$ and $farValue$ are used, respectively. The complete explanation of the filtering can be found in [12, page 24].

3.5.4 Orientation

The default orientation of the controllers doesn't match exactly the direction at which the controllers are *pointing*. Specifically, the direction of the ray determined from the controller's default position and orientation does not point from the front of the controller, where trigger and bumper buttons are located. To correct this, the orientation of the controllers has to be rotated by 30° along the x axis.

The mentioned ray r coming out of the controller is calculated using equation 3.2, where $\boldsymbol{r_o}$ is the ray origin, $\boldsymbol{r_d}$ is ray direction vector, $\boldsymbol{P}$ is the controller position and finally $\boldsymbol{R}$ is the a 3×3 rotation matrix specifying the controller orientation. The vector $(0, 0, -1)$ is the *forward* direction in the application's coordinate system.

$$r = (\boldsymbol{r_o}, \boldsymbol{r_d}) = (\boldsymbol{P}, \boldsymbol{R} \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix}) \tag{3.2}$$

Other than calculating rays coming out of the controllers and processing the individual buttons to change the state of the application accordingly, the object manipulation is computed using the relative position and orientation change of the controllers.

3.5.5 Object Rotation

To rotate the object, the *left bumper button* must be pressed. While pressed, the left controller can be used to rotate the object around its pivot. This is done by moving the controller around the object while the object orientation follows.

The calculation used is similar to arcball technique used for mouse [14]. The actual calculation is shown in equation 3.3, where $\boldsymbol{x}$ is the rotation axis and α is the angle. They are calculated from the controller position $\boldsymbol{P_c}$, position of the controller in last frame $\boldsymbol{P_l}$ and the object's pivot $\boldsymbol{Z}$.

$$\begin{aligned} \boldsymbol{a} &= \boldsymbol{P_c} - \boldsymbol{Z} \\ \boldsymbol{b} &= \boldsymbol{P_l} - \boldsymbol{Z} \\ \boldsymbol{x} &= \boldsymbol{a} \times \boldsymbol{b} \\ \cos\alpha &= \frac{\boldsymbol{a} \cdot \boldsymbol{b}}{\|\boldsymbol{a}\|\|\boldsymbol{b}\|} \end{aligned} \tag{3.3}$$

After the *left bumper button* is released, the rotation continues if the velocity of the controller movement was high enough. This done to simulate the inertia of the object. To do this, several previous positions of the controller have to be recorded in a queue. Upon the button release, an average vector is computed from these stored positions. The oldest position stored and also the oldest position, translated by the average vector, act as $\boldsymbol{P_l}$ and $\boldsymbol{P_c}$, respectively, in the equation 3.3 to calculate the axis and angle of the rotation. The length of this average vector is then used as the velocity of the rotation. This velocity is decreased linearly in every frame until it's zero.

3.5.6 Object Scaling

To scale the object, the user first has to press *left bumper button* and then *right bumper button* and keep them pressed while moving controllers away or towards each other. The amount of scale is then determined by the ratio of two distances of the two controllers: the distance at the start of the scaling d and the distance in the current frame d'.

$$s' = s\frac{d'}{d} \tag{3.4}$$

While scaling is active, the new object scale s' is calculated by multiplying the scale at the start of the scaling, denoted by s, by the mentioned ratio, in every frame, as illustrated by equation 3.4.

3.5.7 Object Pivoting

While in pivoting mode, the left controller will act as a pivot and the object will move with it and rotate around it. This mode is toggled by pressing the *left trigger button.* At the start of the pivoting, the controller position $\boldsymbol{P}$ and orientation $\boldsymbol{Q}$ is stored. During the pivoting, the difference between current state and start state is calculated.

