# Abstract

Jack Watson

Student ID:16008335

Department of Computer Science and Creative Technologies

University of the West of England

Coldharbour Lane

Bristol, UK

[jack2.watson@live.uwe.ac.uk](mailto:jack2.watson@live.uwe.ac.uk)

The Project aims to create a built into Unity tool for creating destructive environments. This tool will select the type of material that will be simulated and then dynamically create said terrain / object.

# Introduction

Preliminary work for the project will aim to discover and implement the destruction of a virtual environment. Individual objects made up of different materials such as: Wood, Glass and Stone will be created with unique variations of destruction.

In older games destruction involved destroying either pixels of a wall (Figure 1) or hitting a crate sprite until it shows a shattered sprite and then left the screen after a delay.

Some games in the last decade have made leaps in this field such as the Dragon Ball Z Budokai Tenkaichi series with entire rocks being destroyed on impact or holes being created (though these were scripted events).

Destructive environments involve terrain or objects in a virtual being interacted by the player with enough force to cause damage like in the real world e.g. if a ceramic vase fell from 10 meters it would shatter.

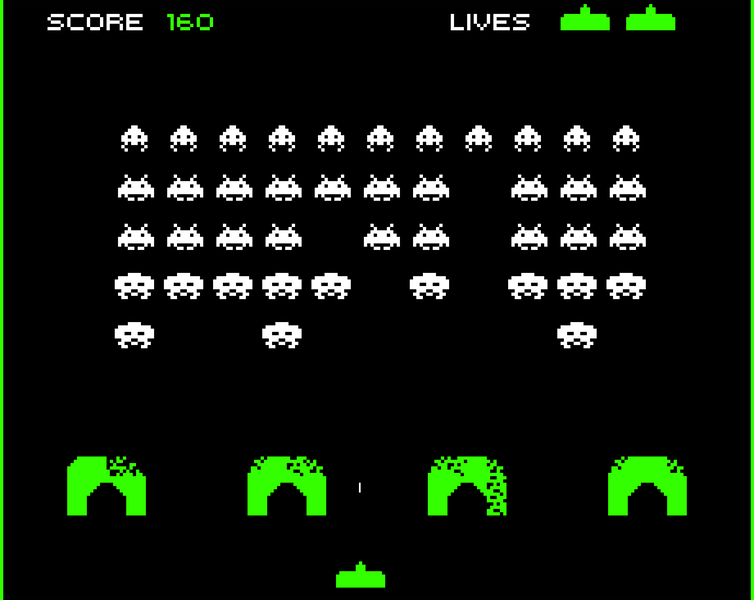


Figure 1

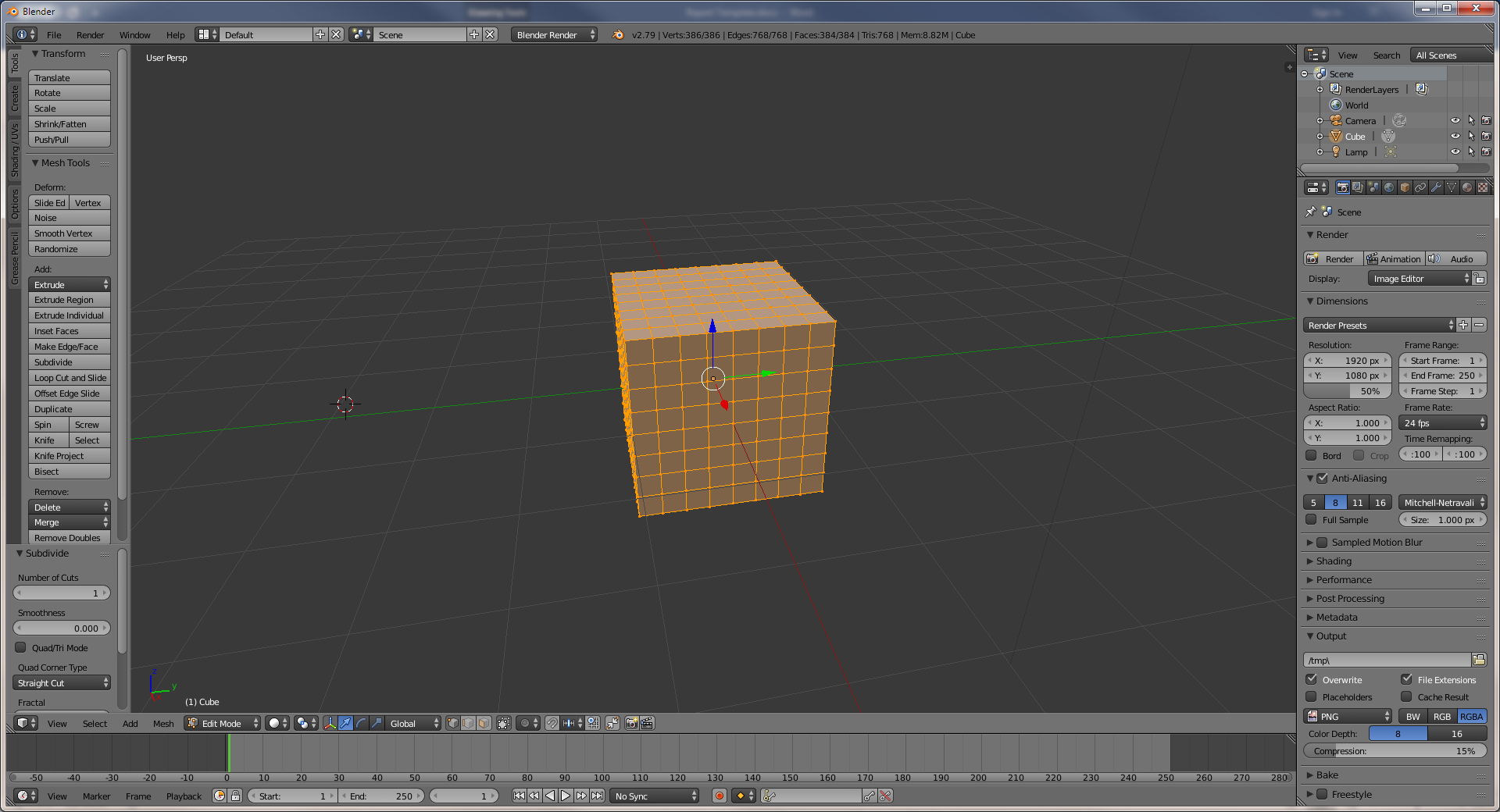
A tool will be devised allowing dynamic destruction via the unity pipeline, quickly creating destructible objects. Done efficiently, memory would be saved allowing the tool to be used in active environments such as in video games.

In a professional context this would allow creators to add the script to an object and destroy it in Unity without need for an editor like Maya or Blender. A tool of convenience.

Initial development of the solution will be covered, completion of the software will not be.

