Computational Fabrication

Volumetric Modelling Using Holographic Projection

Final Report

Jermaine Cheng
1000502

Ng Ping
1000513

Teh Jun Hao
1000464

Content

1. Project Introduction

Figure 1.1: Project Idea Depiction And Conceptualization

The motivation behind our project is to reinvent the way people do Computer-Aided Design (CAD). Traditionally, if someone wants to CAD a model they have to learn how to use a CAD software which may be intimidating to beginners. Our project objective is to create a CAD software which uses a hologram to visualise the designed model while allowing the designer to interact with and sculpt the model using hand gestures made possible by utilising the Leap Motion Controller.

2. Software Overview

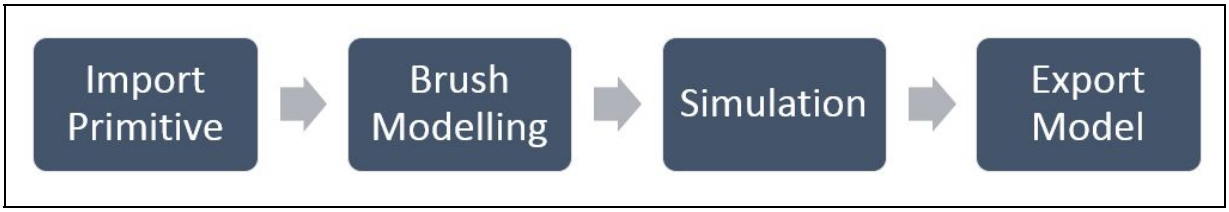

Figure 2.1: CAD Modelling Workflow

As seen in Figure 2.1, the function of our software is to allow the user to create their own CAD design as well as run simulations on their CAD design and eventually be able to export their CAD design for 3D printing. Our software is designed according to the CAD modelling workflow shown above. When the user first launches the software he/she would first select a primitive model as a starting point. Next, the user will employ brush modelling where hand gestures are used to deform, scale and rotate the model which is projected on our hologram set up. In the simulation stage the user can subject their designed model to fracture and deformation simulation which are also displayed using the hologram set up. Finally the user would be able to export their completed design as an .obj file for 3D printing.

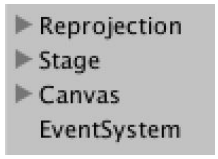

Figure 2.2: Breakdown hierachy on how the application was planned

The project comprises of 3 main categories, the canvas holds all UI and their respective functions, the stage and reprojection sets up our scene for the prism projection and leap motion control seen in Figure 2.2.

Figure 2.3: Animated splash screen with transition to the main hologram module

For our project, to ensure that we implement polished user experience. We incorporated the use of motion tweens and sequences typically used in programmatically generated animation. This allows us to control subtle movements in our user interface and unity objects. Due to the nature of the prototype, we only have one screen in our program state machine active at the moment, as the project grows this module will become a program user interface controller handling all events occurring in the unity scene. A code sample for tweening can be seen in Figure 2.4.

```
using UnityEngine;
using System.Collections;
using DG.Tweening;

public class StartScreenController : MonoBehaviour {

    //Initializing required game objects
    public GameObject ball;
    public GameObject loadingBar;
    public GameObject startScreen;
    public GameObject[] modelingScreen;
    public GameObject[] visualPlanes;

    //Routine to create the bouncing animation
    void bounceUp (){ball.transform.DOLocalMoveY (0, 0.5f).SetEase(Ease.OutQuint).OnComplete(bounceDown);}
    void bounceDown (){ball.transform.DOLocalMoveY (-20, 0.5f).SetEase(Ease.InQuint).OnComplete(bounceUp);}
    void delete() {Destroy(startScreen);}
    void outro(){
        startScreen.transform.DOScale (new Vector3 (0, 0, 0), 1f).SetEase (Ease.OutExpo).OnComplete(delete);
        for (int a = 0; a < modelingScreen.Length; a++) {
            modelingScreen [a].transform.DOScale (new Vector3 (0.2f, 0.2f, 0.2f), 0.5f).SetEase (Ease.OutExpo).SetDelay(0.125f);
        }
        for (int b = 0; b < visualPlanes.Length ; b++) {
            visualPlanes [b].transform.DOScale (new Vector3 (0.5f, 0.5f, 0.5f), 0.5f).SetEase (Ease.OutExpo);
        }
    }
    // Use this for initialization
    void Start (){
        bounceUp ();
    }

    // Update is called once per frame
    void Update () {
        loadingBar.transform.DOScaleX (1.5f, 3f).SetEase (Ease.Linear).OnComplete (outro);
    }
}
```

Figure 2.4: Code snippet for animation sequence with the splash screen

3. Set Number of Hologram Projections

To see the hologram, a n-sided prism would need to be placed in the centre of the screen where the application can dynamically load the viewer for the respective prism as seen in Figure 3.1.

In this piece of code, the reprojection planes and cameras are dynamically set on runtime in coordinations of a circle of set radius. It always ensures that the first plane for projection starts on the top to maximize the screen space to prism useage. It also has an inbuilt input checker just to make sure that the input details for the UI corresponds to an understandable value before parsing the reprojection. Otherwise, an error message would be produced.

