Hand-Tracking Based Interactive 3D Modeling using Mesh Deformation

Sofya Ogunseitan
Stanford University
EE267W Final Write Up
sneechie@stanford.edu

Abstract

The central goal of 3D modeling is to design objects in a digital 3-dimensional space. Industry Standard 3D modeling applications such as Maya, Blender, ZBrush, and 3Ds Max are applicable to a wide range of use cases from sculpting to animation, but also require understanding of a complex user interface which is often cumbersome for beginners. This steep learning curve also requires users to sculpt the computer mouse and keyboard, as opposed to traditional sculpting tools. The goal of this project is to create an interactive virtual reality 3D modeling tool that provides the user with an intuitive user interface and hands-on approach to 3D modeling. The project consists of two main components: a Leap Motion API to incorporate interactivity, and mesh deformation in Unity. By combining these tools, we successfully created a hands-on sculpting interface easily accessible to beginners.

1. Introduction

The purpose of 3D-modeling is to simulate real-life objects in digital space. The industry standard software applications for 3D modeling are applicable to a variety of fields such as visual effects (VFX) and animation, health care, or game creation and film-making. Software packages such as ZBrush, Maya, Blender and 3Ds Max are efficient, flexible, and powerful frameworks applicable to a wide array of industrial 3D modeling needs. However, all of these applications pose a steep learning curve that is inaccessible to beginners or amateur users.

This project proposes a hands-on virtual reality (VR) modeling interface using mesh deformation in Unity and an integrated Leap Motion API Hand-Tracking format like Leap Motion in Unity, this project is a rudimentary application focused on editing meshes using vertex manipulation triggered by converted screen-space coordinates of the API. The project will consist of several hand gestures used to navigate the view, position, rotation, and scale of the object. Furthermore, we will need to implement a specific algorithm for mesh deformation using the vertices of the given mesh.

In what follows, we present a prototype of this application that integrates a mesh deformation algorithm with Leap Motion's API in Unity. In this paper, we will talk in depth about the technical background of mesh deformation in Section 3, and further how we integrated it with the API to create the main section of our prototype's functionality. We will also discuss the strengths and weaknesses of our prototype, future additions and improvements, and the challenges we faced in implementing the functionality.

2. Background

There have been many previous approaches to implementing a more interactive software package for 3D modeling, ranging from sketch-based modeling to touch-based modeling, many of which have unique configurations of mesh deformation and sculpting.

2.1. Mesh Deformation Summary

The goal of mesh deformation is for the user to give as little input as possible and have the deformation algorithm deduce and create the expected result. A shape, or default mesh is deformed by some algorithm based on physical constraints determined by the user. Two commonplace approaches to deformation include surface deformation and space deformation. Surface representations are typically handled by Laplacian coordinates, which incorporate vertices moved by the user and a region of influence around those vertices.

$$V' = argmin \sum_{i=1}^{n} ||\delta_i - L(v_i')||^2 + \sum_{i \in C} ||v_i' - u_i||^2$$

$$\delta_i = L(v_i + t)$$

In this equation, the deformed vertex is computed by the

laplacian coordinates of the original mesh and deformed mesh, added with the users constraints.

2.2. Related Works

In a paper titled "Shape Preserving Mesh Deformation" by Alla Sheffer and Vladislav Krayevoy, researchers implement mesh deformation by describing each vertex with respect to its neighboring vertices, rather than using the global scale of the entire mesh [1]. The three stages of this implementation include representation of the projection plane that contains projections of the current mesh vertex and three of it's neighbors. Using the projected set of angles between each neighbor, they calculate the edge lengths between the current mesh vertex and each neighbor.Next is reconstruction, which calculates the actual position of the mesh vertex using Barycentric weights and finally deformation. The input to computing the deformation is simply a set of control points that define the type of deformation - and the possibilities range from rotation, scaling, and bending. This technique is particularly useful for animation because of it's specificity to a particular section of the mesh.

At SIGGRAPH in 2014, Alec Jacobson, Zhigang Deng, Ladislav Kavan and J.P. Lewis developed a course on Real Time Shape Deformation, mainly focused on the techniques used for character modeling [4] [5]. Some of the concepts, however, are applicable to a broader understanding of the mesh deformation progress that goes from the rest or default position of a mesh to a deformed one. The section taught by Ladislav Kavan specifically talks about Skinning Methods and Deformation Primitives. Kavan presents skinning as a method to control the deformations of a given object using a set of deformation primitives, or the basic skeleton of a particular mesh, usually a virtual character. Although this implementation is beyond the scope of this project, it is definitely something to consider outside the time constraints of this project in order to address future deformation nuances.