# Background Research

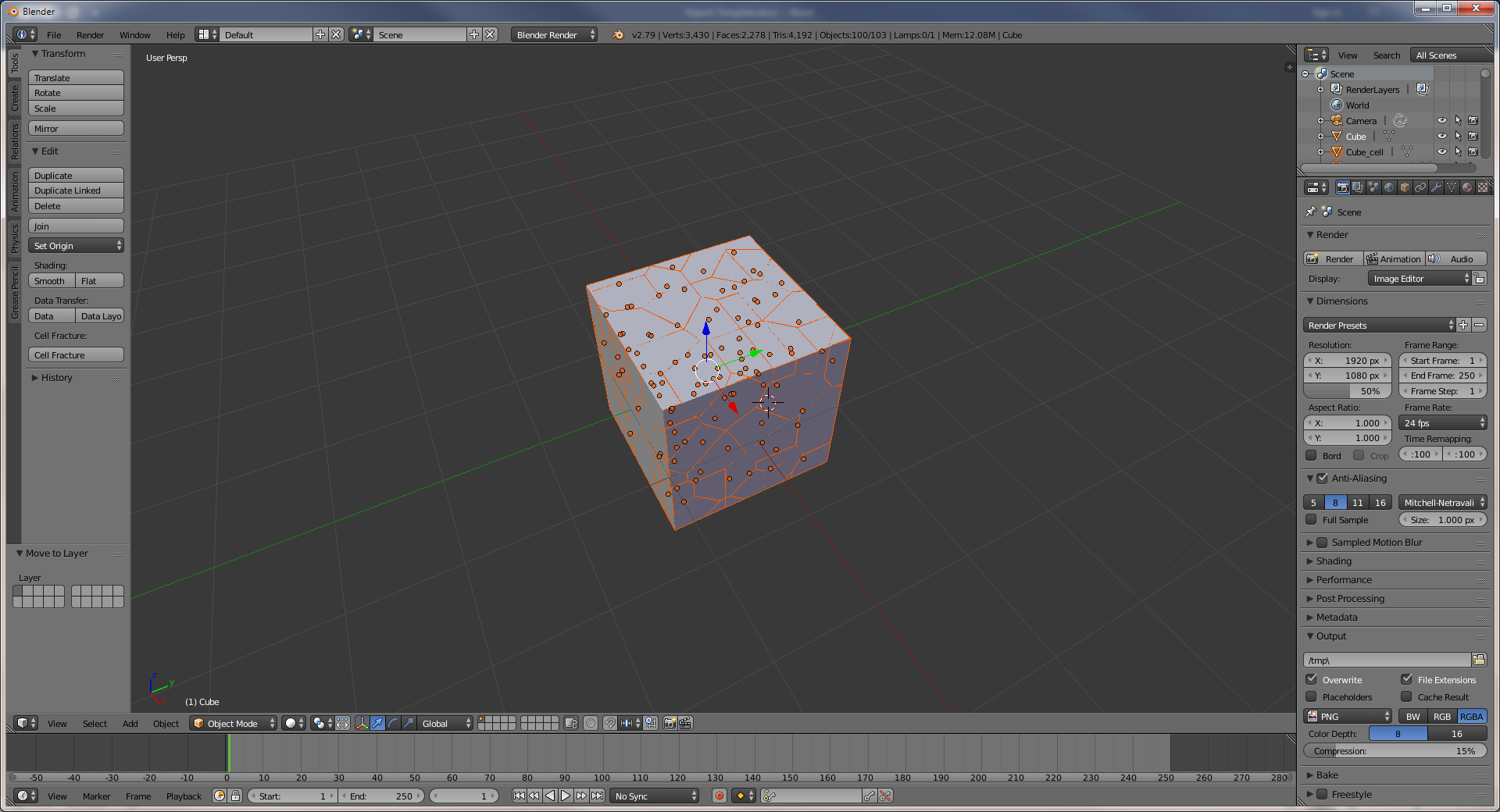
## **Research Aims/Questions:**

* How does basic destruction operate?
* How shattering works in software like blender?
* The Structures of different materials?
* Can destruction be computationally efficient?
* How do meshes operate in Unity?

## **Research methods**

Qualitive research: the task requires exploration of a (personally) unknown field, tutorials will be used and discoveries analysed.

Figure 3 Using the cell fracture addon mentioned in (DanielFilms, 2017).

Will look for accuracy of physical destruction, understanding of basic physics and the makeup of objects in a digital space.

Software’s (like blender) will be studied for how shattering is done, as this is needed for the unity pipeline.

