TUS

Title of Project:
Deformable body physics for the use of small and indie developers within Unity 3D.

Name of Student: *Conor Liddy*
Student Number: *K00262125*
Date: *14 / 03 / 2024*
Word Count:

I declare that no element of this assignment has been plagiarised.

Student Signature: _____

Contents

Introduction, Problem Definition & Background

There does not currently exist a publicly available way to implement deformable body physics into a modern Unity Project, in a simple and user-friendly way. In the current release of Unity, as well as all previous releases, it only includes built in implementation for the use of Rigid bodies. While this is particularly good for applications relating to solid or "Rigid" objects, this does not allow for the use of deformable body physics which can be applied to a Unity object. This became an issue for me personally when I was trying to make a game within Unity 3D during a break between college years. I needed a deformation system for this game and began looking into publicly available solutions for this purpose. Upon looking into this I realized that there really were not any available that fit my use case. One of these solutions I discovered is the "DefKit – Deformable (Soft) Bodies Toolkit," which was released on October 2nd, 2012, by a Unity forum user named "Korzen303". This toolkit is perhaps the closest to the final result I wish to create as it is easy to use for developers and the parameters are easily accessible in the unity editor. However, this option will not work in most cases simply due to the age of the release. The visuals and code are quite outdated, and the toolkit is not compatible with the latest versions of the Unity editor. A more modern alternative which I found is "Scrawk, 2D Deformable body in Unity" physics library. This library was published to GitHub on January 30th, 2022, by a User named "Justin". This physics library does work with the most recent versions of Unity due to its more modern publish date. However, the downside of this system is that, as the name suggests, it only works with Unity 2D. It cannot be applied to a Unity 3D application. The last of these alternatives I will discuss is the "Truss physics for Unity3D". This one was also published in 2012 on Nov 18th, however, this one has continued to be updated and works with the latest versions of Unity. The issue with this library is that it does not allow for the ease of use for the developer as each object would need to be hard coded. All of these also share an overlapping issue which is that they all struggle with more complex objects such as those imported by a developer.

With this known the goal of my final year project is to not only create a deformable body physics library for use with Unity 3D, but also to create in such a way that it can be universally applied to any object within the Unity editor and provide an easy to access set of parameters for the developer adjust from within the unity editor itself. By the end of this thesis, I will have explained how I achieved each of these goals and solved the problems associated with them. I will also discuss future changes which could be made to improve upon the final result.

Feature Set

➢ UNITY INTEGRATION

This package must be written in C# as it is integral to the functionality of the system that it seamlessly and easily works with Unity 3D. Without this feature, the library would be useless on its own. It will also make use of many existing Unity functions which will allow for greater optimisation.

➢ AUTOMATED MESH DETECTION FOR EXISTING UNITY OBJECTS

The package must be able to automatically detect and use existing mesh vertices of objects within Unity. Without this feature developers would be required to code these themselves, similar to other solutions which I discussed. This modularity and ability to apply the package to any object with one action will be a stand-out feature of my deformation physics package.

➢ BREAKING UNITY OBJECTS INTO A 3D ARRAY MASS

One system I considered as a possibility for this project was the use of a 3D array mass system. A solution such as this could be used to simulate volumetric deformation to create a soft body object

similar to jelly. However, after working on this system for a short while I realised that its primary drawback would be in performance cases. This system works by breaking an object into a subset of smaller cubes known as "voxels," this is similar to games like Teardown released in 2022. This would allow the object to move as though it were a non-rigid body as we could simply move around each voxel independently to create this effect. The issue arises due to the quantity of voxels needed, with well over 10'000 objects being created for a simple cube. This greatly affects the performance of the game even when only rendering voxels which appear on the outside faces of the object. The package does still include features from this system which includes a crumble physics feature.

➢ TURNING UNITY OBJECT MESH DATA INTO A DYNAMIC SPRING SYSTEM

The primary system by which this physics system gets implemented is through the creation of a mass-spring system within unity. My solution uses the existing mesh data of unity objects in order to create a dynamic mesh system which can replicate the functionality of a mass-spring system within Unity 3D. This system can be used to implement deformation physics to the mesh of an object in Unity.

➢ DEFORMABLE BODY PHYSICS IMPLEMENTATION

The final result of this project is the creation and implementation of a system that allows for the simulation of deformable bodies within a Unity project. This is done in a way that does not heavily impact performance and is easy for developers to implement and use.

➢ COMPATIBILITY WITH THE LATEST VERSIONS OF UNITY

The library is compatible with the most recent versions of Unity, allowing developers to take advantage of new features and improvements. This ensures that if a developer so needed they could also continue to make updates themselves with newer Unity releases.

➢ UNIVERSAL APPLICATION TO ANY OBJECT WITHIN THE UNITY EDITOR

The solution can be applied to any object in the Unity editor, regardless of its complexity or whether it was imported from external sources. This will be the primary feature which differentiates my package from others which are available online. Most other publicly available options require the code to be changed for each object, or they struggle with more complex models.

➢ USER-FRIENDLY PARAMETER ADJUSTMENT

The solution contains an intuitive user interface within the Unity editor that enables developers to easily adjust parameters related to, deformation, manual sine functions, etc., providing fine-grained control over how objects behave under various forces and interactions.

➢ REAL-TIME SIMULATION PERFORMANCE OPTIMIZATION

In order to make it so that this library is a viable option for developers I ensured that I must optimize computational efficiency to ensure real-time performance even when simulating complex deformations on multiple objects simultaneously. This is because the system will ultimately be useless if it cannot also interact with other objects within unity.

➢ COLLISION DETECTION AND RESPONSE

The package includes support for collision detection algorithms specifically tailored for deformable bodies, allowing them to interact realistically with other rigid or deformable objects in

the scene while preserving physical accuracy. This is needed as it can otherwise not be fully considered a physics system without reaction to the environment around it.

Prospective users

➢ GAME DEVELOPERS

Unity 3D game developers who want to incorporate realistic deformable body physics into their games. This package will provide them with an easy-to-use solution that can be applied to any object within the Unity editor, enhancing the realism and immersion of their game worlds.

➢ VIRTUAL REALITY (VR) DEVELOPERS

VR developers who create immersive experiences can use this library to add a new level of realism by simulating deformable objects in their virtual environments. To create a greater sense of realism within the virtual experience

➢ SIMULATION ENGINEERS

Professionals working in fields such as engineering, architecture, or robotics may find this library useful for simulating real-world behaviours of materials like cloth, rubber, or fluids in order to test designs or simulate physical interactions accurately.

➢ EDUCATIONAL INSTITUTIONS

Universities and educational institutions teaching computer graphics or physics courses can utilize this library as a learning tool for students interested in understanding advanced physics simulations and deformable body dynamics within Unity.

➢ INDIE DEVELOPERS

Independent game developers with limited resources can benefit from using this library instead of having to develop complex soft-body physics systems from scratch, saving time and effort while still achieving high-quality results.

➢ HOBBYISTS/ENTHUSIASTS

Amateur game developers or individuals interested in experimenting with interactive simulations within the Unity engine can explore creating dynamic scenes involving deformable bodies using this user-friendly library.

➢ RESEARCH COMMUNITY

Researchers studying various aspects of soft body dynamics, material behaviour modelling, or computational physics may employ this library as a foundation for conducting experiments or exploring innovative simulation techniques within Unity 3D.

Literature Review and Research

To create this deformable body physics system, I relied on various sources of inspiration and information. A lot of knowledge and understanding went into its development, as well as extensive work to gather and combine the necessary information. In order to ensure the accuracy and effectiveness of my system, I conducted a thorough review of existing literature. This involved exploring research articles, academic journals, and books that covered similar topics or ideas. By drawing from these diverse sources, I was able to establish a strong foundation for my innovative system. Through careful analysis and synthesis of this wealth of knowledge, I gained valuable insights needed to seamlessly integrate different aspects into a cohesive framework.