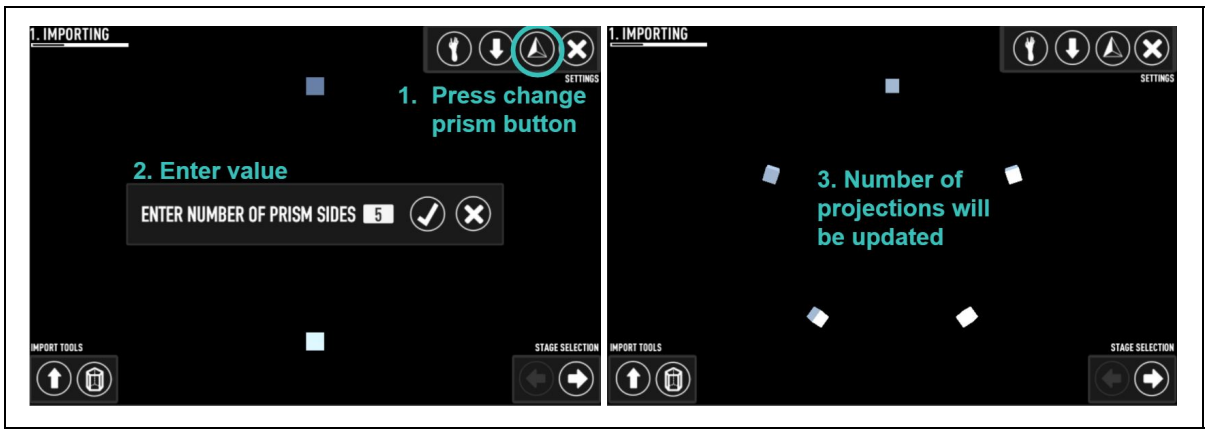

Figure 3.1: Steps to Change Number of Projections

```
public void checkInput() {

    try{
        int sides = Int32.Parse (inputText.GetComponent<Text>().text);

        //Making sure allowing input as mentioned in warning text is enforced
        if ((sides >= 1) && (sides <=9)) {

            canvas.outroPrismScreen();

            //Rubbish collection for scene items
            for (int a = 8; a < planesParent.transform.childCount; a++) {
            if (a == planesParent.transform.childCount - 1) {
                planesParent.transform.GetChild (a).transform.DOScale (new Vector3 (0f, 0f, 0f), 0.25f).OnComplete (clearScene);
            } else {
                planesParent.transform.GetChild (a).transform.DOScale (new Vector3 (0f, 0f, 0f), 0.25f);
            }
            }

            //Coroutines allow us to use delays
            StartCoroutine (generatorSides(sides));

        } else {
            warningAnimation ();
        }
    }
    catch{
        warningAnimation ();
    }
}
```

```
float xCam = cRadius * Mathf.Sin(degStart);
float zCam = cRadius * Mathf.Cos(degStart);
float xPlane = pRadius * Mathf.Sin(degStart);
float zPlane = pRadius * Mathf.Cos(degStart);
degStart += degSep;

cRender[a] = new RenderTexture(512, 512, 24, RenderTextureFormat.ARGB32);

cMats [a] = new Material (Shader.Find("Mobile/Diffuse"));
cMats [a].mainTexture = cRender [a];

cCams[a] = new GameObject("Camera " + a);
cCams[a].transform.parent = cameraParent.transform;
cCams [a].tag = "SubCamera";
cCams [a].layer = 8;
cCams [a].AddComponent<Camera> ();
cCams [a].GetComponent<Camera> ().backgroundColor = new Color(0f,0f,0f,0f);
cCams [a].GetComponent<Camera> ().clearFlags = CameraClearFlags.SolidColor;
cCams [a].GetComponent<Camera> ().targetTexture = cRender[a];
cCams [a].transform.localPosition = new Vector3 (xCam, 0f, zCam);
cCams [a].transform.LookAt(new Vector3 (0f, 0f, 0f));
cCams [a].GetComponent<Camera>().cullingMask = ~(1<<9);


cPlanes[a] = GameObject.CreatePrimitive(PrimitiveType.Plane);
cPlanes[a].name = "Plane " + a;
cPlanes [a].GetComponent<MeshRenderer> ().material = cMats [a];
cPlanes [a].transform.parent = planesParent.transform;
cPlanes [a].transform.localPosition = new Vector3 (xPlane, 0f,zPlane);
cPlanes [a].transform.localScale = new Vector3 (0f, 0f, 0f);
cPlanes [a].transform.DOScale (new Vector3 (-1*0.5f*(4f/sides), 0.5f*(4f/sides), 0.5f*(4f/sides)), 0.25f);;
cPlanes [a].transform.LookAt(new Vector3 (0f, 0f, 0f));
cPlanes [a].layer = 9;
```

Figure 3.2: Code snippet for animation sequence and reprojection functionality

4. Fabricating N-Sided Prism

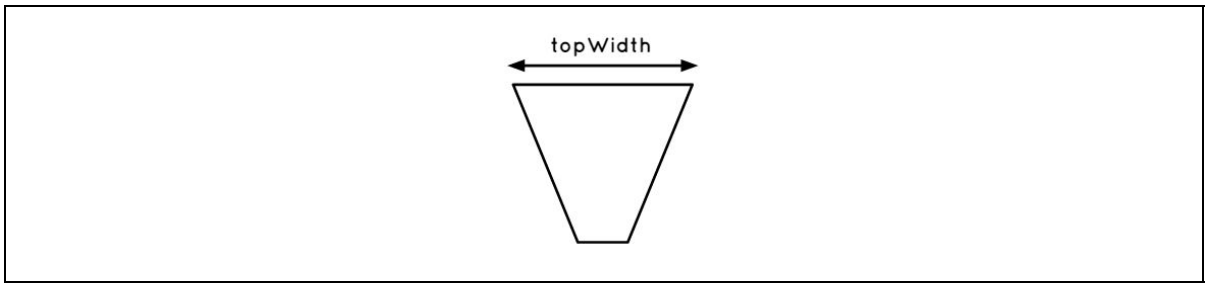

Figure 4.1: Prism face

In order to display n number of hologram projections we would need an n-sided prism with a prism face as seen in Figure 4.1. To make the process of fabricating an n-sided prism convenient, we created a Matlab function which takes two input parameters as seen in Figure 4.2. The Matlab function outputs the SVG file containing the generated prism faces for lasercut.

printPrism(numSides, topWidth)
Function inputs:
numSides - number of faces the prism will have
topWidth - top width of a prism face in mm

Figure 4.2: Code reference for making a n sided prism in Matlab

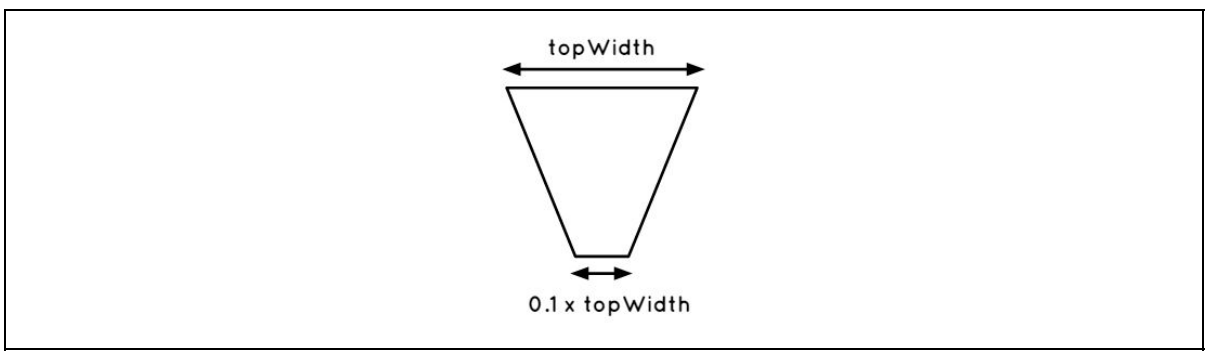

Figure 4.3: Constraint widths on Prism Face

As seen in Figure 4.3, our first constraint relies on the base width of each face being 10% of top width. For each prism face, the smaller the base width the larger the projectable area. However if the base width is too small, the prism will topple easily. Through experimentation we determined that if base width is 10% of top width it achieves a good balance of large projectable area and stability.

The second constraint ensures the plane of each face will intersect with the plane of table surface at 45 degrees as seen in Figure 4.4. Since at exactly 45 degrees, the projected image on each prism face has no distortion and thus provides the best viewing experience.

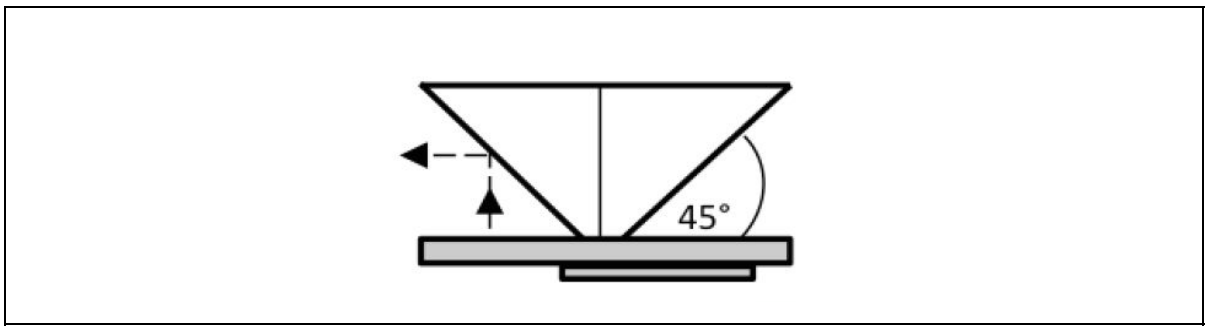

Figure 4.4: 45 Degrees Constraint

Lastly, to get the dimensions of the prism face, we take into account the above constraints to perform geometric calculations to derive the following relationship for h = 0.45*topWidth*tand(180 * (numSides - 2) / numSides) / √2 as seen in Figure 4.5.