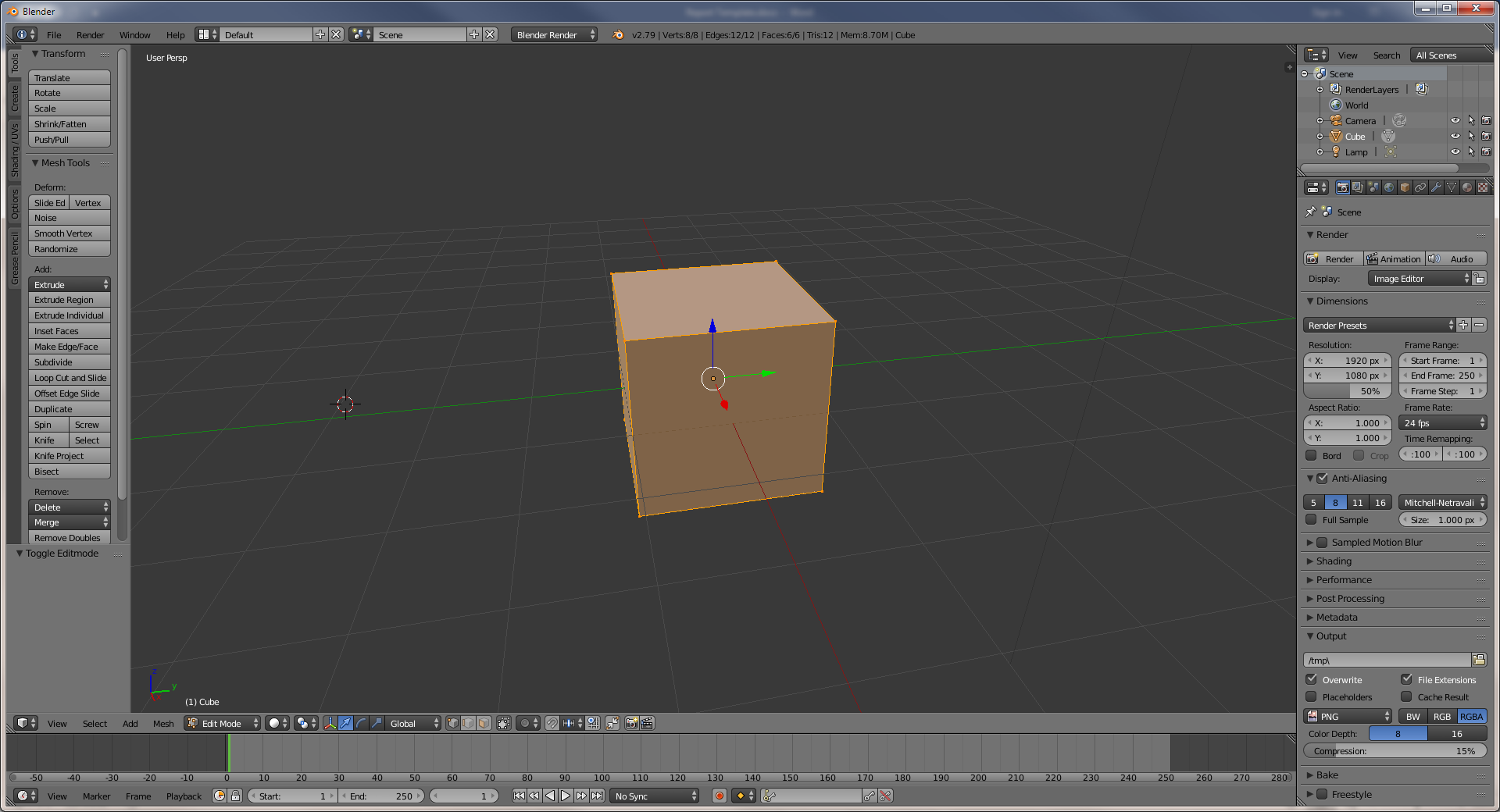
The internet is primarily used to find Unity specific documentation and tutorial aids.

Figure 1

A cube in blender

# Research Findings

## **How does shattering work in Blender?**

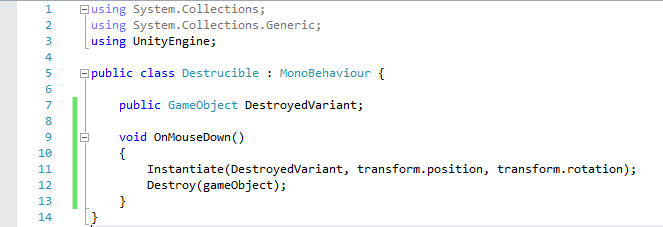
A tutorial by (DanielFilms, 2017) allowed insight towards destroying objects in a conventional manor. The technique involving the replacement of an object with a “shattered” variant.