- ***DAVID H. EBERLY, CHAPTER 4 - DEFORMABLE BODIES, EDITOR(S): DAVID H. EBERLY, GAME PHYSICS (SECOND EDITION), MORGAN KAUFMANN, 2010, PAGES 155-212, ISBN 9780123749031,HTTPS://DOI.ORG/10.1016/B978-0-12-374903-1.00004-9. (https://archive.org/details/gamephysics0000eber_w3s3/page/n5/mode/2up)***

This Book has a large amount of information which I found very Useful for the initial research of my final year project. It contains a large amount of information around the mathematics and systems required to implement a deformable body physics system. This also introduced me to the concept of 3D Array Masses, as well as the idea of implementing elasticity, stress, and strain. It also gave a vast ammount of information on each of these, “Elasticity is the property by which the body returns to its original shape after forces cause deformation. E.g. If a foam ball is flattened, it will return to its original shape, however, if a metal ball is flattened it will stay flat, this is because it is less elastic than the foam ball.”, “Stress is the magnitude of the applied force divided by the surface area over which the force acts. The stress is large when the force magnitude is large or when the surface area is small. A rigid body of mass M is subject to gravitational forces and sits on top of a circular top of a cylinder of radius R, the cylinder has a stress on it of: $g / (\pi R^2)$”. This inclusion of the mathmatical equations required within the actual content of the book is what ultimately saved me a lot of time in the research of how to actually implement such a system. “Strain is the fractional deformation caused by stress. The quantity is dimensionless as it measures a change in dimension relative to the original dimension. The method for measuring this change depends on the type of object and force applied. Stress and strain do nothing on their own (Both are needed). The amount of stress needed to produce a strain is material specific.”

The primary application of this research was in the calculation of deformation amounts. This book provides a wealth of knowledge on the functionallity of planar, linear, and volume deformation. Linear deformation can essentially be defined as the deformation which takes place at the hard edges of an object. E.g “ If a wire of length L and a cross-sectional area of A has a force of F applied to it, a change in length of ΔL occurs.”

$$Y = \frac{\text{Linear Stress}}{\text{Linear Strain}} = \frac{F / A}{\Delta L / L}$$

It also provided a lot of context on the functionality of planar deformation which, is in essence. The deformation which takes place on the flat faces of an object. E.g. “ Consider a thin, rectangular slab, which is L thick and has two (2) more dimensions of X and Y. The large faces of the slab have an area of A = XY. The slab is placed on a table and a force of magnitude F and direction that of the X dimension is applied to the other face. This causes a shearing stress of F/A. The slab

slightly deforms into a parallelepiped with a volume the same as the original slab the area of the large faces also remains the same. The slab has an edge length of L which increases slightly by an amount of ΔL. The shearing strain is ΔL/L. The Shearing modulus is: “.

$$S = \frac{\text{Planar Stress}}{\text{Planar Strain}} = \frac{F / A}{\Delta L / L}$$

The volumetric deformation of an object can also be described in a similar way as it is essentially the deformation of the overall shape of the object as it is unlikely to decrease or increase in overall volume. E.g. “Consider a material occupying a volume of V. A force of F is uniformly distributed over the surface of the material. The direction of the force is perpendicular to each point on the surface. If the surface area is A, then the pressure on the material is P = F/A. If the pressure is increased by ΔP, the volume decreases by ΔV, the volume stress is ΔP and the volume strain is ΔV/V. The Bulk modulous is: “.

$$B = \frac{\text{Volume Stress}}{\text{Volume Strain}} = \frac{\Delta P}{\Delta V / V}$$

- ***IAN MILLINGTON, 12 - COLLISION DETECTION, EDITOR(S): IAN MILLINGTON, GAME PHYSICS ENGINE DEVELOPMENT (SECOND EDITION), MORGAN KAUFMANN, 2010, PAGES 253-289, ISBN 9780123819765, HTTPS://DOI.ORG/10.1016/B978-0-12-381976-5.00012-7. (https://www.sciencedirect.com/science/article/pii/B9780123819765000127)***

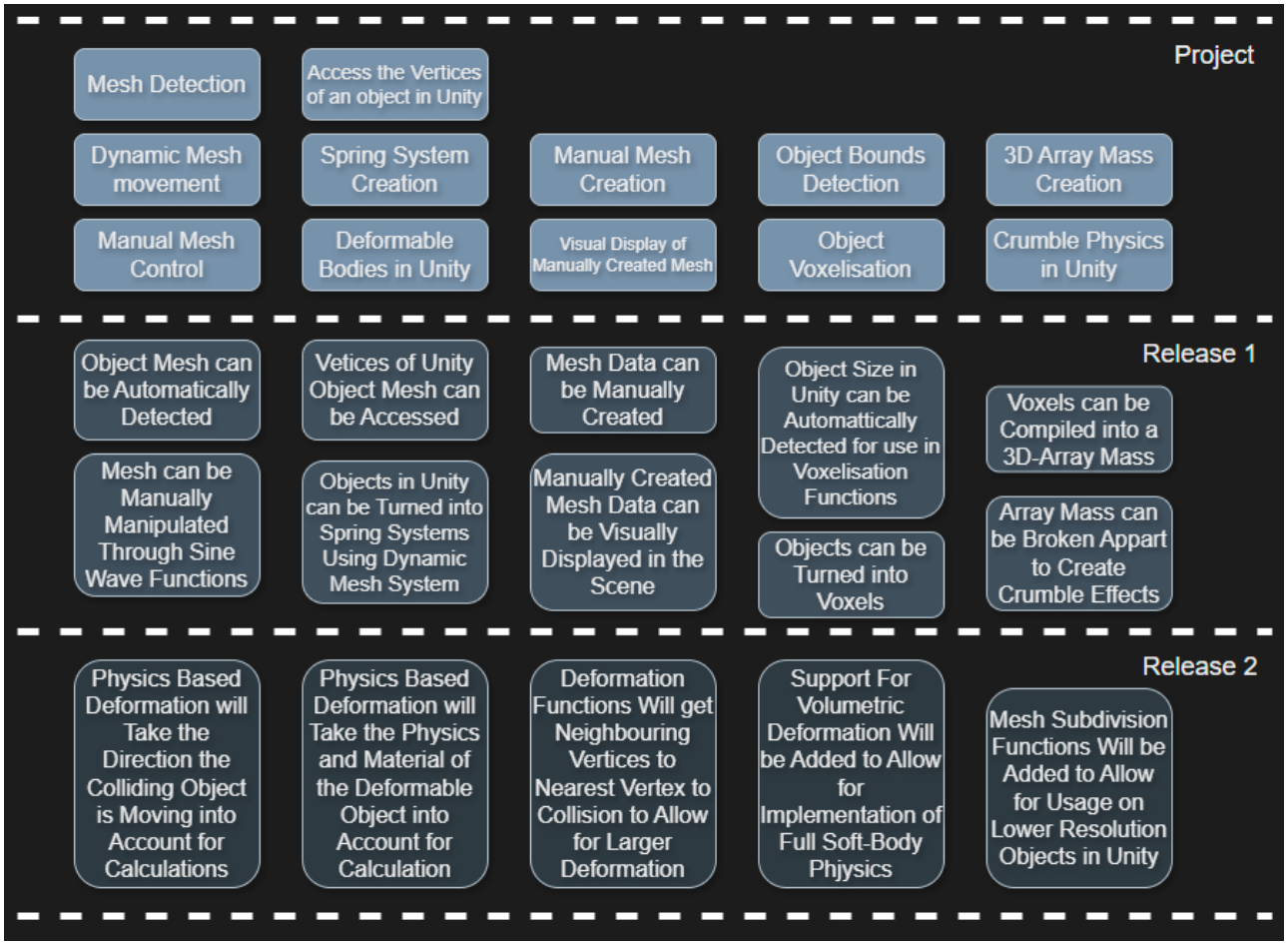
This Book gave a large amount of context on how collision detection is handled in game engines. This was very important to understand as a large amount of the application for deformable-body physics is in collision handling. This is because the library needs to be able to handle the interactions between objects.