The difference in position $\boldsymbol{\Delta P}$ is calculated simply as the difference between current position $\boldsymbol{P'}$ and start position $\boldsymbol{P}$, as illustrated by equation 3.5. On the other hand, the orientation difference $\boldsymbol{\Delta Q}$ is calculated as difference between two *quaternions*, current orientation $\boldsymbol{Q'}$ and start orientation $\boldsymbol{Q}$. This is done by multiplying the current quaternion with the inverse of the start quaternion, as shown in equation 3.6.

$$\boldsymbol{\Delta P} = \boldsymbol{P'} - \boldsymbol{P} \tag{3.5}$$

$$\boldsymbol{\Delta Q} = \boldsymbol{Q'}\boldsymbol{Q}^{-1} \tag{3.6}$$

$$\boldsymbol{R_p} = \boldsymbol{T}(\boldsymbol{P'})\boldsymbol{R}(\boldsymbol{\Delta Q})\boldsymbol{T}(-\boldsymbol{P'}) \tag{3.7}$$

$$\boldsymbol{M'} = \boldsymbol{R_p}\boldsymbol{T}(\boldsymbol{\Delta P})\boldsymbol{M}; \tag{3.8}$$

In the above equations, $\boldsymbol{T}(\boldsymbol{v})$ denotes a translation matrix, which translates by vector $\boldsymbol{v}$, and $\boldsymbol{R}(\boldsymbol{q})$ denotes a rotation matrix, which rotates by quaternion $\boldsymbol{q}$. All above calculations are made in homogeneous space, therefore all matrices are 4 × 4.

The rotation around pivot, denoted by matrix $\boldsymbol{R_p}$, is calculated by rotating in the coordinates of the pivot, in other words, around current controller position. The amount and direction of the rotation is defined by $\boldsymbol{\Delta Q}$. This is illustrated by equation 3.7. The equation 3.8 describes how the transformations are put together to obtain the final model transformation $\boldsymbol{M'}$ of the object. The $\boldsymbol{M}$ denotes the model matrix prior to pivoting, $\boldsymbol{T}(\boldsymbol{\Delta P})$ controller offset translation and $\boldsymbol{R_p}$ the pivot rotation.

3.6 Integration of Oculus Rift

The OculusVR SDK version 0.2.5c was used, however, at the time of writing, a new version, 0.3.1, was released. The version that was used offers only object oriented C++ API, which proved to be difficult to work with. A strictly C API would have been preferred, as OculusVR realized too, so the latest version features such C interface. The latest version also takes care of distortion rendering, which had to be implemented manually.

3.6.1 Integration

Before any data can be read from the Oculus Rift, the API has to be initialized. This is done by series of initialization commands and by creating number of objects, such as a *DeviceManager*, a custom *message handler*, *SensorDevice*, *HMDDevice*, etc. The freeing of resources is similarly cumbersome, as every object has to be destroyed. This is partly done by the API's own allocator, partly manually.

Moreover, the API is primarily focused on DirectX, so integrating it with OpenGL introduces some difficulties. On top of that, the provided example in the SDK has been written for DirectX and OpenGL simultaneously, making use of a wrapping mechanism, which proved to be troublesome to decipher,

Algorithm 8: Oculus Rift program structure

```
begin Initialization
    Initialize OculusVR API
    Create render target
end
repeat
    Read sensor data
    begin Draw
        Render target ← texture
        for left and right eye do  render scene
        Render target ← screen
        for left and right eye do  render distorted image
    end
until termination requested
Free resources
```

as the documentation doesn't cover all the details. Eventually, I overcame all mentioned hurdles and managed to succesfully integrate the API.

3.6.2 Oculus Rift Program Structure

Algorithm 8 shows the minimal basic program structure needed to integrate Oculus Rift. After initialization, in the main program loop, the application reads the sensor data and renders the image accordingly. First the scene is rendered to a texture and then this texture is rendered while the distortion is applied.