Figure 2 After applying multiple subdivisions

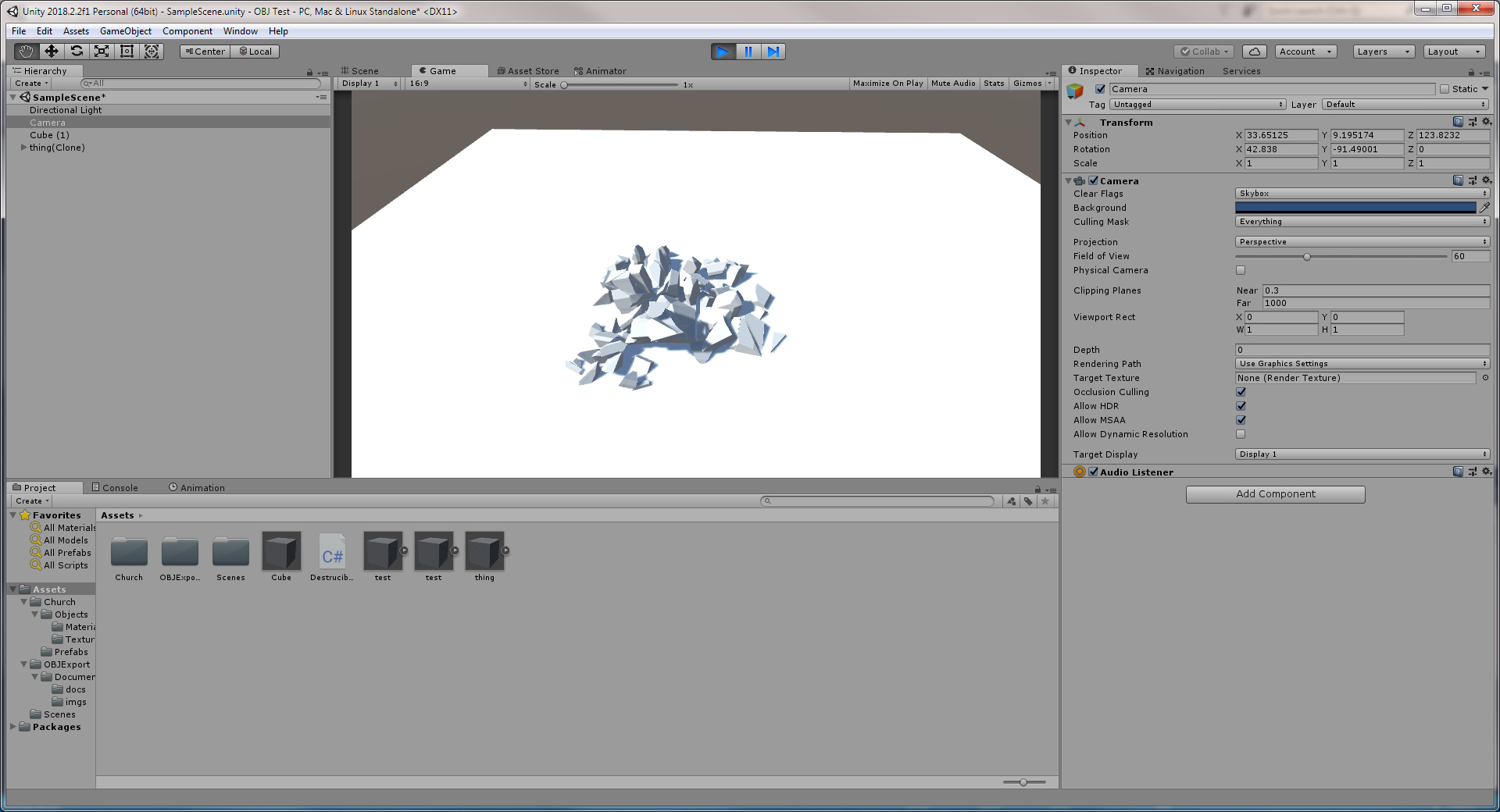


Figure 5 A simple cube shattering from gravity after being clicked on.

While the shatter (as seen in figure 5) is detailed for a rock like destruction, it does not reflect that of glass shatter or wood splinter effect.

Blender does incorporate a surface shatter which for flat objects yields a glass effect. It is not applied throughout the whole object.

Figure 4 And some basic code from (Brackeys, 2017)

## **How to shatter in Unity?**

Afterwards a premade tool allowing objects to be cut was discovered.