- ***SINGH, N.P., SHARMA, B. AND SHARMA, A. (2022). PERFORMANCE ANALYSIS AND OPTIMIZATION TECHNIQUES IN UNITY 3D. [ONLINE] IEEE XPLORE. (HTTPS://DOI.ORG/10.1109/ICOSEC54921.2022.9952025).***

This paper discusses the best techniques to preserve and increase performance within Unity 3D. This was very important as performance is one of the main challenges, and priorities when writing a deformable-body physics system. This is especially important when working with 3D array masses. One suggestion it had to fix this was through Occlusion Culling. Occlusion culling is a technique used in computer graphics to improve the performance of rendering by reducing the number of objects that need to be rendered. In a 3D scene, there are often objects or parts of objects that are not visible to the viewer because they are hidden behind other geometry or obstacles. These hidden objects do not contribute to the final image and can consume valuable computational resources if they are still processed and rendered. In this case this led me to the idea to de-render the voxels at the center of the object while still keeping them mannaged for use in constraints. However, it was also a very prominent problem when working with spring systems at first. This was because I initially had many of the calculations being run continuously in the Update Function and it was leading to performance issues . This book had the suggestion that there should be flags to decide

when the code should run and otherwise they should be left unused. This eventually led to me reconfiguring the code to work in the OnCollisionEnter function, which allows the calculations to only take place when needed.

system design and configuration



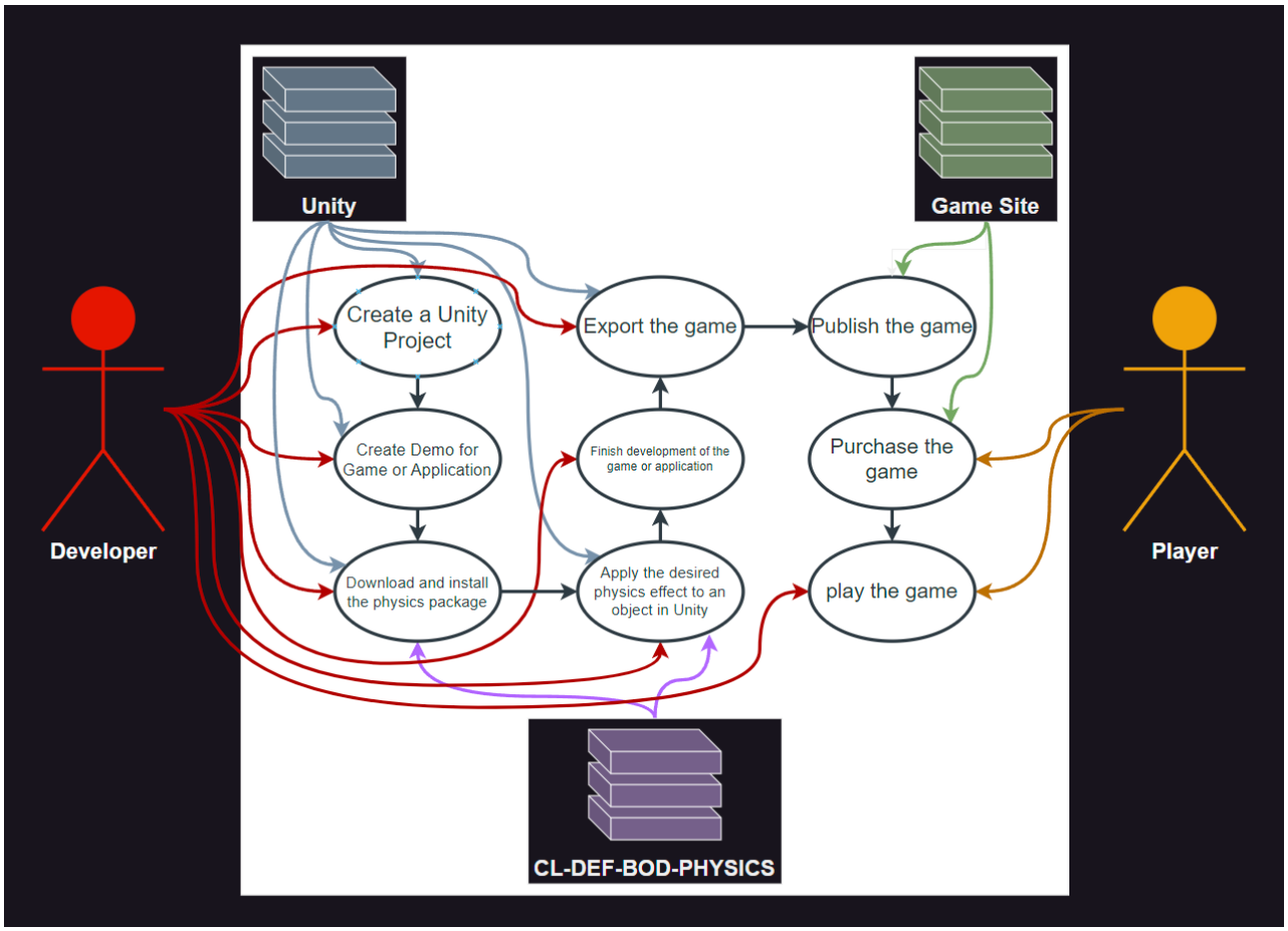
When Working on the creation of this Deformable body physics system there were many features which I knew would be necessary in order to make this system both functional, and usable within Unity. There would be a need to implement some variety of mesh detection, regardless of whether the final version implemented the 3D-Array Mass, or the Mass-Spring System. This would need to be used in order to implement a dynamic mesh movement system which would make a deformable body physics system possible. This would also create a need for manual control over the mesh data to allow for accurate deformation at the correct positions. In order to do this access to the vertices of an object mesh in Unity is required. In order to learn how to do this, knowledge on the creation process is required. To acquire this knowledge, a Manual mesh creation system will need to be created. There will need to be a way to visually display this mesh to the screen. We will also need to detect the bounds of the object in order to have a working 3D-Array mass system as this is required in order to correctly voxelize the object into the 3D grid system. This voxelization will allow for the creation of a crumble physics feature. All of these must be in place in order to allow for the creation of both a 3D-Array Mass, and a Mass-Spring System within Unity.

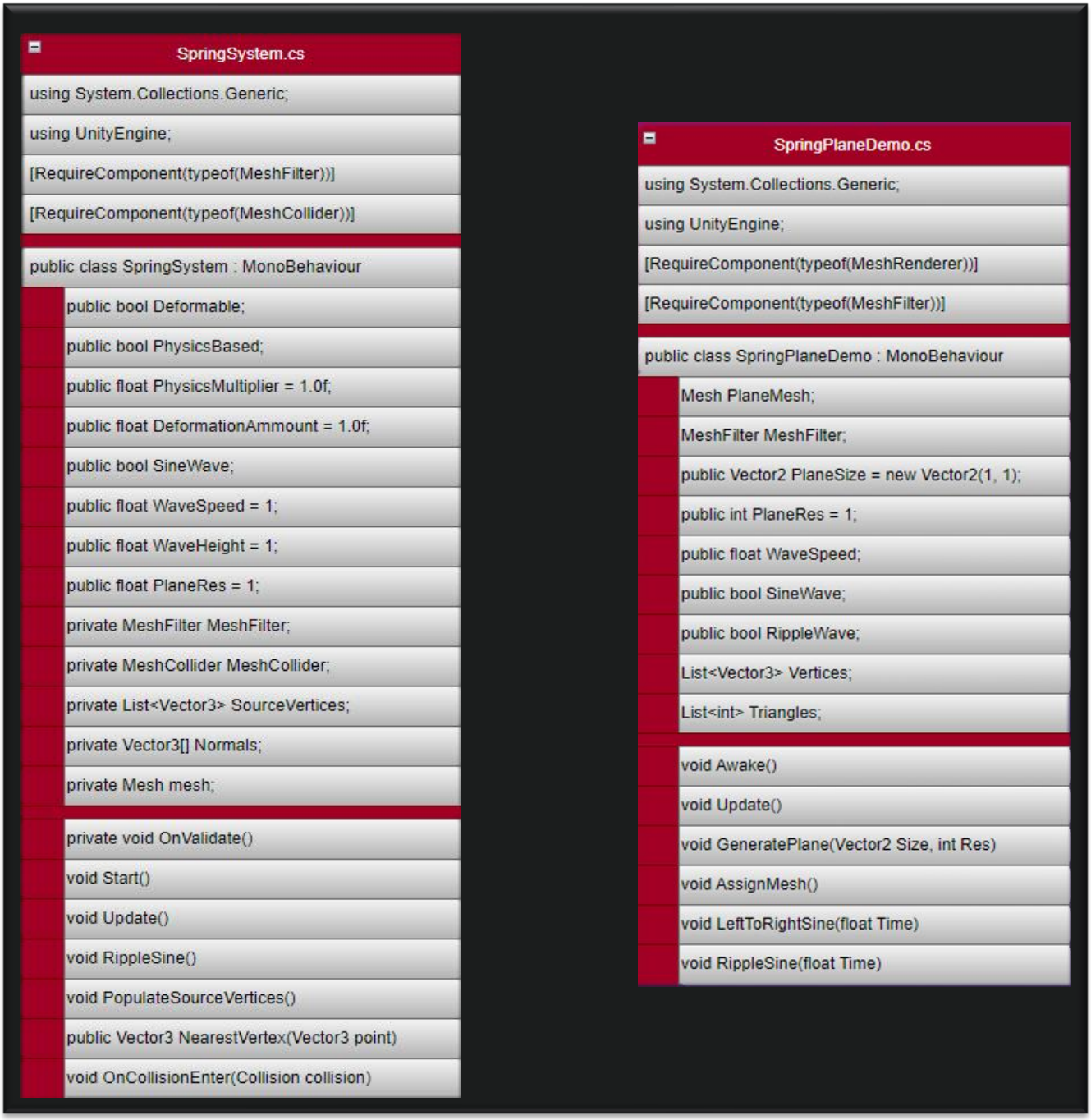
With all of this known, this led to the initial features associated with release one (1) of the final deformable-body physics system. This first release allows for the creation of both a 3D-Array Mass and a Mass-Spring System within Unity for the use of developers. Unlike other available systems, my system allows for automatic mesh detection. It allows for manual mesh manipulation through the use of sine wave functions. Mesh data can be manually created and allows for access to