3.6.3 Data Retrieval

To read the orientation data is fairly straightforward. The *SensorFusion* object's method returns the orientation quaternion for the predicted orientation. The orientation is predicted to minimize the latency of the sensors. Thanks to that, the retrieval is almost instantaneous. The API also offers the orientation in Euler angles, which is useful for calculating the yaw angle, as it is not only affected by where the user is looking, but also by the orientation of the virtual body. Thus, the yaw angle is kept separate to be modified later depending on user input.

3.6.4 Rendering Scene

The first part of rendering for Oculus Rift is to draw the scene to a buffer in the memory, called render target, which is then used as a standard texture. Each eye is rendered separately, with an offset, to ensure stereoscopic vision and thus the depth perception.

The actual rendering process is almost the same as it would be for normal 2D display with a couple of differences:

- Projection matrix is modified
- View matrix is modified
- Rendering is done only on left or right half of the viewport at a time

The projection matrix represents a standard perspective projection, with a field of view calculated with regard to the *resolution* of the display, *screen size* and *eye to screen distance* [9, page 23-24]. The standard perspective projection is set up for the center of the screen, however, to correctly display it for Oculus Rift, the projection has to be shifted to coincide with the center of the lens. This is done by applying a translation transformation that depends on *lens seperation distance*, a value that can be retrieved from the API. The complete calculation can be found in [9, page 24].

The view matrix is calculated depending on the headset orientation, which is read from the sensors earlier, and is then shifted to accommodate for the distance between the left and right eye. This is also done by a simple translation transformation. The amount by which the view is shifted depends on the *interpupillary distance* (IPD), a distance between the eyes that differs for each person. This distance should be specified a priori in Oculus configuration utility. If there is a mismatch between the used IPD and the actual person's IPD, the virtual reality experience may be uncomfortable, may strain the eyes and the sense of scale may be altered [15].

3.6.5 Rendering Distortion

The second part of rendering is distortion. To accommodate for the distortion introduced by lenses, a barrel distortion has to be applied, as described in section 2.1.4. This is done by using a special distortion shader [9, page 27]. A simple rectangle spanning the entire screen is rendered using this shader, which uses the previously mentioned texture with rendered scene. The shader distorts and positions the texture accordingly. The API provides the parameters for the distortion, such as *lens center*, or *warp function coefficients*, which control the amount of distortion. Several other parameters have to be calculated, such as *scale* or *screen center*, to correctly position the rendered image.

The barrel distortion pulls pixels towards its center. Because of this, the edges of the screen would have unused blank space and considerable amount of field of view would be lost. To avoid this, the scene is first rendered in higher resolution than the actual screen can display. The ideal scale factor for the resolution can be also retrieved from the API. The shader then scales and positions the distorted image to fill as much space as possible [9, page 28].

The chromatic aberration correction is also done by the above shader. It works on the same principle as the distortion, however it takes different set of parameters and each color is transformed separately according to these parameters.

Chapter 4

Testing and Discussion

The application was first developed for a standard 2D monitor screen to allow for easier debugging. First the sculpting process was implemented, with a mouse as a controller, following the Razer Hydra integration and finally Oculus Rift integration. The 2D screen mode is available in the application if no Oculus Rift device is connected.

4.1 Testing

Overall, the application has been tested to contain no memory leaks. This was critical, considering that the remeshing process reallocates big amount of memory every frame, which in case of a memory leak would inevitably lead to a crash of the application.

The testing of the sculpting process was done both by actual sculpting and programmatically to accommodate for all possible combinations. Therefore, the remeshing process can be considered stable. There was very little testing involved in the integration of Razer Hydra, other than finding suitable button bindings, thanks to the API being very straightforward. The Oculus Rift integration has been tested against the open source sample application in the OculusVR SDK, specifically, to make sure that all common calculations are matching and that they provide the same result.