This being essential, as objects that shatter are required to break into pieces, this is covered in a simple manor with mesh alteration in the cut.

(BLINDED-AM-ME, 2018)’s MeshCut and MeshMaker class were downloaded and implemented using an array of game objects that stored the output.

public class CutInHalf : MonoBehaviour

{

void OnMouseDown()

{

GameObject[] pieces = MeshCut.Cut(gameObject, transform.position, transform.right, GetComponent<Renderer>().material);

for (int i = 0; i < pieces.Length; i++)

{

if (!pieces[i].GetComponent<BoxCollider>())

pieces[i].AddComponent<BoxCollider>();

if (!pieces[i].GetComponent<Rigidbody>())

pieces[i].AddComponent<Rigidbody>();

}

All the code below the cut adds a collider if there isn’t one to give the objects physics and prevent falling through the floor via gravity.

Figure 3 This cube upon the click of the mouse

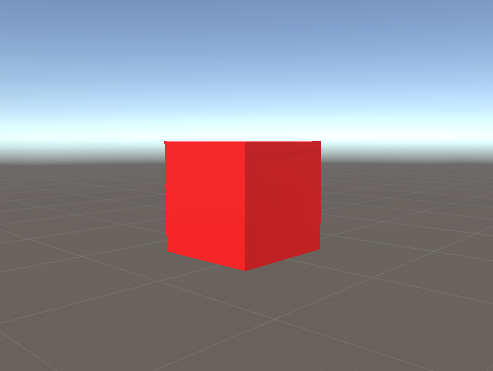


Figure 4 Is cut in two

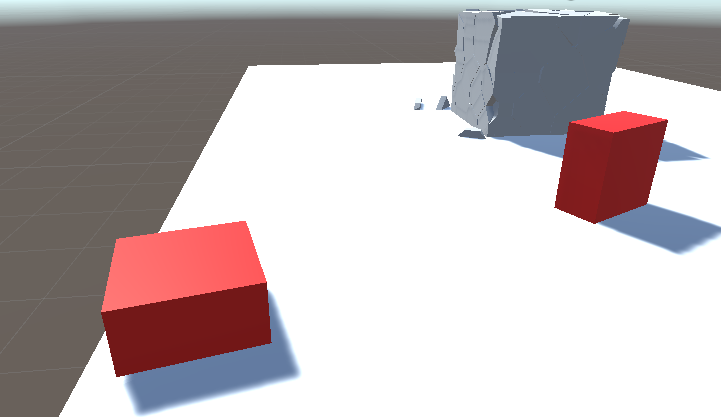
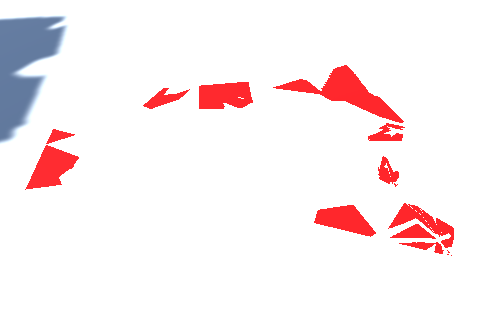


Figure 5 Although multiple clicks results in only shaving triangles of the object.



However, with the code provided in the classes used, gives insight into the mesh logic and can be extracted.

// set the blade relative to victim

\_blade = new Plane(victim.transform.InverseTransformDirection(-normalDirection),

victim.transform.InverseTransformPoint(anchorPoint));

// get the victims mesh

\_victim\_mesh = victim.GetComponent<MeshFilter>().mesh;

// reset values

\_new\_vertices.Clear();

\_leftSide = new Mesh\_Maker();

\_rightSide = new Mesh\_Maker();

A copy of the scripts object mesh is made. A “blades” position is established based on the scripts gameobject and inverted direction passed into the function.

The second class, Mesh Maker is then used to instantiate two objects representing the new objects that will be made as a result of the cut.

Mesh Maker includes all core data for a mesh such as vertices, normal and UV’s (as well as tangents).

A main function being AddTriangle which creates a new face for the shape using passed core data mentioned above.