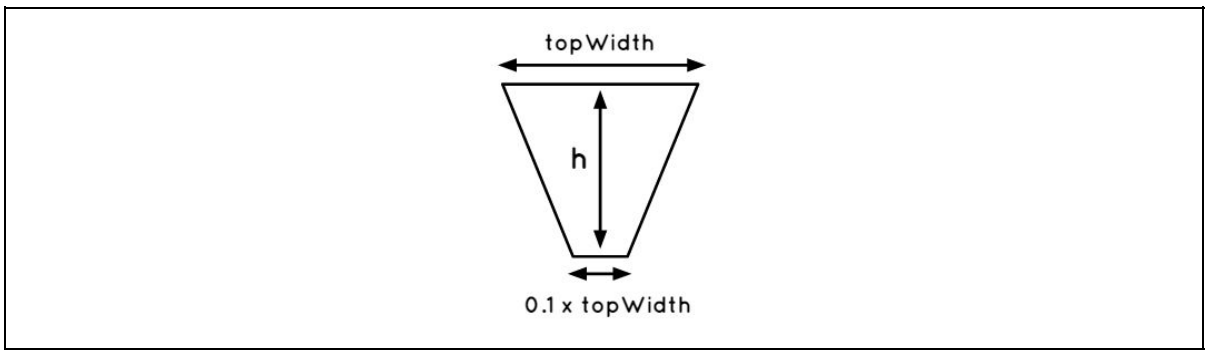

Figure 4.5: Dimensions of Prism Face

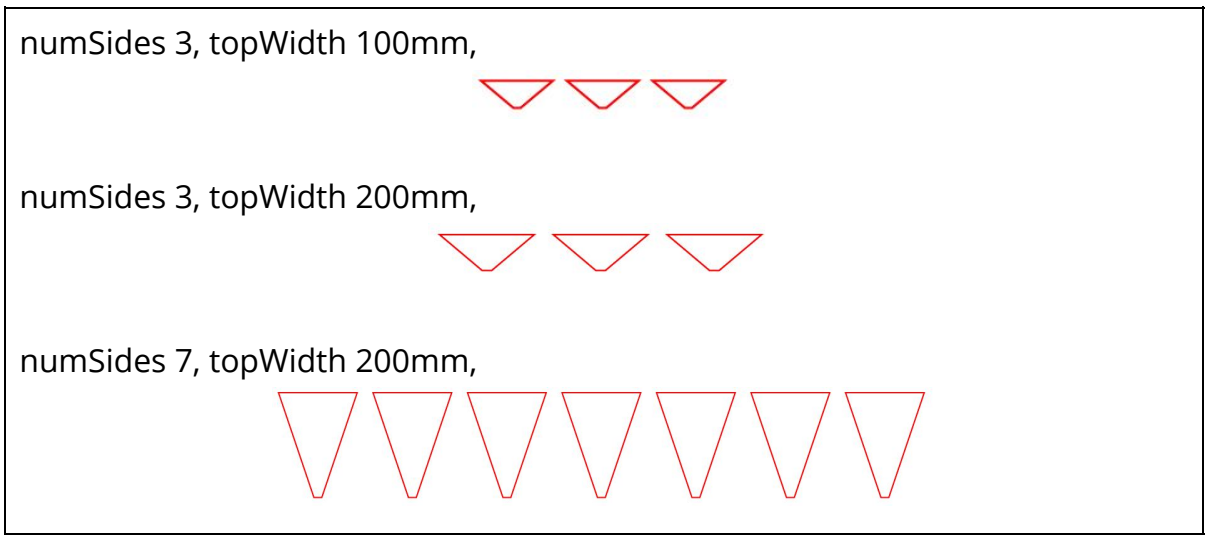

Figure 4.6: Sample prism SVG outputs generated by printPrism()

5. Primitive Selection

Option 1: Import .obj File as Primitive

To import as obj we create a parser to understand the .obj file and breakdown the data into chunks understandable by unity as seen in Figure 5.1.1. In this functionality, the input checker is also used to ensure that any error messages will be displayed to the users and for simplicity all obj files have to be placed on the desktop as seen in Figure 5.1.2 and 5.1.3.

Figure 5.1.1: How to Import .obj Primitive

```
using UnityExtension;
using UnityEngine;
using UnityEngine.UI;

using System;
using System.IO;
using System.Collections;
using System.Collections.Generic;

public class ModelingController : MonoBehaviour {

    public GameObject item;

    // On Click Events for sphere mesh
    public void exportObject() {
        var lStream = new FileStream(System.Environment.GetFolderPath(System.Environment.SpecialFolder.Desktop)+"/cube.obj", FileMode.Create);
        var lOBJData = item.GetComponent<MeshFilter>().mesh.EncodeOBJ();
        OBJLoader.ExportOBJ(lOBJData, lStream);
        lStream.Close();
    }

    public void importObject() {
        var lStream = new FileStream(System.Environment.GetFolderPath(System.Environment.SpecialFolder.Desktop)+"/bunny.obj", FileMode.Open);
        var lOBJData = OBJLoader.LoadOBJ(lStream);
        item.GetComponent<MeshFilter>().mesh.LoadOBJ(lOBJData);
        lStream.Close();
    }
```

Figure 5.1.2: Code snippet for the exporting and importing of files using the file parser controller

```
public static void LoadOBJ(this Mesh lMesh, OBJData lData)
{
    List<Vector3> lVertices = new List<Vector3>();
    List<Vector3> lNormals = new List<Vector3>();
    List<Vector2> lUVs = new List<Vector2>();
    List<int>[] lIndices = new List<int>[lData.m_Groups.Count];
    Dictionary<OBJFaceVertex, int> lVertexIndexRemap = new Dictionary<OBJFaceVertex, int>();
    bool lHasNormals = lData.m_Normals.Count > 0;
    bool lHasUVs = lData.m_UVs.Count > 0;
```

Figure 5.1.3: Code snippet for data components of the file parsing data controller

Option 2: Choose from a List of Default Primitives

To do mesh swapping we have to create a primitive helper function to assist with memory since upon creation and assigning the mesh to the mesh filter, we would want to destroy the created primitive to free space as seen in Figure 5.2.2.

Press the Change Primitive Model button multiple times to toggle between the various default primitives as seen in Figure 5.2.1. There are four default primitives available: Cube, Cylinder, Sphere & Capsule. The user can rotate the object using his or her right hand by pinching index finger and thumb together then moving the hand around while maintaining the pinch.