4.2 Optimizations

One of the challenges I faced, was to optimize the application and maximize the performance. As with any graphical application, the main challenge is to ensure high and steady frame rate. However, when dealing with a virtual reality headset, the frame rate is even more crucial. Any temporary or permanent drop in the frame rate will cause very uncomfortable feeling and might even cause nausea [15]. The current version of Oculus Rift is designed to run at 60 Hz, in other words, 60 frames per second.

The usual problem with graphical applications, such as games, is to render huge amounts of polygons with various post-processing effects efficiently. Luckily, this has not been a problem in this case, as the amount of polygons and GPU processing in general was negligible in comparison to current GPU capabilities. Therefore, not only the GPU computation was insignificant, but also the passing the data from CPU to GPU, which is the usual bottleneck in other cases, was very fast. Thanks to this, the application is able to send the entire mesh with all its vertices and triangles to the GPU in every frame and render it in less than 5 ms.

4.2.1 Remeshing Process

The biggest problem was optimizing the remeshing process, which takes place on the CPU. Even with a data structure and algorithms designed to make the process as efficient as possible, it suffers from limitations in terms of triangle count.

Every operation in the remesh process has at most linear complexity, except for collapse and split phases, which include edge sorting. Therefore, the theoretical asymptotic complexity of the whole remeshing process is $O(n \log n)$. Since the sorting is a crucial part of the process, the asymptotic complexity cannot be further improved.

However, in practical terms, the most expensive operation happens to be the consolidation process. Even though its complexity is linear, it is the biggest bottleneck of the remeshing process. This is mainly due to the quantity and size of memory allocations that goes on in the consolidation.

There are several opportunities to optimizing the consolidation process. One of them is to consolidate not every frame, but only once in a time. This however causes lags that would hinder the sculpting process. A partial consolidation can also be performed, specifically on the edges, as the number of edges is what slows down the split/collapse operations the most when the mesh data structure is unconsolidated. However, to ensure smooth user experience, the remeshing process would have to be offloaded to a separate thread. That was however outside of the scope of this thesis, but it is an interesting avenue for future work.

4.2.2 Code Optimizations

As the asymptotic complexity cannot be further improved, I focused on more practical and low level optimizations. To find problematic parts of the code, a CPU profiling tool *Very Sleepy* [16] was used. Several pieces of code were found and optimized, such as the following:

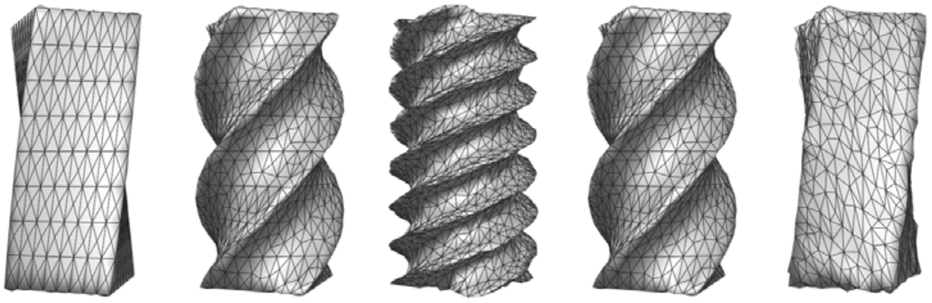


Figure 4.1: Box twist test: a 3D box is gradually twisted (which increases mesh density) to a certain point and then un-twisted (which decreases it).

- Comparing edge length as distance squared, to avoid costly square root function.
- Reserving enough space in arrays in advance, to avoid frequent reallocating.
- In splitting and collapsing process, precomputing edge lengths and them instead of actual edges, to avoid recomputation.

4.3 Performance Measurements

The most complex process is remeshing, thus the only relevant data to measure is the remeshing time. The operations like sculpting or rendering both run under 5 ms. To measure the time, two tests were devised, a box twist test and continuous sculpting test.

A machine running Windows 8 64bit with Intel i5-3570K @ 3.4 GHz CPU was used to measure the data. The remeshing time isn't affected by the GPU speed, as the whole remeshing process happens on CPU.