\_leftSide.AddTriangle(

new Vector3[] { \_victim\_mesh.vertices[p1], \_victim\_mesh.vertices[p2], \_victim\_mesh.vertices[p3] },

new Vector3[] { \_victim\_mesh.normals[p1], \_victim\_mesh.normals[p2], \_victim\_mesh.normals[p3] },

new Vector2[] { \_victim\_mesh.uv[p1], \_victim\_mesh.uv[p2], \_victim\_mesh.uv[p3] },

new Vector4[] { \_victim\_mesh.tangents[p1], \_victim\_mesh.tangents[p2], \_victim\_mesh.tangents[p3] },

sub);

The victims mesh is used to create the new one. The new triangles are added to the mesh maker objects Vector arrays containing core information.

// Left Mesh

Mesh left\_HalfMesh = \_leftSide.GetMesh();

left\_HalfMesh.name = "Split Mesh Left";

// Right Mesh

Mesh right\_HalfMesh = \_rightSide.GetMesh();

right\_HalfMesh.name = "Split Mesh Right";

The new meshes created are based on the modified mesh maker values.

The original victim is used for the left side. For the right side, a new object is created using the modified right half mesh maker object.

GameObject leftSideObj = victim;

GameObject rightSideObj = new GameObject("right side", typeof(MeshFilter), typeof(MeshRenderer));

rightSideObj.transform.position = victim.transform.position;

rightSideObj.transform.rotation = victim.transform.rotation;

rightSideObj.GetComponent<MeshFilter>().mesh = right\_HalfMesh;

If the triangle sides detected by the blade are not on the same side of a plane as each other,

sides[0] = \_blade.GetSide(\_victim\_mesh.vertices[p1]);

sides[1] = \_blade.GetSide(\_victim\_mesh.vertices[p2]);

sides[2] = \_blade.GetSide(\_victim\_mesh.vertices[p3]);

if (sides[0] == sides[1] && sides[0] == sides[2])

then the triangle at the core is cut.  
  
 { // cut the triangle

Cut\_this\_Face(

new Vector3[] { \_victim\_mesh.vertices[p1], \_victim\_mesh.vertices[p2], \_victim\_mesh.vertices[p3] },

new Vector3[] { \_victim\_mesh.normals[p1], \_victim\_mesh.normals[p2], \_victim\_mesh.normals[p3] },

new Vector2[] { \_victim\_mesh.uv[p1], \_victim\_mesh.uv[p2], \_victim\_mesh.uv[p3] },

new Vector4[] { \_victim\_mesh.tangents[p1], \_victim\_mesh.tangents[p2], \_victim\_mesh.tangents[p3] },

sub);

}

This means as the information is fed into the leftSide and rightSide vectors the original object is deleted face by face as the counter is incremented.

// The capping Material will be at the end

Material[] mats = victim.GetComponent<MeshRenderer>().sharedMaterials;

if (mats[mats.Length - 1].name != capMaterial.name)

{

Material[] newMats = new Material[mats.Length + 1];

mats.CopyTo(newMats, 0);

newMats[mats.Length] = capMaterial;

mats = newMats;

}

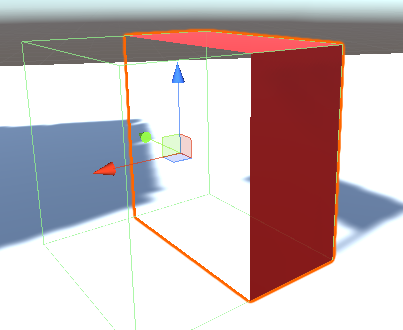
The materials are copied from the original and then allocated to the new objects.

// assign mats leftSideObj.GetComponent<MeshRenderer>().materials = mats;

rightSideObj.GetComponent<MeshRenderer>().materials = mats;

A function called Capping is run. This recreates the face at the cut to connect all the other faces, forming the original shape.

i.e without this function a 5 sided cube is produced.



Capping copied the vertices from the initial cut.

capVertpolygon.Add(\_new\_vertices[i]);

capVertpolygon.Add(\_new\_vertices[i + 1]);

The ‘cap’ is filled. The blades normal is used for producing the faces normal.