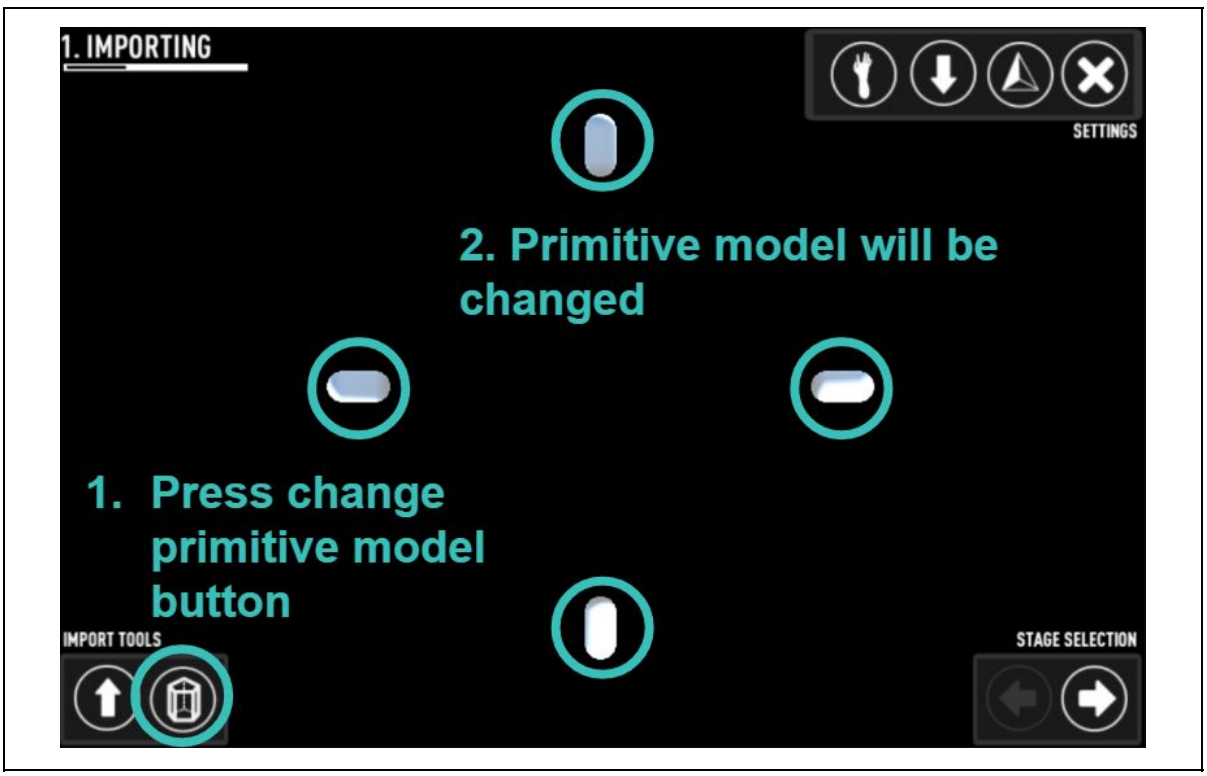

Figure 5.2.1: Steps to Select Default Primitive

```
public void primitiveSwapper(){
    curPrim += 1;
    curPrim = curPrim % 4;
    item.GetComponent<MeshFilter> ().mesh = PrimitiveHelper.GetPrimitiveMesh (primDict[curPrim]);
    item.GetComponent<MeshCollider> ().sharedMesh = item.GetComponent<MeshFilter> ().mesh;
}
```

Figure 5.2.2: Code snippet for primitive swapper controller

6. Brush Modelling

a. Modeling State Overview

The next stage involves modelling primitive as seen in Figure 6.1.1. At the modelling stage three new buttons appear - Brush, Increase Vertices Count & Scale. Here the object can be rotated by pinching the right hand too as seen in Figure 6.1.2.

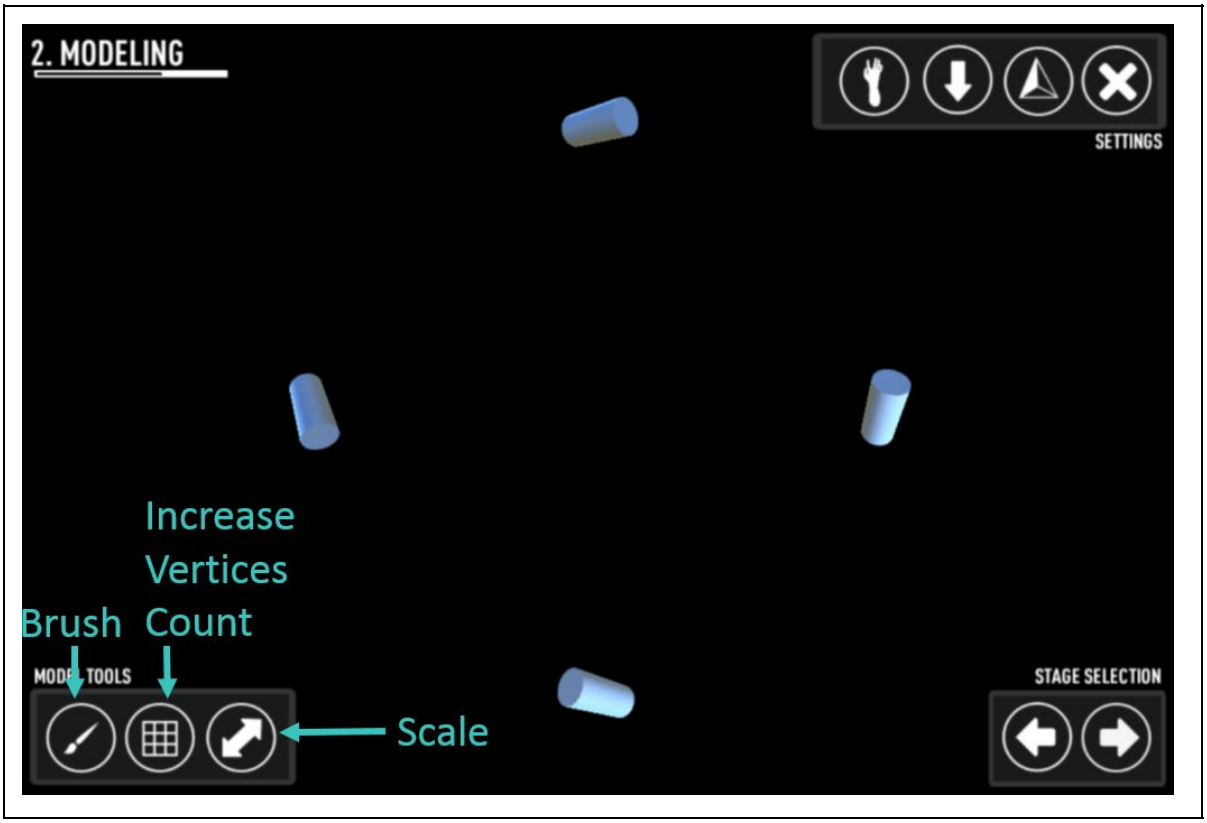

Figure 6.1.1: Modelling UI

```
using UnityEngine;
using UnityEngine.UI;
using System.Collections;
using Leap;

public class GestureController : MonoBehaviour {

    //Retrieves the handcontroller which controls all behaviours
    public HandController hc;
    public GameObject item;

    //Important information
    public Vector3 leftHandPosition;
    public Vector3 leftHandNorm;
    public Vector3 rightHandPosition;
    public Vector3 rightHandNorm;
    public float swipeUp;
    public float swipeRight;

    // Update is called once per frame
    Frame currentFrame;

    // Use this for initialization
    void Start () {
        hc.GetLeapController().EnableGesture(Gesture.GestureType.TYPECIRCLE);
        hc.GetLeapController().EnableGesture(Gesture.GestureType.TYPESWIPE);
        hc.GetLeapController().EnableGesture(Gesture.GestureType.TYPE_SCREEN_TAP);
    }

    // Update is called once per frame
    void Update () {
        currentFrame = hc.GetFrame();
        GestureList gestures = this.currentFrame.Gestures();
        foreach (var g in gestures) {
            //Debug.Log(g.Type);
        }

foreach (var h in hc.GetFrame().Hands) {
    if (h.IsRight) {
        swipeUp = h.PalmPosition.ToUnity ().y - rightHandPosition.y;
        swipeRight = h.PalmPosition.ToUnity ().x - rightHandPosition.x;
        rightHandPosition = h.PalmPosition.ToUnity ();
        rightHandNorm = h.PalmNormal.ToUnity ();
    } else if (h.IsLeft) {
        leftHandPosition = h.PalmPosition.ToUnity ();
        leftHandNorm = h.PalmNormal.ToUnity ();
    }
}

item.GetComponent<Transform> ().localEulerAngles = new Vector3 (item.GetComponent<Transform> ().localEulerAngles.x + swipeUp,
    item.GetComponent<Transform> ().localEulerAngles.y + swipeRight,
    item.GetComponent<Transform> ().localEulerAngles.z);
swipeUp = 0;
swipeRight = 0;
    }
}
```

Figure 6.1.2: Code Snippet for the gesture recognition and position coding for the leap motion and the unity objects

b. Scale Button

When the scale button is pressed, you can scale the model by pinching your right hand then moving it towards the hologram (scale up) or away from the hologram (scale down). This works via a throttle system where the initial position the hand is detected by the leap motion is the neutral zone as seen in Figure 6.2.1. Move towards and away from the hologram screen while using the pinching action will scale the objects at different speeds. The related coding component can be seen in Figure 6.2.2.