The first project we looked at by Takeo Igarashi, Satoshi Matsuoka, and Hidehiko Tanaka at the University of Tokyo and the Tokyo Institute of Technology, implemented a sketch-based model of 3D free-form sculpting [3]. Their implementation takes a 2D drawing and generates a suitable arrangement of polygons to add a third dimension to the artist's creation. The algorithm of mesh development begins with user input as a 2D stroke and generates polygons based on the start and stop positions of the stroke. This implementation, however, is prone to illegal strokes that intersect themselves and other issues that would naturally arise from converting a free-form sketch to free-form digital sculpting.

Similarly, at the University of Manchester, Xin Bao and Toby L.J. Howard implemented SEED, Sketched Based 3D Model Deformation [2]. The implementation was focused on using a single control stroke to perform a deformation rather than a group of control points or a ROI (Region of Interest).

One unique method for approaching mesh deformation we found called "Mesh Puppetry" by researchers at CalTech and Microsoft presents a detail-preserving mesh manipulation technique [6]. This technique supports direct manipulation of vertices in the mesh and uses Inverse Kinematic-based deformation constraints, in addition to self-intersection avoidance.

3. Methods for Mesh Deformation

We used Unity's physics applications to implement a suitable version of mesh deformation. There are five essential parts to doing this: Tracking original and deformed vertices, responding to user input, implementing attenuated velocities for all effected vertices, debugging, and integrating mesh deformation with API.

3.1. Keeping Track Mesh Vertices

The first step in creating a suitable implementation of mesh deformation is to have a consistent method to keep track of both the vertices of the original mesh and the vertices that will be deformed. We did this by keeping separate arrays of vectors. There is an array for the original vertices, deformed vertices, and the velocities of the all vertices. Unity gives us a method to extract the array of original vertices: mesh.vertices().

3.2. User Input

We started our implementation by using simple mouse input included by Unity, as it provides solid 2D screen coordinates that would be easy to test the functionality of mesh deformation. For the final product, however, we needed to convert the Leap Motion's API input into 2D screen space coordinates. The integration of these two scripts included checking for user input, casting each input as a ray that points from the camera to scene with the meshes, and keeping track of the intersection of the ray and the surface of the mesh. We kept track of the intersections by using Unity's implementation of Ray-casting, which includes a function "Physics.Raycast" which takes in a ray as a parameter and returns the intersection point on the mesh. Using this, we were able to determine which vertices to apply a force to.

3.3. Attenuated Force and Velocities of each Vertex

Taking the user input, we calculate an attenuated force based on the position of the vertex being deformed and its distance from the actual point on the screen. We need to account for the fact that its not just one vertex being displaced. It is a group of vertices surrounding the spe-

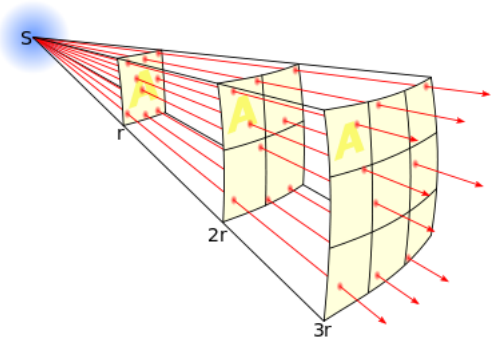

Figure 1. Inverse Square Law Diagram

Figure 2. Grabbing Vertices Hand Gesture

cific point referenced by the user. We calculate the attenuated force of each vertex effected by deformation using the Inverse-Square Law. This law states that a "physical quantity or intensity is inversely proportional to the square of the distance from the source of that physical quantity." This law is applied specifically when some "force or energy is evenly radiated outward from a point source in 3D space", which is why it is particularly helpful for our project implementation.

$$intensity \propto \frac{1}{distance^2}$$

Using logic based off this concept, we can compute the magnitude of the vector from the point to the current vertex on the mesh and set it inversely proportional to the force applied to it.

The next step in this implementation is to convert this attenuated force to the velocity of an individual vertex. This we calculate using Newton's Second Law of motion, first to calculate the acceleration and then to derive the velocity with the acceleration value multiplied by Unity's feature "Time.deltaTime"(Unity).

$$acceleration = Force/mass$$

We can pretend that the unit mass of each vertex is one to calculate the velocity:

$$\Delta velocity = acceleration * \Delta Time$$

Finally we can calculate the direction of the velocity by normalizing the screen space point first given to us by the user input.