UV’s use the center, which is divided by the number of vertices total

center = center / vertices.Count;

displacement = vertices[i] - center;

newUV1 = Vector3.zero;

newUV1.x = 0.5f + Vector3.Dot(displacement, left);

newUV1.y = 0.5f + Vector3.Dot(displacement, upward);

displacement uses the center and the combination of vectors displacement and direction inside of Dot create the UV vector.

FlipFace(final\_verts, final\_norms, final\_uvs, final\_tangents);

This function flips the caps depending on what side is capped. This is done by swapping vertices, normals and tangents around.  
  
 private static void FlipFace(

Vector3[] verts,

Vector3[] norms,

Vector2[] uvs,

Vector4[] tangents)

{

Vector3 temp = verts[2];

verts[2] = verts[0];

verts[0] = temp;

temp = norms[2];

norms[2] = norms[0];

norms[0] = temp;

Vector2 temp2 = uvs[2];

uvs[2] = uvs[0];

uvs[0] = temp2;

Vector4 temp3 = tangents[2];

tangents[2] = tangents[0];

tangents[0] = temp3;

}

The add triangle function is used again to fill cap vertex

\_rightSide.AddTriangle(final\_verts, final\_norms, final\_uvs, final\_tangents,

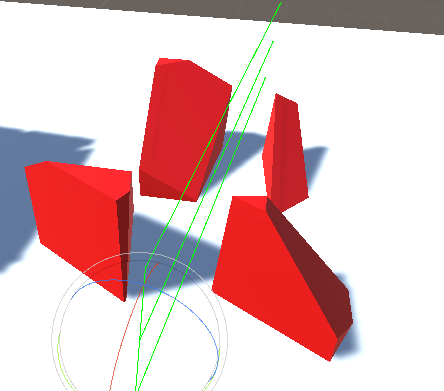
\_capMatSub);

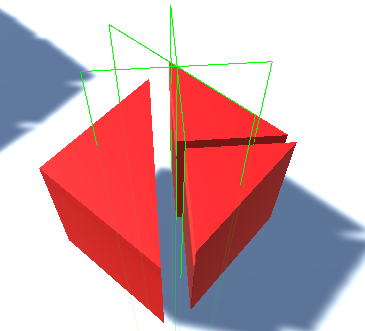
\_leftSide.AddTriangle(final\_verts, final\_norms, final\_uvs, final\_tangents,

\_capMatSub);

When the blade is used from (W., 2015), the coordinates are more detailed than a mouse’s postion.

This removed the issue before where vertices would disappear and thin slices of cube were removed after the initial cut.

This resulted in:  


With multiple blades cutter shapes could be made to create different combinations of destruction.  
  
However, in testing this proved difficult.  
  


This combination should have produced six pieces, but only produced three.

# Evaluation

Currently, using logic from the (BLINDED-AM-ME, 2018) code base allows some of the desired effects, as shattering an object includes effectively cutting it into multiple pieces. However, the tool only supports cutting the first object hit by the ray cast.

For the demo, pieces should be produced by multiple blades simultaneously to allow a shatter like the Blender cube shattered prior.

The ordering of the blades based on the material selected will be the final goal.

# References

# Bibliography

BLINDED-AM-ME. (2018, April 15). *Unity Assets*. Retrieved November 18, 2018, from Github[online]: https://github.com/BLINDED-AM-ME/UnityAssets

Brackeys. (2017, February). *SHATTER / DESTRUCTION in Unity (Tutorial)* [online]. Retrieved [October 29, 2018], from Youtube: https://www.youtube.com/watch?v=EgNV0PWVaS8

DanielFilms. (2017, April). *How to make objects shatter in blender*[online]. Retrieved October 29, 2018, from Youtube: https://www.youtube.com/watch?v=SlJEInN\_Els

Games, B. t. (2016, September 15). *Procedural Mesh Tutorial, Part 4: All About Vertices* [online]. Retrieved December 06, 2018, from Youtube: https://www.youtube.com/watch?v=IJ9Tla\_Q4gk

W., D. (2015, September 13). *Unity simple Mesh cutting Code* [online]. Retrieved December 07, 2018, from Youtube: https://youtu.be/xgoUmrhXyYE