Figure 6.2.1: Throttle and hand movement side by side comparison

Note: The hand model appears upside down because because the image is meant to be projected onto the prism as a hologram, when the user sees the projected hologram on the prism it would be reversed.

```
if ((zoomToggle == 1) && (handController.GetComponent<GestureController>().pinchStrengthL > 0.9)) {
    zSpeed = handController.GetComponent<GestureController> ().leftHandPosition.z - zPos;
    item.transform.localScale = new Vector3(zSpeed*0.0001f + item.transform.localScale.z,
        zSpeed*0.0001f + item.transform.localScale.y,
        zSpeed*0.0001f + item.transform.localScale.z);
    zPos = zSpeed;
    zSpeed = 0;
}
```

Figure 6.2.2: Code Snippet for the zoom feature.

c. Brush Button

Figure 6.3.1: Brush Modes

As seen in Figure 6.3.1, when the Brush button is activated it will either be green or red. In this mode, white means deactivated, green means extrusion while red means depression. When Brush is activated, a ray is cast from the centre of the user's left palm to model's centre and a red sphere is observed at the

intersection between the casted ray and the model's mesh as shown in the Figure 6.3.1. The vertices around the red sphere would be transformed.

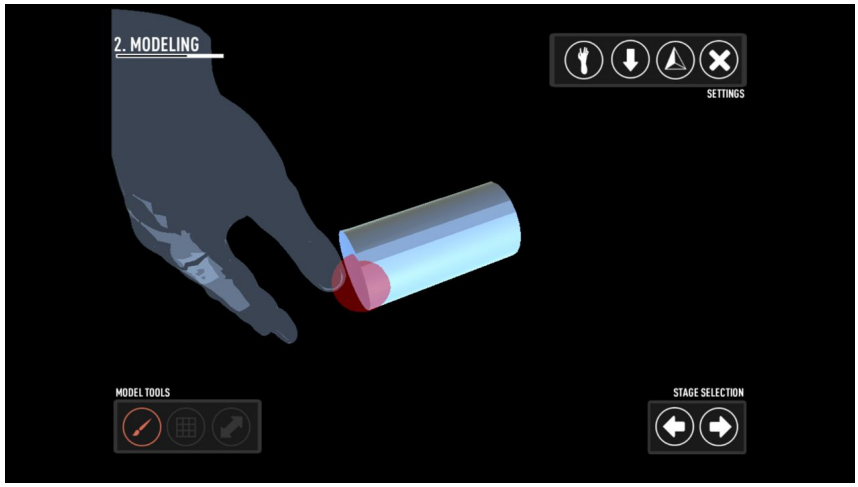

Figure 6.3.1: Ray Casting to Position Red Sphere

d. Extrusion / Depression

Both extrusion and depression are done by selectively ray casting and ensuring intersection with the sphere. Upon intersection, we transform the mesh vertices based on their normals via activation in pinching as seen in Figure 6.4.1 and 6.4.2. We tested the similar extrusion and intrusion using a pinch and drag movement but it does not provide a cohesive interaction as it feels buggy. The coding component is in Figure 6.4.3.