3.4. Accounting for Cumulative Velocities

A challenge that we faced with mesh deformation was accounting for the fact that the velocity would continuously accumulate when any force was applied by user input. To compensate, we added spring and dampening forces. Since we previously stored the original vertices and the new displaced vertices in different vectors, we can compute the displacement by applying a spring force as the displaced vertices get farther away.

For the purpose of this project, however, we don't want the displaced vertices to return to the original mesh. We want the mesh to respond directly to the parameters of the user's input. In order to this, we used the uniform local scale of the current object being deformed and applied a very small scalar to it that prevents it from scaling upward indefinitely. We only apply this scalar to the uniform local scale in the computation of the displacement.

3.5. Combining Leap Motion with Mesh Deformation

We tested our mesh deformation implementation using mouse input, which is provided by Unity's Input functionality. For our final implementation, we had to switch from mouse coordinates to the coordinates of the Leap Motion.

The first and main step in making this transition was to compute the Leap Motion coordinates into screen space. We did this by first normalizing the leap motion vector, recentering the point, and finally using the screen width and height to compute the 2D coordinates.

Our final step in this process was adding a check to verify user input and using the Leap Motion screen coordinates to compute the ray casted from the camera to the scene.

The gesture used to call the function that handles the user input to trigger mesh deformation consists of the thumb and index finger closed together. The user can direct any point on the mesh to be deformed by moving this gesture throughout the scope of the Leap Motion's field of view.

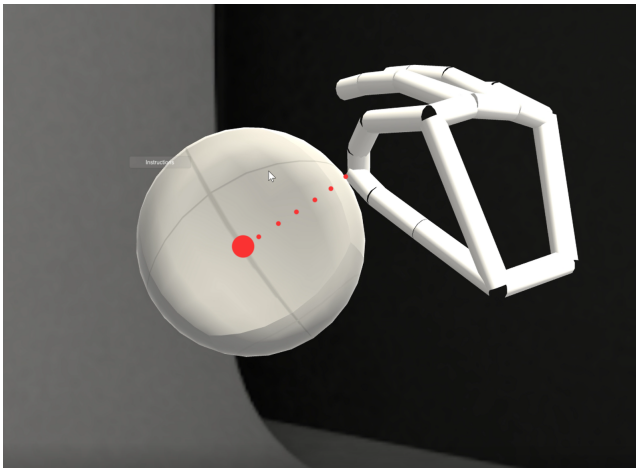

Figure 3. Grabbing Vertices Hand Gesture

4. Results

Our resulting application was a prototype of Interactive 3D modeling. We have constructed a plan for future improvements based on the feedback we received from demo day.

Our current prototype contains six different functions regarding the Leap Motion API and mesh deformation. Users can edit the given mesh using mesh deformation that is based on the displacement of groups of vertices on the mesh based on the position and motion of the user's right hand. Users can adjust the view of the mesh and it's rotation using the other hand gestures displayed in Figure 5.

This project loosely replicates the concept of sculpting in a physical space. The strengths of our implementation include a straightforward user manual and interface, and a suitable method for sculpting and deforming meshes with enough vertices.

Most of the feedback directed us towards potential improvements regarding the current hand gestures used to manage the view of the mesh and the responsiveness of the Leap Motion. It was difficult at times for the Leap Motion to recognize the hand gestures of the user based on their distance from the Leap Motion and the accuracy of the hand gestures. The mesh deformation visibility was also weakness in our implementation. Some of the users were unable to notice the displacement of the vertices upon gesturing for them to deform. Other weakness in our implementation that we will address in the future can be found in Section 6.

Ultimately however, our prototype allowed us to create distinct and abstract 3D models using the mesh deformation algorithm and the current hand motions. With some functionality additions and tweaking of the Leap Motion, we should be able to create a finished product that overall replicates the process of sculpting in physical space.

5. Discussion

One of the most difficult implementations of the project was integrating the Leap Motion API with the mesh deformation implementation. The issues were mainly with converting the Leap Motion API coordinates to screen space coordinates that would be suitable for mesh deformation. The direction of the vector that points from the camera to the scene was incorrect because of faulty normalization calculations.

Another difficult aspect of this project was choosing the hand gesture for each function. Ideally, the best hand gestures would give the user a natural intuition for editing both the view of the scene and the mesh being sculpted. Our prototype contains six different hand gestures for functions including zooming in and out, rotating the mesh, resetting the mesh to it's main initial position, and deforming a vertex. Our initial implementation exchanged the zooming in and out motions in Figure 4 for the current ones displayed in Figure 5.