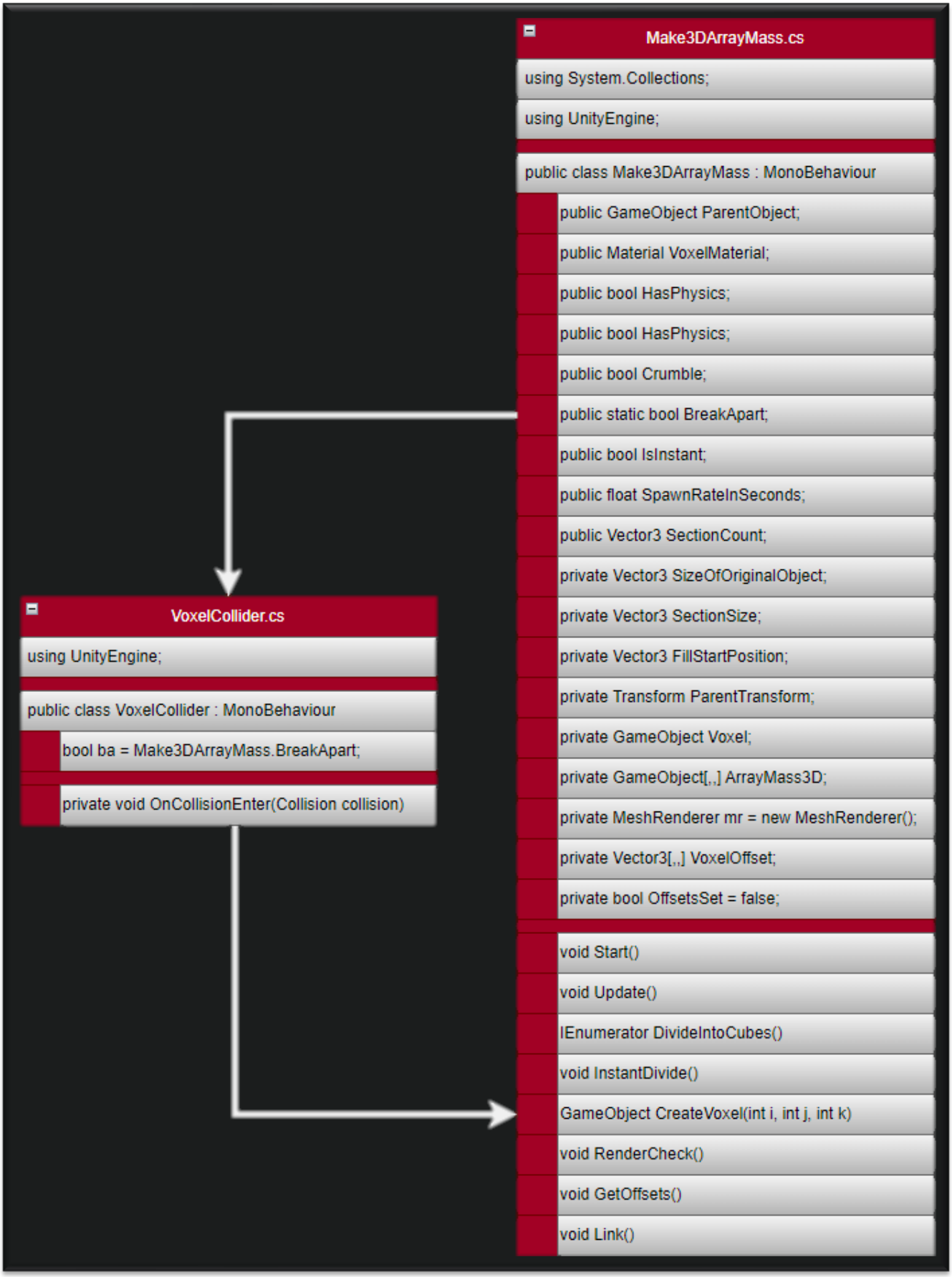
the vertices within the mesh. Object size and bounds can be automatically detected for use in a 3D - Array Mass. This can then be used to voxelize the object, which can then break apart using the crumble feature. The objects can also be visually displayed to the screen through the use of Unity's Mesh Renderer.

Even with this there is still room for improvement and features which could be added. Such as support for the physics-based deformation to take the direction the colliding object is travelling, the material of the object being hit, and the inclusion of volumetric deformation. Currently there is only support for smaller deformations. In order to allow for larger deformations there needs to be a method to get neighbouring vertices to the vertex which has been hit. This would allow for smoother deformation and dents. The system also currently struggles with lower resolution objects as there are not enough polygons within the mesh to be manipulated. This could be fixed given more time through the implementation of a mesh subdivision system which would increase the resolution of the objects.

The process of using the package is very simple. The developer will create a Unity project, they will create a demo for the game or application. They can then follow the steps provided in the User Guide in order to download and install the physics system. They can then apply the desired effect to the objects within unity. The developer can then finish the game and export the game to be published on a game seller site such as steam or Gog. The player can then purchase and play the game in order to experience the implementation of the Deformable Body Physics System.




SpringSystem.cs
using System.Collections.Generic;
using UnityEngine;
[RequireComponent(typeof(MeshFilter))]
[RequireComponent(typeof(MeshCollider))]
public class SpringSystem : MonoBehaviour
public bool Deformable;
public bool PhysicsBased;
public float PhysicsMultiplier = 1.0f;
public float DeformationAmmount = 1.0f;
public bool SineWave;
public float WaveSpeed = 1;
public float WaveHeight = 1;
public float PlaneRes = 1;
private MeshFilter MeshFilter;
private MeshCollider MeshCollider;
private List<Vector3> SourceVertices;
private Vector3[] Normals;
private Mesh mesh;
private void OnValidate()
void Start()
void Update()
void RippleSine()
void PopulateSourceVertices()
public Vector3 NearestVertex(Vector3 point)
void OnCollisionEnter(Collision collision)
SpringPlaneDemo.cs
using System.Collections.Generic;
using UnityEngine;
[RequireComponent(typeof(MeshRenderer))]
[RequireComponent(typeof(MeshFilter))]
public class SpringPlaneDemo : MonoBehaviour
Mesh PlaneMesh;
MeshFilter MeshFilter;
public Vector2 PlaneSize = new Vector2(1, 1);
public int PlaneRes = 1;
public float WaveSpeed;
public bool SineWave;
public bool RippleWave;
List<Vector3> Vertices;
List<int> Triangles;
void Awake()
void Update()
void GeneratePlane(Vector2 Size, int Res)
void AssignMesh()
void LeftToRightSine(float Time)
void RippleSine(float Time)


Make3DArrayMass.cs
using System.Collections;
using UnityEngine;
public class Make3DArrayMass : MonoBehaviour
public GameObject ParentObject;
public Material VoxelMaterial;
public bool HasPhysics;
public bool HasPhysics;
public bool Crumble;
public static bool BreakApart;
public bool IsInstant;
public float SpawnRateInSeconds;
public Vector3 SectionCount;
private Vector3 SizeOfOriginalObject;
private Vector3 SectionSize;
private Vector3 FillStartPosition;
private Transform ParentTransform;
private GameObject Voxel;
private GameObject[,,] ArrayMass3D;
private MeshRenderer mr = new MeshRenderer();
private Vector3[,,] VoxelOffset;
private bool OffsetsSet = false;
void Start()
void Update()
IEnumerator DivideIntoCubes()
void InstantDivide()
GameObject CreateVoxel(int i, int j, int k)
void RenderCheck()
void GetOffsets()
void Link()
VoxelCollider.cs
using UnityEngine;
public class VoxelCollider : MonoBehaviour
bool ba = Make3DArrayMass.BreakApart;
private void OnCollisionEnter(Collision collision)

Implementation & TESTING

There were two (2) primary methods which were considered in order to implement this physics system into a Unity environment. The first of these which will be discussed, is the use of a three-dimensional array mass system in order to simulate volumetric deformation, and the second one which will discussed is the idea to use a mass-spring system to simulate planar deformation. While both of these systems have their own merits and uses, this section will aim to explain the thought process and reasoning behind chosing one over the other and the challenges and benefits associated with each.