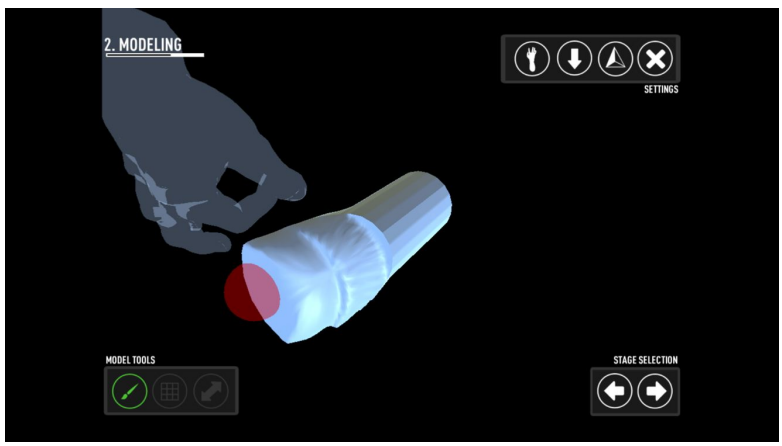

Figure 6.4.1: Brush Extrusion Feature

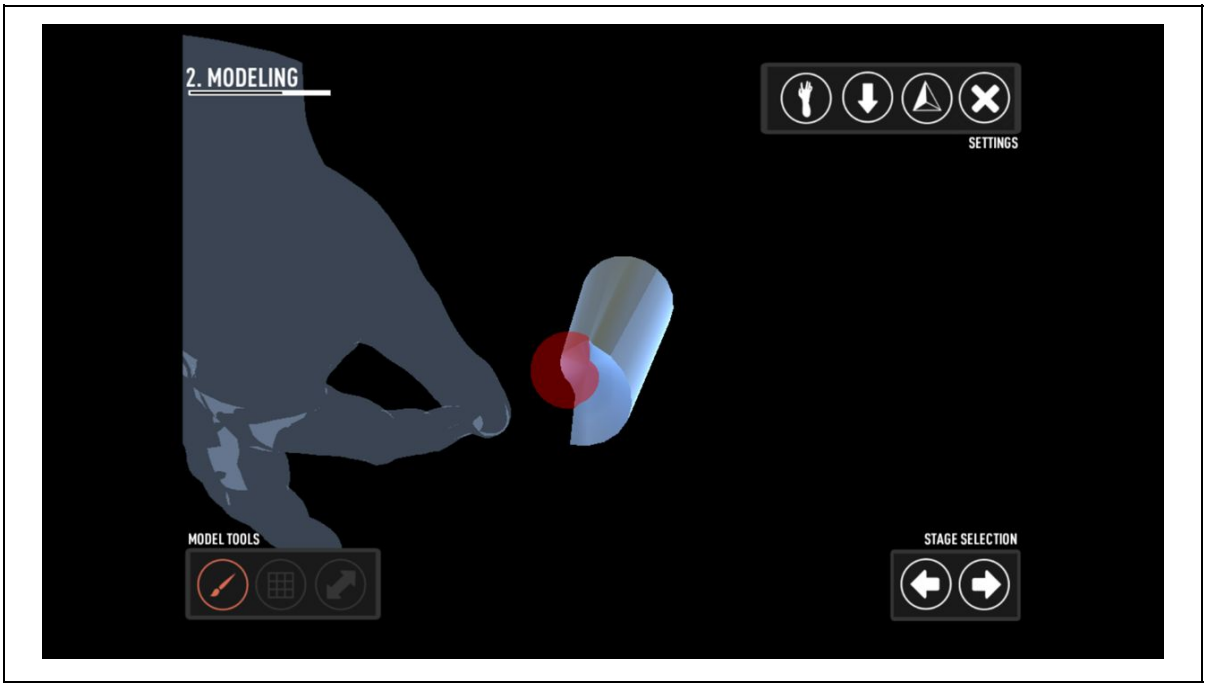

Figure 6.4.2: Brush Depression Feature

To use use this feature, position your left hand and use your right hand to rotate the object such that the red sphere is now located at the position where you want to make an extrusion / depression. Then pinch your left index finger with your left thumb. Once the pinch has been detected the extrusion / depression will start and continue until the pinch is released.

```
if ((brushToggle == 1)||(brushToggle == 2)) {
    sphere.SetActive (true);
    foreach (var h in handController.GetComponent<HandController>().GetFrame().Hands) {
        if (h.IsLeft) {
            dirOrigin = h.PalmPosition.ToUnity();
            dirDirection = -h.PalmPosition.ToUnityScaled ();
            pullMag = dirOrigin.magnitude;
        }
    }

    RaycastHit hit;
    //Debug.DrawRay (dirOrigin, dirDirection,Color.red,0.1f);
    if (Physics.Raycast (dirOrigin, dirDirection, out hit, Mathf.Infinity, layerMask)) {
        hitOrigin = hit.point;
        sphere.transform.position = hitOrigin;
    }

    if (handController.GetComponent<GestureController> ().pinchStrengthL == 1) {
        Vector3[] vertices = item.GetComponent<MeshFilter> ().mesh.vertices;
        for (int a = 0; a < item.GetComponent<MeshFilter> ().mesh.vertexCount; a++) {

            float selectionScale = sphere.GetComponent<Transform> ().localScale.x;
            Vector3 scaledVertices = new Vector3 ((vertices [a].x * item.transform.localScale.x),
                (vertices [a].y * item.transform.localScale.y),
                (vertices [a].z * item.transform.localScale.z));
            float circleValue = (sphere.transform.position - item.transform.rotation*scaledVertices ).magnitude;

            if (circleValue < selectionScale) {
                if (brushToggle == 1) {
                    vertices [a] = vertices [a] + Vector3.Scale(vertices[a],new Vector3(0.01f,0.01f,0.01f));
                } else if (brushToggle == 2){
                    vertices [a] = vertices [a] - Vector3.Scale(vertices[a],new Vector3(0.01f,0.01f,0.01f));
                }

            }
        }
        MF.mesh.vertices = vertices;
        MF.mesh.RecalculateNormals();
        item.GetComponent<MeshCollider> ().sharedMesh = MF.mesh;
```

Figure 6.4.3: Code Snippet for the extrusion feature.

e. Increase Vertices Count Button

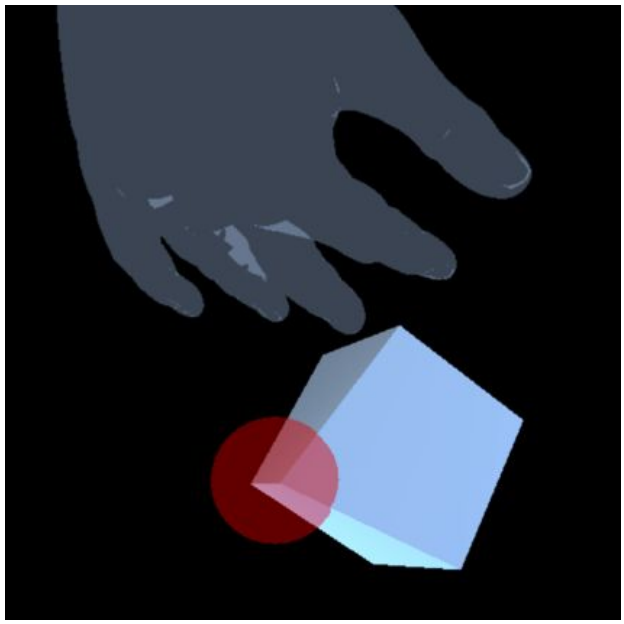

Figure 6.5.1: Extrusion with Eight Vertices Count

Here we use the default cube primitive as an example. By default the cube only has eight vertices thus the image in Figure 6.4.2 shows what happens when the cube is extruded where only one vertex is transformed which leads to straight edges after extrusion.

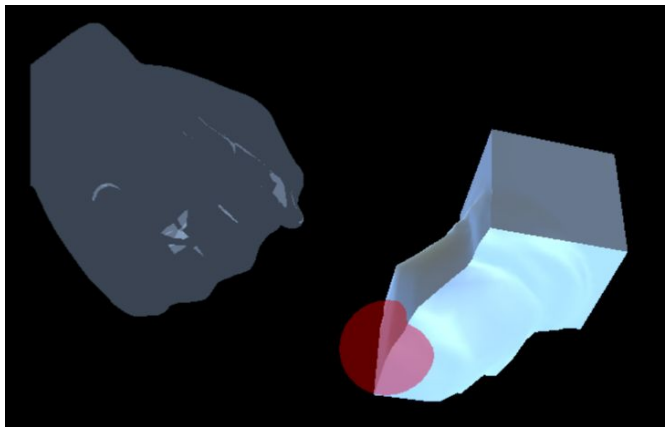

Figure 6.5.2: Extrusion with High Vertices Count

If we press the Increase Vertices Count button a few times (each press increases the number of existing vertices shown in Figure 6.4.4) then perform extrusion as shown in Figure 6.5.2, the extrusion now has a much smoother profile due to the

increased number of vertices being transformed. The code for the subdivision by Increase Vertices Count is Figure 6.5.3.

```
public void subdivide(){
    Mesh m = MF.mesh;
    Debug.Log (2 * Mathf.Pow (m.vertexCount, 0.5f));
    if ( 2* Mathf.Pow (m.vertexCount, 0.5f) < 75) {
        MeshHelper.Subdivide (m);
        MF.mesh = m;
        item.GetComponent<MeshCollider> ().sharedMesh = m;
    }else {
        subButton.GetComponent<Button> ().interactable = false;
    }
}
```

Figure 6.4.3: Code Snippet for the subdivision

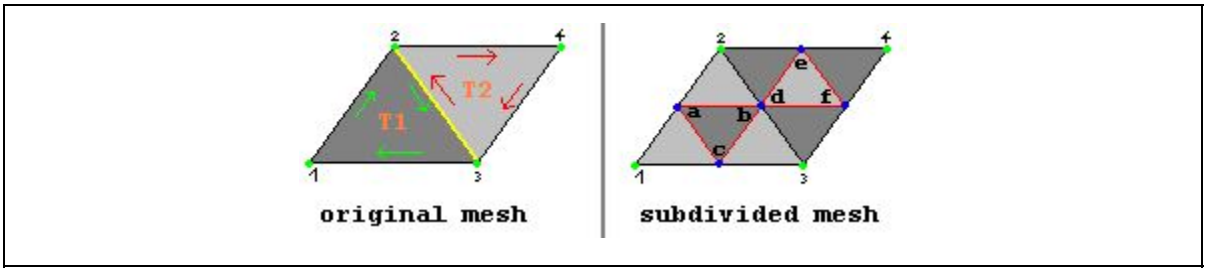

Figure 6.4.4: Subdivision algorithm visualized

7. Rigged Model

Clicking the hand icon in the settings menu will allow a leap motion test with a rigged model as seen in Figure 7.1. In its simplest form, 3D rigging is the process of creating a skeleton for a 3D model so it can move.

Figure 7.1: Hand Motions Can Be Used to Control Rigged Models

8. Simulation

a. Drop Test and Deformation Simulation

For our project, we have 2 simulation modes: Drop Test (fracture) and Deformation using the Leap Motion controller. Object to be imported must have the UV mesh file in order for the fracturing or deformation to work.

The Chart below shows what classes the object requires for fracturing/ deformation transformations to occur (apart from the transform and rigidbody classes by default): Mesh Filter, Mesh Renderer, Mesh Collider, Deform & Fracture script and the Leap Motion grabbable script (to communicate with the Leap Motion hand controller).