Adjusting the Leap Motion to recognize user input was also challenge. The flexibility of it's recognition was minimal, and because the feedback of mesh deformation was not very visible, sculpting the object appeared to progress slowly, especially on meshes that had less vertices that the sphere. There were also limitations on the type of sculpting that could be done on the different meshes because the default shapes consist of a uniform arrangement of vertices. Many of the resulting meshes that we created using our program looked similarly abstract. In future improvements, we intend to diversify the type of sculpting tools that can be used on each mesh in addition to the option to subdivide meshes that come with a default low quantity number of vertices.

Finally, computing the correct uniform scalar for the mesh deformation was a challenge. Although this value could still use some adjusting, the scalar applied to each modification was based on the size of the dent or deformation made on the user input. We wanted to implement a version of mesh deformation that would satisfy user expectations regarding the length of time they hold the "grab vertex" position. In essence, the first step to calculating this was to keep the default mesh scales consistent (between spheres, cubes, capsules, etc) in the beginning. We then would apply a uniform scalar that would make the deformation on a separate, much smaller scale than the default scale. This makes the attenuated force appear significantly less cumulative, and stops the mesh from expanding indefinitely.

6. Future Work

Some of the weaknesses in our implementation were caused by the time constraint of two weeks, so there are

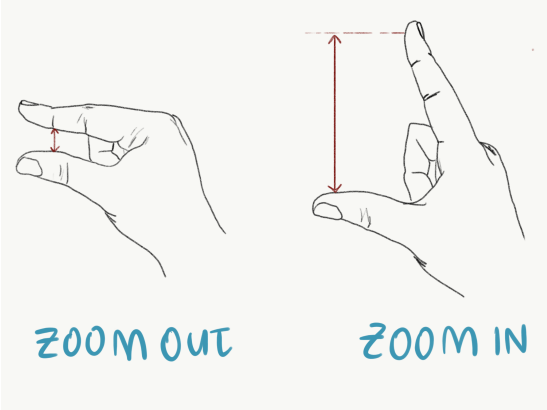

Figure 4. Original Zooming In and Out Gestures

some features that could improve our software application.

6.1. Hand Gestures

Creating more options for the user is one improvement we want to build in the future. Currently, we have six simple gestures and only one is used for mesh deformation. While the other five hand gestures can manipulate the view, rotation, and scale of the mesh, creating a more expansive list of tools for the user is important. Sculptors use a variety of tools to create art and the original purpose of our software application is to replicate the tactile process of crafting real life objects.

6.2. Functionality

The users who tested our prototype at the demo made suggestions that referenced the functionality of our implementation. In the future, we would like to incorporate some type of visual feedback for the user. Because our implementation of mesh deformation depends on user input, it does take a bit longer to display the results of the user's motion. To compensate, we could implement functionality that would allow the user to visually see which areas of the mesh are being deformed by their motion.

Furthermore, we would like to include a subdivide option for certain types of meshes. If we include various types of meshes that the user can start off with, we would implement an algorithm that subdivides the shapes so that it is more easily deformed. Our mesh deformation algorithm depends on the quantity of vertices on the current mesh, which ultimately means that default meshes with less vertices will be significantly more difficult to sculpt in a suitable way. For example, a default cube has eight vertices. With this small quantity of vertices, there is a very limited number of ways the artist would be able to sculpt the mesh, so we would need to subdivide it before beginning the program. The reason we used a cube for the prototype is because in 3D digital space they start off with 482 vertices.

6.3. User Interface

Our user interface currently consists of a panel for instructions. A new user interface would include some options to edit different types of meshes. For example, we could include options to begin with a cube, capsule, cone, or cylinder rather than just a sphere. We could include some of the improvements discussed above, such as options to switch the type of mesh deformation currently being used. In addition, we could include an option to clear the mesh and start over.

Another aspect of the user interface is the hand gestures used to trigger the functionality of our prototype. The current menu of hand gestures can be found in Figure 2. Based on initial feedback, we would like to incorporate a more psychologically based design of different hand gestures for different functions.

6.4. Types of Mesh Deformation

Currently, our model has one specific type of mesh deformation that is loosely based on applying velocity force to a specific group of vertices and maintaining that the deformed mesh. This concept makes the mesh appear to be displaced by the user's hand gestures. To add more variety, we could program more functions that would modify the given mesh. For example, we could include functionality that would allow users to extrude specific faces of the mesh or rotate vertices an edges accordingly. Although these tools would make the overall application less simplistic, they would enhance functionality. The accessibility or straightforwardness of the application can be compensated for with an understandable user interface.

6.5. Other Improvements

Additional features that we would want to implement include an option to save and export deformed models. This feature is less relevant to the scope of this class and would only be an additional feature to our implementation for the purpose of polishing the product as an actual software package.