4.3.1 Box Twist Test

This test consists of twisting a 3D box by a certain amount and then twisting it back. This causes rapid increase in number of triangles when twisting and rapid decrease when untwisting.

The figure 4.2 shows the measured data. The rate at which triangles were added or removed was very steep, causing it to barely keep above 60 FPS. Also note, that the untwisting is slower than the twisting. This is because edge collapse, which is a more expensive operation, is prevalent when untwisting. This is also clearly demonstrated by the second test.

4.3.2 Continuous Sculpting Test

The second test is closer to actual process of sculpting, when we either add or decrease detail. To add detail, the pull tool was used to continuously pull the mesh and add triangles. The prevalent operation in this case is edge split. In the second case, when decreasing detail, the edge collapse was the dominant operation. This was achieved by slowly decreasing target edge length and remeshing at the same time.

As can be seen in figures 4.3 and 4.4, the relationship between number of triangles and remeshing time is linear. When comparing the remeshing times in the two graphs, it is clear that the edge collapse is more expensive operation than the edge split.

4.4 Examples of Sculpting Results

The figure 4.5 shows the example results of meshes sculpted in the application. All shown meshes are under 30 thousand triangles and were sculpted in about 10-20 minutes each. All tools were used to some extent, depending on the model shape.

The symmetry mode was enabled in all shown examples and they show that it helps a great deal when sculpting. If it wasn't used, the process of sculpting of these meshes would be tedious and the result wouldn't be as good. In case of the first example, the *head*, the symmetry mode was enabled to sculpt the shape of the head and the face. Then it was disabled to sculpt the hair.

When showcasing the application to new users, I observed that they were able to grasp the controls rather quickly and that they were able to sculpt basic shapes in a matter of minutes. Most of them were very excited after trying it out and provided valuable feedback. One of the sculpting results of a first time user is the second model, the *piglet*.

The third example, the *spider*, demonstrates the usefulness of the drag tool. It was used to extrude the legs from its body. The eyes and joints on the legs were made using the pull tool, which was also used to inflate the spider's abdomen.

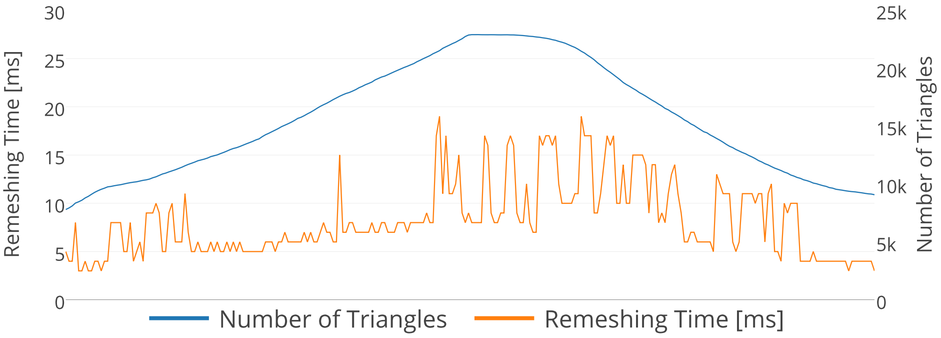


Figure 4.2: Box twist test: remeshing time

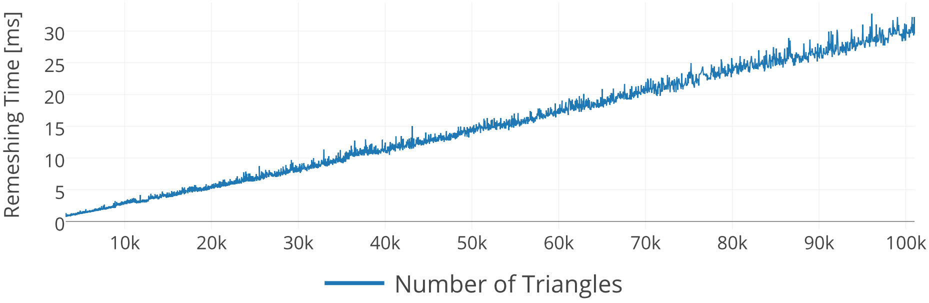


Figure 4.3: Continuous sculpting test: adding detail.