Three-Dimensional Array Mass

The three-dimensional array mass was the first system which I had considered as a potential solution to solve this problem. The initial reasoning for chosing this method was due to the large amount of information and research available on it. This large wealth of knowledge and access to study material meant that this was the system and method of which I had the greatest understanding. A three-dimensional array mass system is arguably one of the simplest and most common ways of simulating a volumetric deformation system. Volumetric deformation.

FUNCTIONALITY

The basic functionality of a three-dimensional array mass is very simple in theory. It is a computational model that takes a solid object and represents it as a grid-like structure consisting of smaller interconnecting masses. Each mass represents a small portion of the overall object, and it is the interaction between these masses that results in the simulation of a volumetric deformation effect. In the case of this project it is to be used within the context of a game development project so this means that the masses which it will be broken down into voxels. These voxels will then interact with each other through some variety of connections, such as springs, or in this case, constraints. This connectivity is what will ultimately allow an exchange of information between adjacent voxels and regions.

With the array mass set up we can now begin to process any extrernal forces and collisions which are detected. External forces which are applied to certain voxels or masses within the array system can then be used in order to induce deformations in the overall structure. There are manny different external systems which can cause this, such as, gravity, collisions, or even user defined inputs such as pulling or pushing specific areas or voxels within the mass. As these forces act upon indevidual voxels or masses effects can then be transmitted to neighbouring voxels through the use of the connections which have already been established in order to propogate the interaction throughout the entire system, this means we would end up with localised deformations which spread over time due to inter-voxel communication.

The actual calculations for these deformations must be performed throughout every step of the process. These calculations are based around the corresponding physics equations associated with the required type of deformation, e.g. planar deformation equations used on flat surfaces, edge deformation equations used for edges and so on. These calculations determine how much each voxel must move from its original position due to internal stress and strain caused by the external forces applied.

Once all connections have been made and the calculations have been performed, we must then focus on the visualisation and rendering of these deformations. In order to observe, analyse, and ultimately make use of these systems we must employ rendering techniques to disay these changes

in shape and appearance over time accurately. It is for this case that Unity's in-built Mesh Filter and Mesh Renderer components can be used effectively. The Mesh Filter allows for the creation of a dynamically updating object mesh which can be reflected in the Mesh Collider component to continue to allow for accurate collisions. These can then be displayed within unity by passing that data to the Mesh Renderer component.

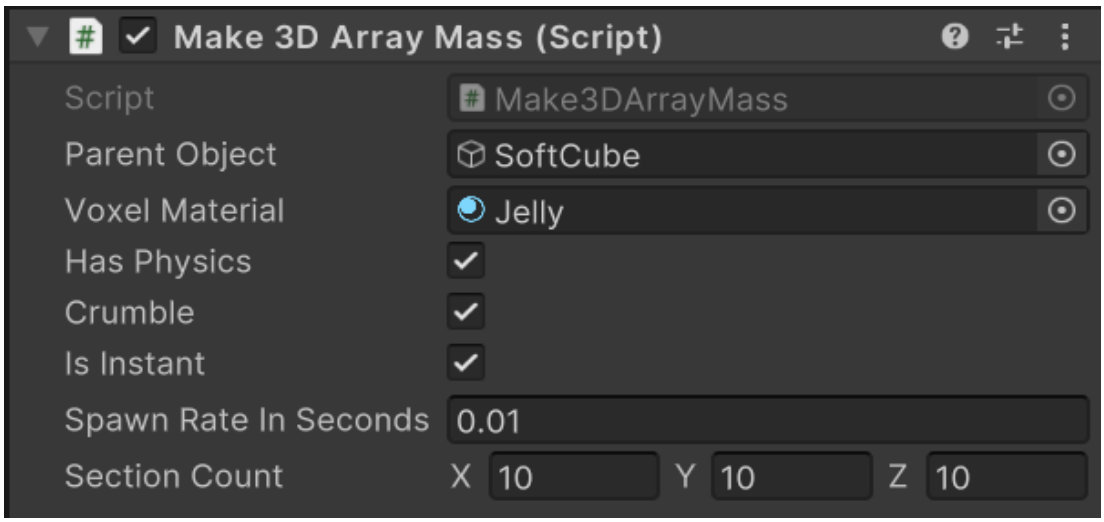
Overall, a 3D array mass system for simulating volumetric deformation allows researchers or designers to study and understand how objects or materials behave under different conditions. It can be used in various fields like computer graphics, engineering (structural analysis), medical simulations (organ deformations), or material science research.

CODE EXPLAINATION

- *Make3DArrayMass.cs*

One of the key features of defining variables within Unity is the definition of whether or not a variable is assigned as being public or private. Aside from the obvious effects this will have on the variables usage within the code, it is also important to note that within Unity scripting, any variables which are defined as public, will also be visable within the inspector window of objects which the script is attached to. It is this system which will allow for user input into the code. This will allow for the creation of a simple UI for the use of developers. With the variables defined we can now see these changes reflected in the inspector window within the unity editor. All public defined variables can be seen here, and can be adjusted to the developers preference.

```
12      //Public
13          public GameObject ParentObject; //Template game object to be replaced by soft-body
14          public Material VoxelMaterial; //defines the material of each voxel
15          public bool HasPhysics; //Defines whether the Voxels have a Rigidbody applied
16          public bool Crumble;
17          public static bool BreakApart; //Defines if the object shoud break into voxels
18          public bool IsInstant; //Defines which creation method to use
19          public float SpawnRateInSeconds; //Defines the spawn rate when not in instant mode
20          public Vector3 SectionCount; //Stores the total amount of voxels
21
22      //Private
23          private Vector3 SizeOfOriginalObject; //Expected size of total array in physical
24          private Vector3 SectionSize; //Voxel size
25          private Vector3 FillStartPosition; //Starting oint of array
26          private Transform ParentTransform; //Transform of parent object
27          private GameObject Voxel; //This will store the voxels used in the array mass
28          private GameObject[,,] ArrayMass3D; //This is a 3D array which will be used to store and manage the voxels
29          private MeshRenderer mr = new MeshRenderer(); // Used to store the MeshRenderer of the Voxels
30          private Vector3[,,] VoxelOffset; // Stores the voxel offsets
31          private bool OffsetsSet = false; // boolean used to check if offsets are set
```



When defining these there were several considerations to be made, such as what would need to be adjusted by the user, what would be convenient to have control over, and would there be any fringe cases to take into account. It is for these reasons that this layout and distribution was decided on. The first public variable is the ability to define the parent object which the changes will be based on and applied to. Later in the code this will be assigned to the object the script is attached to as a default, however the abiliy to apply it to a different object than the one that is being affected allows for greater versatility in the usage of the feature. By applying it to a different object than the one affected it would allow a user to destroy the initial object after it becomes voxelised in order to save on memory or even just clean up a scene. Next the user is asked to define a voxel material. Similar to the parent object this will default to the same material as the parent object, however, it may still be useful for a developer to be able to change this for testing purposes. These are then followed by four (4) boolean values used to define the primary behaviours of the script. “HasPhysics” is used to define whether or not the subsiquent voxels which get created should have a rigidbody component attached to them or not. Having this component will allow them to behave with gravity and mass in consideration. “Crumble” is used to define if the created voxels should be linked together with the use of predefined constraints, or if they should fall apart to create the crumble effect. This will also link into the application of the second class and script “VoxelCollider.cs”, however, this will be explained later. “BreakAppart” is the actual functional use version of “Crumble” as it is the one used throughout the code after being set to equal the same value as “Crumble”. The reason for crumble still existing despite this is because due to the way Unity scripting functions, static variables can not be seen within the inspector, so “Crumble” is needed in order to allow a user to adjust this value. “BreakApart” is required to be static in order to interact with “VoxelCollider.cs”. “IsInstant” is used to define if all voxels should be spawned in at once or if they should spawn onee by one. “SpawnRateInSeconds” is used to define the time gap between each voxel spawn when not in “InstantDevide” mode. Lastly for the public variables, is the “SectionCount”. This is defined as a Vector3, which will allow the user to define the number of sections which the object will be divided into on the x, y, and z axes. This is important as some objects may be taller than they are wide, or deeper than they are tall, and so on. This variety makes this a key feature.