7. Conclusion

Overall, the prototype we built does create a virtual world that allows users to physically interact with and edit a given mesh. The original product we wanted to create requires a keen attention to the psychological, artistic, and technical aspects of 3D modeling software packages, and more so because our prototype is interactively involving the user through Hand Tracking. Considering all these different topics, we were able to create an application that has in-

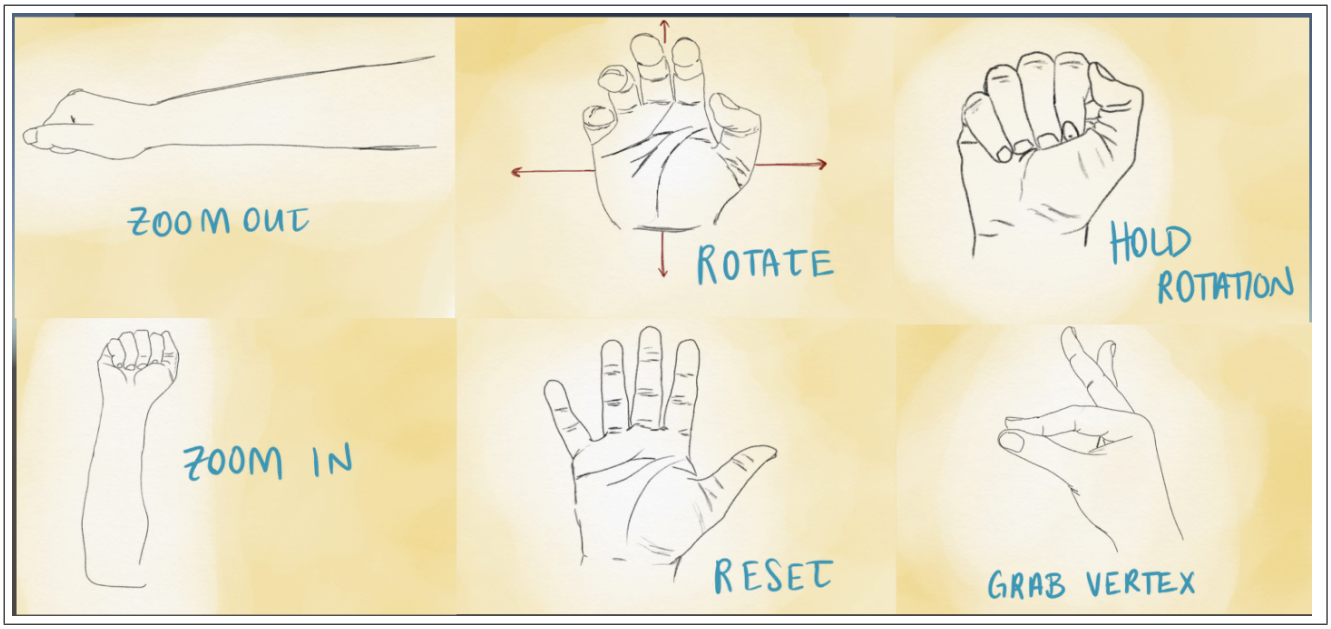

Figure 5. Example of a short caption, which should be centered.

tuitive instructions that straightforwardly explain the functionality. Like all skills, however, 3D modeling with any software package requires practice and an overall understanding of how each motion can impact the mesh directly. We made this process more approachable with our prototype, and with future additions and improvements, this application will be able to accurately involve the user without the expense of artistic detail.

References

[1] V. K. Alla Sheffer. Shape preserving mesh deformation. *University of British Columbia*, 1(1):1, 2004.

[2] X. Bao and T. L. J. Howard. Seed: Sketch-based 3d model deformation. *School of Computer Science, The University of Manchester*, 1(1):1 – 12, 2014.

[3] T. Igarashi and H. T. Satoshi Matsuoka. Teddy: A sketching interface for 3d freeform design. *University of Tokyo, Tokyo Institute of Technology*, 1(1):1–8, 2000.

[4] A. Jacobson, Z. Deng, L. Kavan, and J. Lewis. Skinning: Real-time shape deformation. 2014.

[5] L. Kavan. Direct skinning methods and deformation primitives. *University of Pennsylvania, SIGGRAPH Course 2014 Skinning: Real-time Shape Deformation*, 1(1):1 – 11, 2014.

[6] Y. T. M. D. H. B. B. G. Xiaohan Shi, Kun Zhou. Mesh puppetry: Cascading optimization of mesh deformation with inverse kinematics. *State Key Lab of CADCG, Zhejiang University, Caltech*, 1(1):1 – 9, 2007.