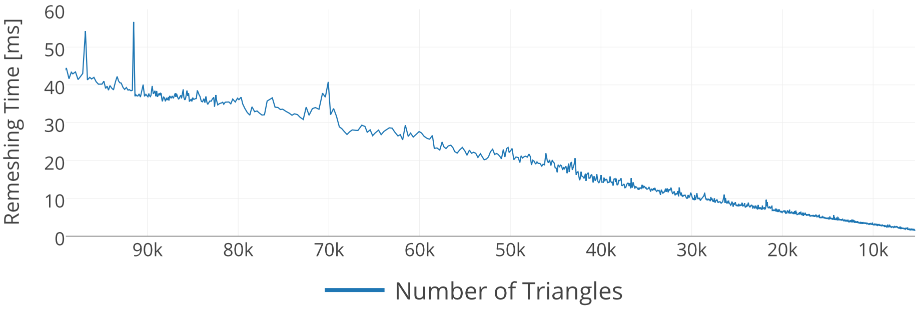


Figure 4.4: Continuous sculpting test: removing detail.

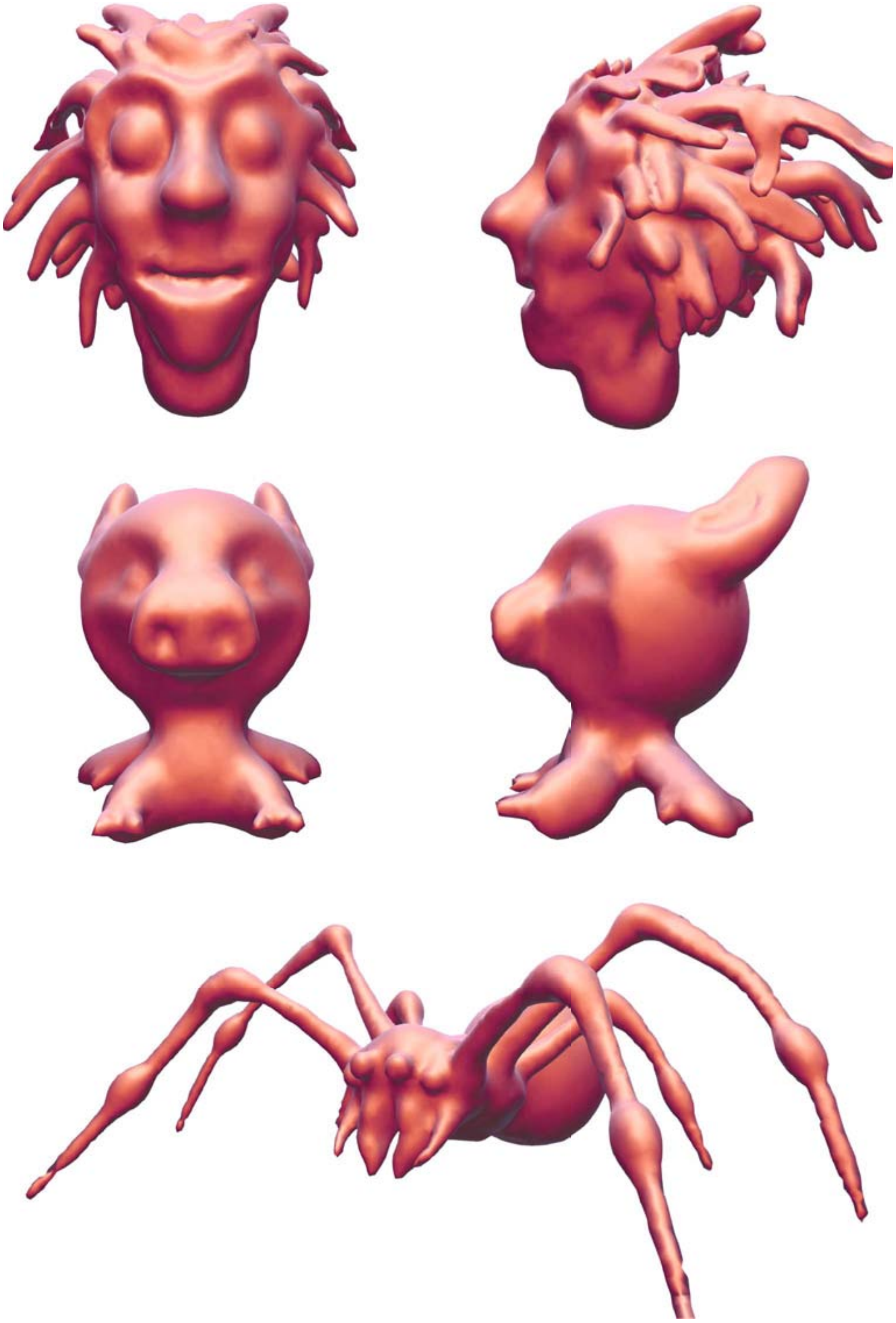


Figure 4.5: Examples of sculpting results: *head* (top), *piglet* (middle), *spider* (bottom)

Conclusion

During the development of this application I familiarized myself with sculpting and algorithms related to sculpting, particularly the remeshing process. However, the scientific work on this topic is extensive and I've only scratched the surface. Nevertheless, I've managed to implement a functioning sculpting system with basic tools that can be used to sculpt meshes with up to 100 thousand triangles. The system is being displayed in a virtual reality headset, Oculus Rift, and controlled by a position and orientation tracking device, Razer Hydra. Thus, all the primary and secondary goals of this thesis were met.

Other than implementing the sculpting and remeshing mechanism, the integration of the Razer Hydra and the Oculus Rift proved to be a challenge. To make them work together and turn it into a smooth virtual reality experience required a great deal of experimenting. Particularly, developing sculpting tools that are controlled by a position tracking device was no easy task. The fact that no previous work has been published on using this combination of peripherals to make a sculpting application made it even more difficult, but at the same time, more exciting.

The resulting application provides a simple and easy process of modeling, but what sets it apart is the real sense of scale and depth it provides, which is something you can't get in a commonly available desktop 3D modeling software. Turning something like sculpting, which is inherently physical and tangible, into an enhanced virtual reality is in my opinion remarkable and we should continue to explore the possibilities and the new doors that it can open.

Future work

The sculpting system that was implemented is still very basic and lacking some features of professional sculpting software, such as textured brushes, masking, custom brush curves, various deformations like skewing or bending and many more. In future, I would like to implement some of these features.

However, to implement more features, a better user interface has to be devised. Adding a graphical user interface (GUI) into virtual reality is no easy task, as traditional 2D GUI suffers from serious drawbacks when displayed in a virtual reality headset [15], for instance, elements placed near the borders of the screen won't be fully visible to the user. The most intuitive way to implement it is to put the interface elements in the 3D space and treating them like physical objects, which would require a great deal of further effort.

One feature that is very important for performance and mesh quality is non-uniform subdivision, i.e., the mesh has greater density only where the details are, while large flat surfaces have low density. This is something that could also be implemented, but it would require redesigning the remeshing process. Offloading the remeshing process to a separate thread could also prove to be beneficial to the performance.

The virtual reality hardware is progressing rapidly. The most important feature that is coming in the near future is the positional tracking of the Oculus Rift headset. This is something very important for virtual reality immersion and for reducing motion sickness. The practical consequence for this sculpting application would be the ability to look around corners and to easily see the result from different angles with just the movement of the head.