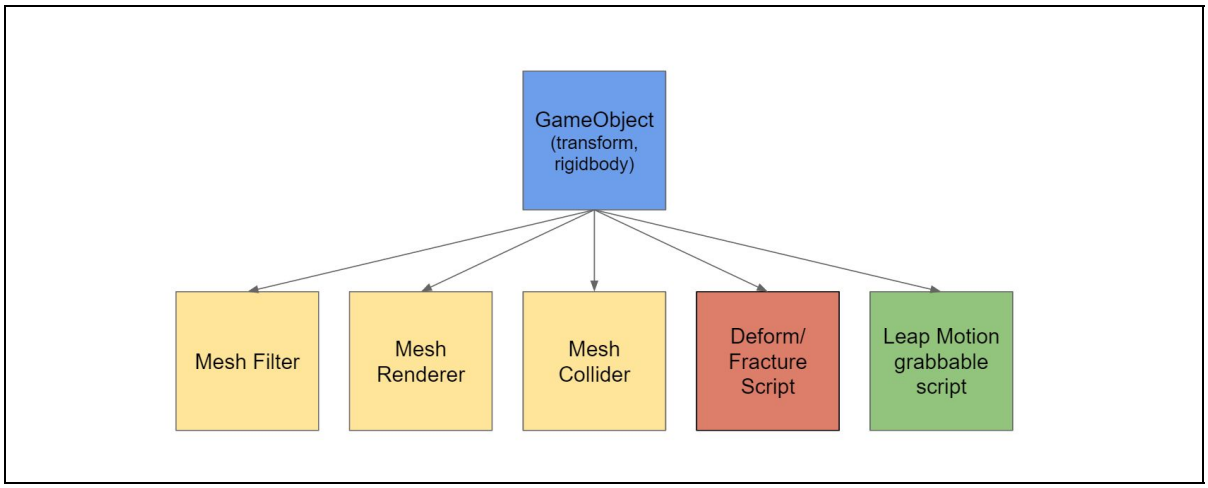

Figure 8.1.1: Setup of GameObject for Fracturing/Deformation Transformations
For Object to deform, we have allowed some variables to be changed, Impact Force, Impact Shape and Impact Type.

Impact force is given by the calculation of the following float variables: Sum of Forces * Force Multiplier - Force Resistance.

Impact Shape can be in the form of a spherical impact or a flat impact to determine the volume of vertices impacted upon collision.

Impact Type is the toggle between fracture physics and deformation physics. Below shows a condensed form of the impact function in the script.

```
public void Impact(Vector3 impactPoint, Vector3 impactForce, impactShape,
impactType)
...
float impactRadius = impactForce.magnitude;
...
float vertexForceMagnitude = Mathf.Max(0, (impactRadius - distance)) *
c_CompressionResistance;
...
if (distance < impactRadius && vertexForceMagnitude > 0)
movedVertexToForceMagnitudeMap.Add(i, vertexForceMagnitude);
```

Code snippet 8.1.2: Impact function in simulation tests

b. Fracture Simulation

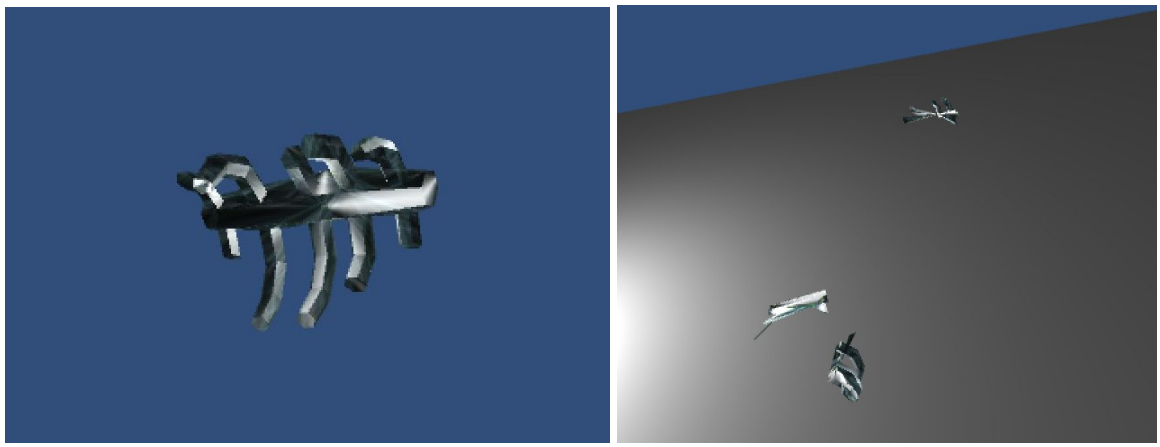

Figure 8.2.1: Fracturing GameObject

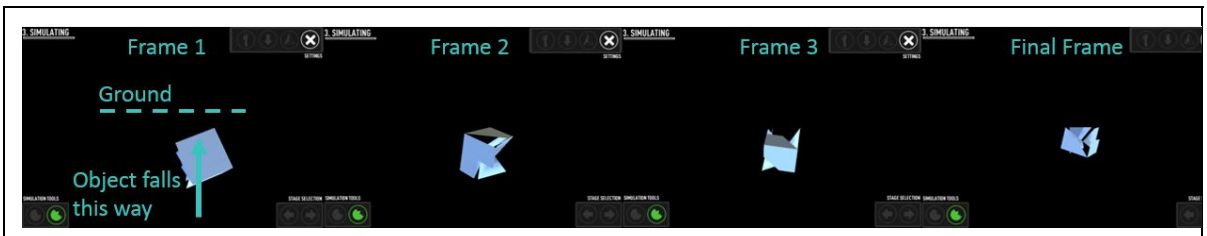

Figure 8.2.2: Fracture Simulation Integrated into our Software

In the Drop Test also known as Fracture Simulation, our software simulates the model dropping on a hard surface and visualises how the model would fracture. Note: the object falls upwards as shown on the screen but would be seen as falling downwards after projected as a hologram on the prism since the direction would be reversed.

c. Deformation Simulation

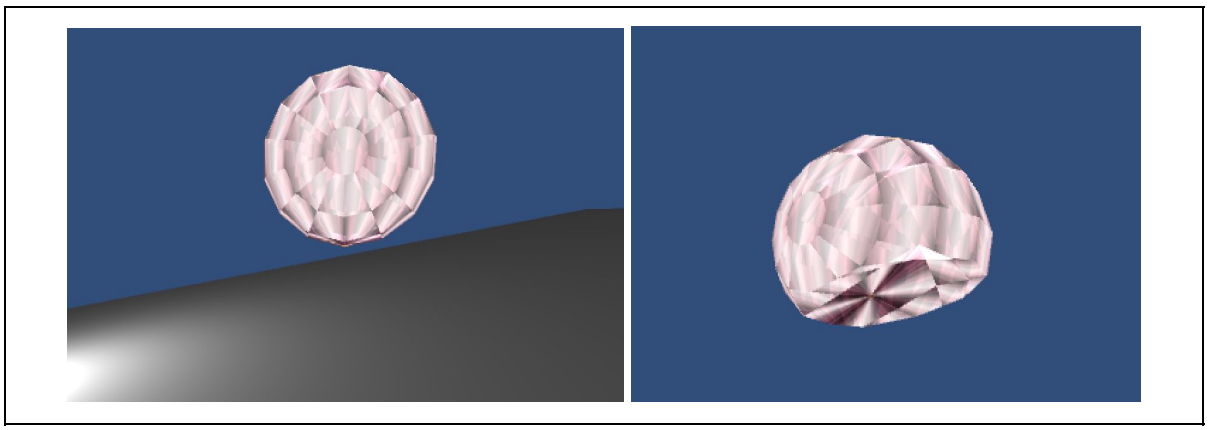

Figure 8.3.1: Deformed GameObject

Interfacing with the Leap Motion was also essential for the application. On the Hand Controller in the same scene, it contains the “Grabbing Hand” script which looks for objects with a “Grabbable“ script attached. This allows the Leap Motion controller to know which objects are available to interact with.