With all variables initialized, we must then move on to the “Start” function. The "Start" function is a special method in Unity that is automatically called when the object with this script attached to it starts running. It is commonly used for initialization and setup tasks. In the case of Unity, this applies to whenever an object becomes active, either when the developer enters play mode and sets it to do so, or whenever it becomes active in the final published game.The first stages of this start sequence are to define the default values explained before. This is done from lines 40 – 50, with the first thing being to set the “BreakApart” boolean value to equal the value of “Crumble” which is input by the user.This is then followed by two (2) if statements designed to set the default values of the ParentObject and VoxelMaterial variables. These if statements simply check if the current values for either one are null, and if they are, then set them to equal the object the script is attatched to. For the ParentObject this is done by having it equal to gameobject. This is because by default in Unity C# script, “gameobject” is used as a reference to the object the script is attached to. For the VoxelMaterial we have it equal to the The parent objects “GetComponent<Renderer>().material”. This will return whatever matrerial is attached to the defined parent object.

```
Unity Message | 0 references
void Start()
{
    BreakApart = Crumble;

    if (ParentObject == null) //if nothing is attatched then use the object the script is linked to
    {
        ParentObject = gameObject;
    }

    if (VoxelMaterial == null) //if nothing is attatched then use the material of the object the script is linked to
    {
        VoxelMaterial = ParentObject.GetComponent<Renderer>().material;
    }

    //Initialize The 3DArray
    int XVal = (int)SectionCount.x;
    int YVal = (int)SectionCount.y;
    int ZVal = (int)SectionCount.z;
    ArrayMass3D = new GameObject[XVal, YVal, ZVal];
    VoxelOffset = new Vector3[XVal, YVal, ZVal];

    //Define the section size
    SizeOfOriginalObject = ParentObject.transform.lossyScale;
    SectionSize = new Vector3(
        SizeOfOriginalObject.x / SectionCount.x,
        SizeOfOriginalObject.y / SectionCount.y,
        SizeOfOriginalObject.z / SectionCount.z
        );

    //Define the array start position
    FillStartPosition = ParentObject.transform.TransformPoint(new Vector3(-0.5f, -0.5f, -0.5f))
                        + ParentObject.transform.TransformDirection(new Vector3(SectionSize.x, SectionSize.y, SectionSize.z) / 2.0f);

    //Get the current parent transform
    ParentTransform = new GameObject(ParentObject.name + "CubeParent").transform;

    //Define the render method
    if (IsInstant)
    {
        InstantDivide();
    }
    else
    {
        StartCoroutine(DivideIntoCubes());
    }

    if (!BreakApart)
    {
        RenderCheck();
    }
}
```

Following this is then the definition and creation of the two (2) 3D arrays which are used to identify the voxels within the code. To do this the x, y, and z values are taken from the section count vector3 variable and are casted to int values. This is done because the dimentions of a 3D array must be defined by three (3) int values. These x, y, and z values are then passed into both the “ArrayMass3D” “GameObject” array, and the “VoxelOffset” Vector3 array. This is then followed by the deffinition for the SectionSize which is the declaration of the size of the voxels. This is found by dividing the section count by the total size of the parent object on each axis. The start position for the voxel fill is then set and the current part transform is then gotten for later use. Next there is a check to see if the user has selected the division to be instant or not. In order to not stall the entire code I make use of a coroutine in order to have the slow divide function run simultaneously to the rest of the code. Finally there is an iff statement to check if the center voxels of the object need to be de-rendered. This was one of the many things which were used to try and save performance.

After the start function runs, then the update function will begin running. The "Update" function is another special method in Unity that is called once per frame. It is commonly used for continuous actions or updates that need to be performed during gameplay. It can be used to check for user input, update object positions, perform calculations, and handle other real-time changes. Any code placed within the "Update" function will execute repeatedly until the game ends or the script is disabled or the game object it is attatched to is destroyed. In this case it is used to continuously Set the offset data and uphold the constrants between each of the voxels through the use of the “Link” function. These are both set to only be used if the object is not set to “Crumble”.

```
94      void Update()
95      {
96          if (!BreakApart)
97          {
98              GetOffsets();
99              Link();
100         }
101     }
```

In the start function section it was discussed that there was a slow divide function which was run as a coroutine adjacent to the rest of the code excecution. This is done for the excecution of the “DivideIntoCubes” function.

```
1 reference
109 IEnumerator DivideIntoCubes() //This will spawn each voxel one-by-one for demonstration purposes
110 {
111     for (int i = 0; i < SectionCount.x; i++) //X
112     {
113         for (int j = 0; j < SectionCount.y; j++) //Y
114         {
115             for (int k = 0; k < SectionCount.z; k++) //Z
116             {
117                 //create Voxel
118                 Voxel = CreateVoxel(i, j, k);
119
120                 ArrayMass3D[i, j, k] = Voxel; //Add Voxel to 3D array mass
121
122                 //Wait for set spawn rate
123                 yield return new WaitForSeconds(SpawnRateInSeconds);
124             } //Z
125         } //Y
126     }// X
127     ParentObject.transform.localScale = new Vector3(0, 0, 0);
128 }
```

The DivideIntoCubes function is defined as an IEnumerator. An IEnumerator is used in Unity to create a coroutine, which allows for the execution of code over multiple frames. Coroutines are useful when you want to perform actions that need to be spread out or delayed over time. In this specific case, the "DivideIntoCubes" method is an IEnumerator because it is being used as a coroutine to spawn each voxel one-by-one over time. By implementing DivideIntoCubes as an IEnumerator, it allows other parts of the code or scripts to start and stop this process easily and gives control over when and how often new voxels are spawned.

The main body of the function consists of a triple nested for loop. This is done as it is the easiest way to iterate through the 3D array. This is because it allows for the assignment of an x, y, and z values. Within the loops The method CreateVoxel(i, j, k) is called to create a new voxel at the current iteration position (i, j, k). which corresponds to the current (x, y, z) values. The newly created voxel GameObject is assigned to the Voxel variable. The Voxel is then added to the ArrayMass3D 3D array at its corresponding index (i, j, k). As this function is the slow divide function After each voxel creation and addition to the array, we yield return a WaitForSeconds with a user specified duration of SpawnRateInSeconds. This means that after creating one voxel, we pause execution of this coroutine for that duration before continuing with the next iteration. Once all iterations are complete and all voxels have been created and added to ArrayMass3D, we modify the scale of ParentObject (the original object) by setting it to Vector3(0,0,0). This is done is a simple method of cleaning up the parent object without destroying it.

```
1 reference
void InstantDivide() //This version will be used in engine and will instantly load all voxels
{
    for (int i = 0; i < SectionCount.x; i++) //X
    {
        for (int j = 0; j < SectionCount.y; j++) //Y
        {
            for (int k = 0; k < SectionCount.z; k++) //Z
            {
                //Create voxel
                Voxel = CreateVoxel(i, j, k);

                ArrayMass3D[i, j, k] = Voxel; //Add Voxel to 3D array mass
            } //Z
        } //Y
    }// X
    ParentObject.transform.localScale = new Vector3(0, 0, 0);
}
```