Bibliography

[1] Moraes, C. Sculpting process image. 2013, [Cited 2014-05-09]. Available from: http://www.ciceromoraes.com.br/?p=970

[2] Ebert, D. S.; Musgrav, K. F.; Peachey, D.; et al. *Texturing & modeling: a procedural approach*. Morgan Kaufmann, 2003.

[3] Thürrner, G.; Wüthrich, C. A. Computing vertex normals from polygonal facets. *Journal of Graphics Tools*, volume 3, no. 1, 1998: pp. 43–46.

[4] Stănculescu, L.; Chaine, R.; Cani, M.-P. SMI 2011: Full Paper: Freestyle: Sculpting Meshes with Self-adaptive Topology. *Comput. Graph.*, volume 35, no. 3, June 2011: pp. 614–622.

[5] Brochu, T.; Bridson, R. Robust topological operations for dynamic explicit surfaces. *SIAM Journal on Scientific Computing*, volume 31, no. 4, 2009: pp. 2472–2493.

[6] Dunyach, M.; Vanderhaeghe, D.; Barthe, L.; et al. Adaptive Remeshing for Real-Time Mesh Deformation. Eurographics Short Papers, 2013, pp. 29–32.

[7] Dyn, N.; Levine, D.; Gregory, J. A. A butterfly subdivision scheme for surface interpolation with tension control. *ACM Transactions on Graphics (TOG)*, volume 9, no. 2, 1990: pp. 160–169.

[8] Stabinger, S. Oculus Rift photos. Dec. 2013, [Cited 2014-05-09]. Available from: http://en.wikipedia.org/wiki/Oculus_Rift

[9] Antonov, M.; Mitchell, N.; Reisse, A.; et al. *Oculus SDK overview*. Oculus VR, Inc., July 2013.

[10] Abrash, M. Latency the sine qua non of AR and VR. Dec. 2012, [Cited 2014-04-30]. Available from: http://blogs.valvesoftware.com/abrash/latency-the-sine-qua-non-of-ar-and-vr/

[11] Razer. Razer Hydra photo. 2011, [Cited 2014-05-09]. Available from: http://www.razerzone.com/gb-en/gaming-controllers/razer-hydra-portal-2-bundle/

[12] Sixense. *Sixense SDK Overview*. Sixense Entertainment, Inc., 2011.

[13] Erdőkövy, Z. The Razer Hydra 3D model. Jan. 2012, [Cited 2014-05-09]. Available from: http://www.zspline.net/blog/2012/01/15/the-razer-hydra-3d-model/

[14] Shoemake, K. ARCBALL: A User Interface for Specifying Three-dimensional Orientation Using a Mouse. In *Proceedings of the Conference on Graphics Interface '92*, 1992, pp. 151–156.

[15] Yao, R.; Heath, T.; Davies, A.; et al. *Oculus VR Best Practices Guide*. Oculus VR, Inc., 2014.

[16] Chapman, N.; Mitton, R.; Engelbrecht, D.; et al. Very Sleepy CPU Profiler. 2012, [Cited 2014-05-12]. Available from: http://www.codersnotes.com/sleepy

Appendix A

Acronyms

VR Virtual Reality

HMD Head Mounted Display

FOV Field of View

DOF Degrees of Freedom

GPU Graphics Processing Unit

POI Point of Intersection

SDK Software Development Kit

IPD Interpupillary Distance

GUI Graphical User Interface

Appendix B

Contents of enclosed CD

- readme.txt ... the file with CD contents description
- exe ... the directory with executables
- src ... the directory of source codes
  - thesis ... the directory of LaTeX source codes of the thesis
  - RiftSculpt ... the directory of application source codes
- BP_Krs_Vojtech_2014.pdf ... the thesis text in PDF format