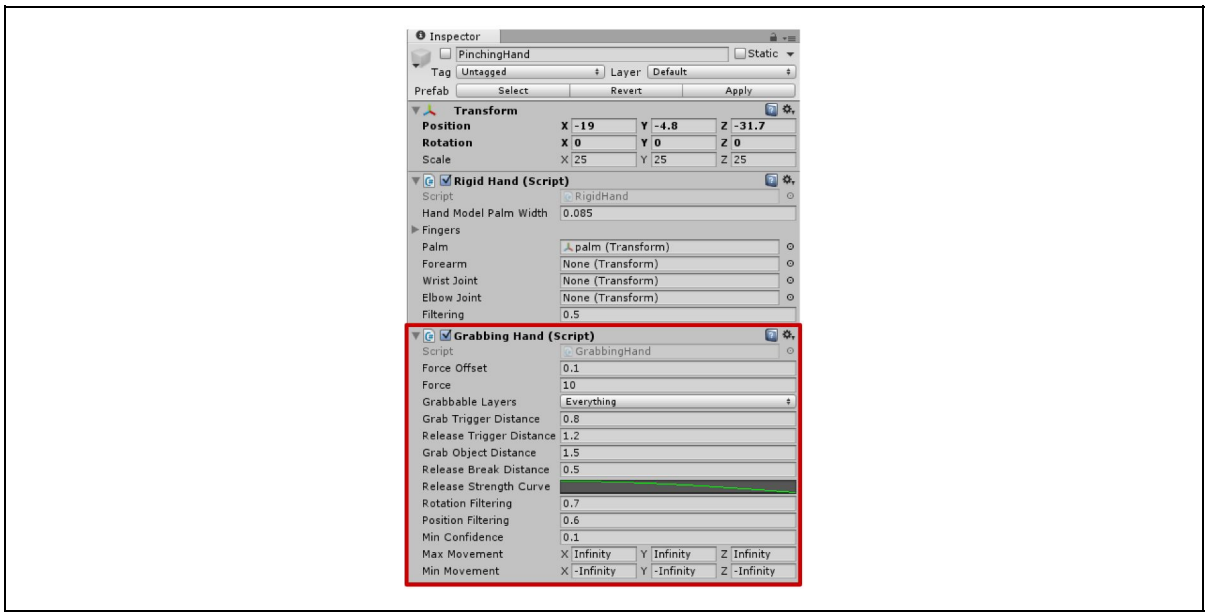

Figure 8.3.2: Leap Motion Hand Controller elements

This allows the user to deform the vertices of the object with the Leap Motion hand controller, seen below.

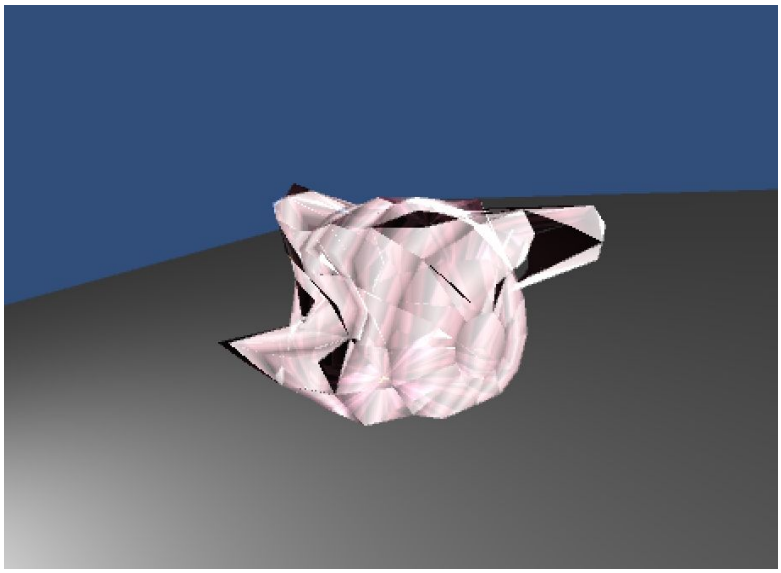

Figure 8.3.3: Leap Motion Object Deformation

Code snippet below shows how the control inputs are relayed from the "Grabbable" to the "Deform" script.

```
if (grabbable == null || !grabbable.centerGrabbedObject) {
    Vector3 delta_position = active_object_.transform.position -
current_pinch_position_;

    Ray pinch_ray = new Ray(current_pinch_position_, delta_position);
    RaycastHit pinch_hit;

if(Physics.Raycast(pinch_ray, out pinch_hit)){
                        MeshObject deformer =
pinch_hit.collider.GetComponent<MeshObject> ();
                        if (deformer) {
                                Vector3 point = pinch_hit.point;
                                point += pinch_hit.normal * forceOffset;
                                Vector3 direction = pinch_hit.normal;


deformer.Impact (point, direction * force, deformer.m_ImpactShape,
deformer.m_ImpactType);
                                                        }
```

Code Snippet 8.3.4: Leap Motion Control Inputs for grabbing objects

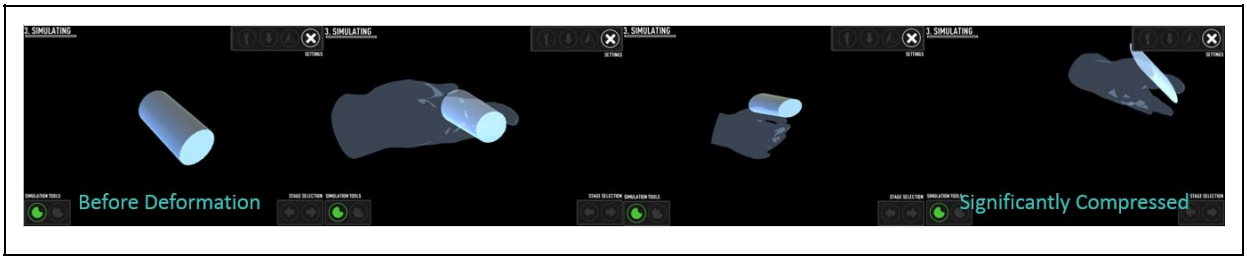

Figure 8.3.5: Deformation Simulation Integrated with Leap Motion in our Software

In the Deformation Simulation in our software, users can deform the model with their hands. A collision between the hand model and the simulated model will cause deformation of the simulated model.

9. Exporting Finished Model as .obj File

After the user has completed the design of the model and/or simulation, he/she can export the finished model as an .obj file to be used for 3D printing or other purposes.

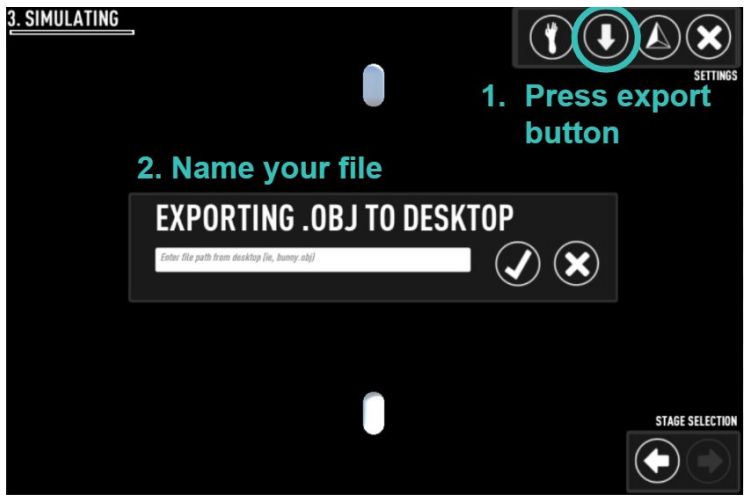

Figure 9.1.1: Exporting Your Completed Model

The code for exporting can be seen in Figure 5.1.2.

10. Conclusion

The unity application will show how we combined these features together in our project. Though tedious with much unexpected obstacles along the way, we have learned many other computer graphics algorithms and techniques outside of lessons from this final project!