The InstantDivide function is almost identical to the DivideIntoCubes function with the minor changes of the function being a void instead of an IEnumerator. This is because it does not need to be run as a coroutine. This also leads to the removal of the SpawnRateInSeconds method.

```
2 references
GameObject CreateVoxel(int i, int j, int k) //Called When Creating a new Voxel
{
    GameObject Voxel;

    Voxel = GameObject.CreatePrimitive(PrimitiveType.Cube);

    //Define the voxel
    Voxel.transform.localScale = SectionSize;
    Voxel.transform.position = FillStartPosition +
        ParentObject.transform.TransformDirection(new Vector3((SectionSize.x) * i, (SectionSize.y) * j, (SectionSize.z) * k));
    Voxel.transform.rotation = ParentObject.transform.rotation;
    Voxel.gameObject.tag = "Voxel";

    // Define the parent and material
    Voxel.transform.SetParent(ParentTransform);
    Voxel.GetComponent<MeshRenderer>().material = VoxelMaterial;

    //Apply Rigidbody
    if (HasPhysics)
    {
        Voxel.AddComponent<VoxelCollider>();
        Voxel.AddComponent<Rigidbody>();
    }


    return Voxel;
}
```

Both divide functions make reference to a function called CreateVoxel. This is used for the full creation and definition of each voxel The method GameObject CreateVoxel(int i, int j, int k) is defined with parameters i, j, and k representing the voxel's x, y, and z position within 3D space. Firstly within the method is the declaration of a GameObject variable named Voxel. Using GameObject.CreatePrimitive(PrimitiveType.Cube), a new cube primitive game object is created and assigned to Voxel. The size of the voxel (scale) is set using SectionSize which represents the desired size of each voxel. The position of the voxel is determined by adding FillStartPosition (the starting point of array filling) to a transformed vector based on i,j,k indices multiplied by their respective section sizes. This ensures that voxels are positioned correctly within the array relative to ParentObject's transform. The rotation of the voxel is set to match ParentObject's rotation. A tag "Voxel" is assigned to this newly created Voxel game object using .gameObject.tag. This will allow for reference in other scripts if it is needed by the developer. he parent transform for this voxel is set using .SetParent(ParentTransform). This means that all voxels will have ParentTransform as their parent in order to keep them organized under one container game object. Finally, if HasPhysics flag is true, A script called VoxelCollider (handling collisions specific to voxels) and a Rigidbody component are added as components to this Voxel game object. After completing these steps for creating and configuring a single voxel, it returns this GameObject instance named 'Voxel'.

```
1 reference
void RenderCheck() // used to only render the outside faces of an object
{
    for (int i = 0; i < SectionCount.x; i++) //X
    {
        for (int j = 0; j < SectionCount.y; j++) //Y
        {
            for (int k = 0; k < SectionCount.z; k++) //Z
            {
                mr = ArrayMass3D[i, j, k].GetComponent<MeshRenderer>();
                //Check if the voxel is on the outside face of the object
                if (k != 0 && k != SectionCount.z-1 && j != 0 && j !=SectionCount.y-1 && i != 0 && i != SectionCount.x-1)
                {
                    mr.enabled = false; // if not then dont render
                }
            } //Z
        } //Y
    }// X
}
```

As explained, the start function ends with a render check which is used as a method to help improve performance by de-rendering any voxels which are not on the surface of the object. The method void RenderCheck() is defined as a void. Within the method, another nested for loop structure is used to iterate over each voxel position in the 3D array based on the SectionCount values. For each iteration, we retrieve the MeshRenderer component of the voxel from ArrayMass3D using ArrayMass3D[i, j, k].GetComponent<MeshRenderer>() and assign it to mr (MeshRenderer variable). Then there's an if statement that checks if a voxel is located on any of the outside faces of the object by comparing its indices (i, j, k) with boundaries determined by SectionCount values minus one. If any index (k, j, i) is neither at index 0 or at the maximum boundary (SectionCount.z-1 or SectionCount.y-1 or SectionCount.x-1), this means that voxel is not on an outer face. only voxels positioned at either end along all three axes will be considered part of outer faces while others are considered internal voxels within those bounds. If a voxel isn't on an outside face according to these conditions, its MeshRenderer component's enabled property is set to false so it won't render visually.

```
1 reference
void GetOffsets() // Sets the offset data for the Voxels
{
    if (!OffsetsSet)
    {
        for (int i = 0; i < SectionCount.x; i++) //X
        {
            for (int j = 0; j < SectionCount.y; j++) //Y
            {
                for (int k = 0; k < SectionCount.z; k++) //Z
                {
                    if (i > 0)
                        VoxelOffset[i, j, k] = ArrayMass3D[i, j, k].transform.position - ArrayMass3D[i - 1, j, k].transform.position;
                    if (j > 0)
                        VoxelOffset[i, j, k] = ArrayMass3D[i, j, k].transform.position - ArrayMass3D[i, j - 1, k].transform.position;
                    if (k > 0)
                        VoxelOffset[i, j, k] = ArrayMass3D[i, j, k].transform.position - ArrayMass3D[i, j, k - 1].transform.position;
                } //Z
            } //Y
        }// X

        OffsetsSet = true;
        Debug.Log("Offsets Set");
    }
}
```

As explained, one method for transfering information between voxels is through the use of constraints. The GetOffsets function is used as a means to establish a baseline for these constraints. Within the method there's an if statement condition checking if OffsetsSet is false. This ensures that offsets are only set once and not repeatedly. If OffsetsSet is false, a nested for loop structure is once again used to iterate over each voxel position in the 3D array based on the SectionCount values. For each iteration, there are three conditions (if statements) that check if i, j, or k indices are greater than zero. If i > 0, VoxelOffset at position [i, j, k] is calculated by subtracting ArrayMass3D[i-1, j, k].transform.position from ArrayMass3D[i, j, k].transform.position. this calculates the offset between current voxel and its neighbor on the previous X-axis index (i-1). If j > 0, VoxelOffset at position [i, j, k] is calculated by subtracting ArrayMass3D[i, j-1, k].transform.position from ArrayMass3D[i, j, k].transform.position. If k > 0: VoxelOffset at position [i, j, k] is calculated by subtracting ArrayMass3D[i, j, k-1].transform.position from ArrayMass3D[i, j, k].transform.position.

After setting all necessary offsets for voxels in their respective positions within SectionCount bounds, OffsetsSet flag variable becomes true to indicate that offsets have been set.

```
220     void Link() // Set the Voxels to follow each other every frame
221     {
222         if (OffsetsSet)
223         {
224             for (int i = 0; i < SectionCount.x; i++) //X
225             {
226                 for (int j = 0; j < SectionCount.y; j++) //Y
227                 {
228                     for (int k = 0; k < SectionCount.z; k++) //Z
229                     {
230                         if (i > 0)
231                             ArrayMass3D[i, j, k].transform.position = ArrayMass3D[i - 1, j, k].transform.position + VoxelOffset[i, j, k];
232                         if (j > 0)
233                             ArrayMass3D[i, j, k].transform.position = ArrayMass3D[i, j - 1, k].transform.position + VoxelOffset[i, j, k];
234                         if (k > 0)
235                             ArrayMass3D[i, j, k].transform.position = ArrayMass3D[i, j, k - 1].transform.position + VoxelOffset[i, j, k];
236                     } //Z
237                 } //Y
238             }// X
239         }
240     }
```

With the offsets set we can now begin to link the voxels together. This is done thtough the use of the Link Function. The function first checks the value of the OffsetsSet flag is true. If it is then we once again iterate through a nested for loop We then set use three (3) if statements in an identical layout to the GetOffsets function. In this case they are used to assign the current transform position to equal to itself plus the associated offset.

- *VoxelCollider.cs*

```
    Unity Script | 1 reference
3   public class VoxelCollider : MonoBehaviour
4   {
5       bool ba = Make3DArrayMass.BreakApart;
6
        Unity Message | 0 references
7       private void OnCollisionEnter(Collision collision)
8       {
9           if (collision.gameObject.tag == "Voxel")
10          {
11              GameObject Voxel = collision.gameObject;
12
13              if (!ba)
14                  Physics.IgnoreCollision(Voxel.GetComponent<Collider>(), GetComponent<Collider>());
15          }
16      }
17  }
```

The second script referenced for this system is the “VoxelCollider.cs” script. This is necessary as another way to save on performance which is the 3D array masses most prominent issue. It makes reference to the static BreakApart bool which is created in the “Make3DArrayMass.cs” script. If this true then it will disable all collisions between objects which are tagged with “Voxel”. This script is then attatched to all voxels.

It was at this point that work was stopped on the implementation of a 3D array mass system. Throughout the creation of this system there were many compounding problems related to the performance of the system overall. Even within an empty scene where only the deformable object and a colliding object exists, there were still detrimental performance issues. There were many attempts made to improve upon this. Such as de-rendering all voxels that are not visible and disabling collisions between voxels. However, the issues were more fundamental to Unity itself and could not be fixed through code alone. In order to have the 3D-Array Mass reach a high enough fidelity to simulate volumetric deformation, we are required to spawn several thousand objects in unity. It is the handling of these objects on the side of Unity that leads to the performance issues. It is for this reason that it was decided to move to a Mass-Spring System.

Mass-Spring System

FUNCTIONALITY

A mass-spring system is a computational model used to simulate the behaviour of objects or materials by representing them as masses connected by springs. It is commonly employed in physics simulations, computer graphics, and animation to mimic the dynamics and deformation of structures. The basic functionality of a mass-spring system is that the objects or materials being simulated are represented as individual point masses. Each mass represents a small portion of the object's volume or an element within a material. The connectivity between masses is established using springs that link pairs (or groups) of neighboring masses together. These springs function as elastic elements, storing and releasing energy when deformed under applied forces. External forces such as gravity, user inputs like pushing or pulling on specific points, or other environmental factors act upon certain masses within the system to induce motion and deformations. As external forces act on individual masses, they transmit these effects through the connected springs to influence neighboring masses' behavior. This interaction propagates throughout the entire system and causes dynamic changes in positions and velocities over time due to inter-mass communication. By tracking displacement values for each mass from its original position at any given simulation time step, it becomes possible to calculate various measures related to deformation—such as strain or stress—in order to analyze structural integrity or material behavior. Similar to the implimentation of the 3D-Array Mass system using voxels as a guide, in this case, we will instead make use of mesh vertices as a mase for the deformation. These deformations will also require a change in the specific physics deformation calculations being used. This is because unlike the 3D-Array Mass which uses volumetric deformation, this method instead uses planar deformation as the primary form of deformation. This requires a different set of equations and calculations in order to correctly perform them.

CODE EXPLAINATION

- *SpringPlaneDemo.cs*

When considering the mass-spring system as a potential option initially for ways in which to implement the deformable body system, the idea of using the already existing mesh data of Unity objects as a starting point and modifying its functionallity in order to make it work as a mass spring system in addition to its standard rendering functionality. There would be many advantages to a system like this. It removes the need to create a map of the object mesh as it already exists within the mesh filter data. It would allow for implementation of fully accurate and concave collision detection as it would allow for the use of the mesh collider system within Unity to facilitate one hudered percent (100%) accurate collision detection and object interaction within Unity scenes. Using the already existing mesh vertices as the masses for the spring mass system also eliminates the need to spawn and manage thousands of objects, which was the primary cause of performance issues before. In order to make use of the Unity mesh system as a base for this method, an understanding of how Unity handles meshes as well as how to make a mesh such as this dynamic and changeable through code. As a means to this end the first step to achieving this goal is to create a script which can create its own mesh from scratch, and then dynamically move the vertices around in 3D space at run time. It must also show accurate collisions as this will be integral to the deformation on collision functionallity. This was the purpose of the creation of the “SpringPlaneDemo.cs” script. It was a valuable testing ground which eventually led to the full creation of the final system. It also established a baseline for the functions and methods which would later be elaborated on in the final version of the ”SpringSystem.cs” script, which contains all of the desired features and functionality which was outlined in the introduction.

```
4     [RequireComponent(typeof(MeshRenderer))]
5     [RequireComponent(typeof(MeshFilter))]
6
      Unity Script (1 asset reference) | 0 references
7     public class SpringPlaneDemo : MonoBehaviour
8     {
9         Mesh PlaneMesh;
10        MeshFilter MeshFilter;
11
12        public Vector2 PlaneSize = new Vector2(1, 1);
13        public int PlaneRes = 1;
14        public float WaveSpeed;
15        public bool SineWave;
16        public bool RippleWave;
17
18        List<Vector3> Vertices;
19        List<int> Triangles;
```

In order to ensure that the user does not try to use the script on an object that does not have the required components for it to function, Unity allows for the use of the [RequireComponent(typeof())] which will prevent the developer from entering the play mode without having the correct components attached to the object. This is important as this script is intended to be attached to an Empty object within Unity, which by default does not come with any components. As explained before the use of public variables within Unity scripting is to allow for the user to adjust these parameters within the inspector window. In this case the user is given the ability to adjust the size of the plane through the use of a vector2 variable. A vector2 is used here instead of a vector3 because a plane is being created which does not have any need for a height value as it is a flat object. The user can also decide the plane resolution. This is used to determine the density of the generated mesh. Higher resolution mesh data will be more resource intensive, but will also allow for a greater variety of effects. The speed of the sinewave can be adjusted as well as the type of wave. Sine wave will go from edge to edge and ripple wave will go from corner to corner.

```
      Unity Message | 0 references
22    void Awake()
23    {
24        PlaneMesh = new Mesh();
25        MeshFilter = GetComponent<MeshFilter>();
26        MeshFilter.mesh = PlaneMesh;
27    }
```

The Awake function is a special method in Unity that is automatically called when a script or object wakes up. It is executed before the Start function and can be used for initialization tasks. Within this method a new Mesh object named PlaneMesh is declared using the new Mesh() constructor. The GetComponent<MeshFilter>() function is called to retrieve the MeshFilter component attached to the current game object (the one with this script). It is this mesh filter which will allow for the creation of the dynamic mesh. The PlaneMesh mesh created earlier is then assigned to the mesh property of that retrieved MeshFilter component using MeshFilter.mesh = PlaneMesh.

```
Unity Message | 0 references
30  void Update()
31  {
32      PlaneRes = Mathf.Clamp(PlaneRes, 1, 50);
33
34      GeneratePlane(PlaneSize, PlaneRes);
35      if (SineWave)
36      {
37          LeftToRightSine(Time.timeSinceLevelLoad * WaveSpeed);
38      }
39      if (RippleWave)
40      {
41          RippleSine(Time.timeSinceLevelLoad * WaveSpeed);
42      }
43
44      AssignMesh();
45  }
```

As previously explained, the Update function is called once per frame. It is commonly used for continuous actions or updates that need to be performed during gameplay. In this case it is used to continuously regenerate the mesh with the changes applied by the sine wave. Within the method the first thing done is to clamp PlaneRes between one (1) and fifty (50) using Mathf.Clamp(). This ensures that the resolution of the plane remains within a valid range as, beyond these two (2) values the plane no longer performs in any different ways. Next, the method GeneratePlane(PlaneSize, PlaneRes) is called to generate or update the plane mesh based on provided size (PlaneSize) and resolution (PlaneRes). If the boolean variable SineWave is true, then the method LeftToRightSine(Time.timeSinceLevelLoad * WaveSpeed) is called. This suggests that a left-to-right sine wave effect will be applied to modify or animate vertices of the generated plane mesh over time. Similarly, if the boolean variable RippleWave is true, then another method named RippleSine(Time.timeSinceLevelLoad * WaveSpeed) will be called to create a ripple-like sine wave effect on vertices of generated plane mesh over time. The reason these flags are performed in the update function, instead of in the start or awake function is because this allows for the realtime changing between the two methods. Finally, after any modifications have been made to manipulate vertex positions based on wave effects or other factors, we call another function named AssignMesh() which applies these changes to update/render them onto actual Mesh used by this object.

```
1 reference
47  void GeneratePlane(Vector2 Size, int Res)
48  {
49      //Nodes
50      Vertices = new List<Vector3>();
51      float XPerStep = Size.x / Res;
52      float YPerStep = Size.y / Res;
53
54      for (int y = 0; y < Res + 1; y++)
55      {
56          for (int x = 0; x < Res + 1; x++)
57          {
58              Vertices.Add(new Vector3(x * XPerStep, 0, y * YPerStep));
59          }
60      }
61
62      //Springs
63      Triangles = new List<int>();
64      for (int row = 0; row < Res; row++)
65      {
66          for (int col = 0; col < Res; col++)
67          {
68              int i = (row * Res) + row + col;
69              Triangles.Add(i);
70              Triangles.Add(i + (Res) + 1);
71              Triangles.Add(i + (Res) + 2);
72
73              Triangles.Add(i);
74              Triangles.Add(i + (Res) + 2);
75              Triangles.Add(i + 1);
76          }
77      }
78  }
```

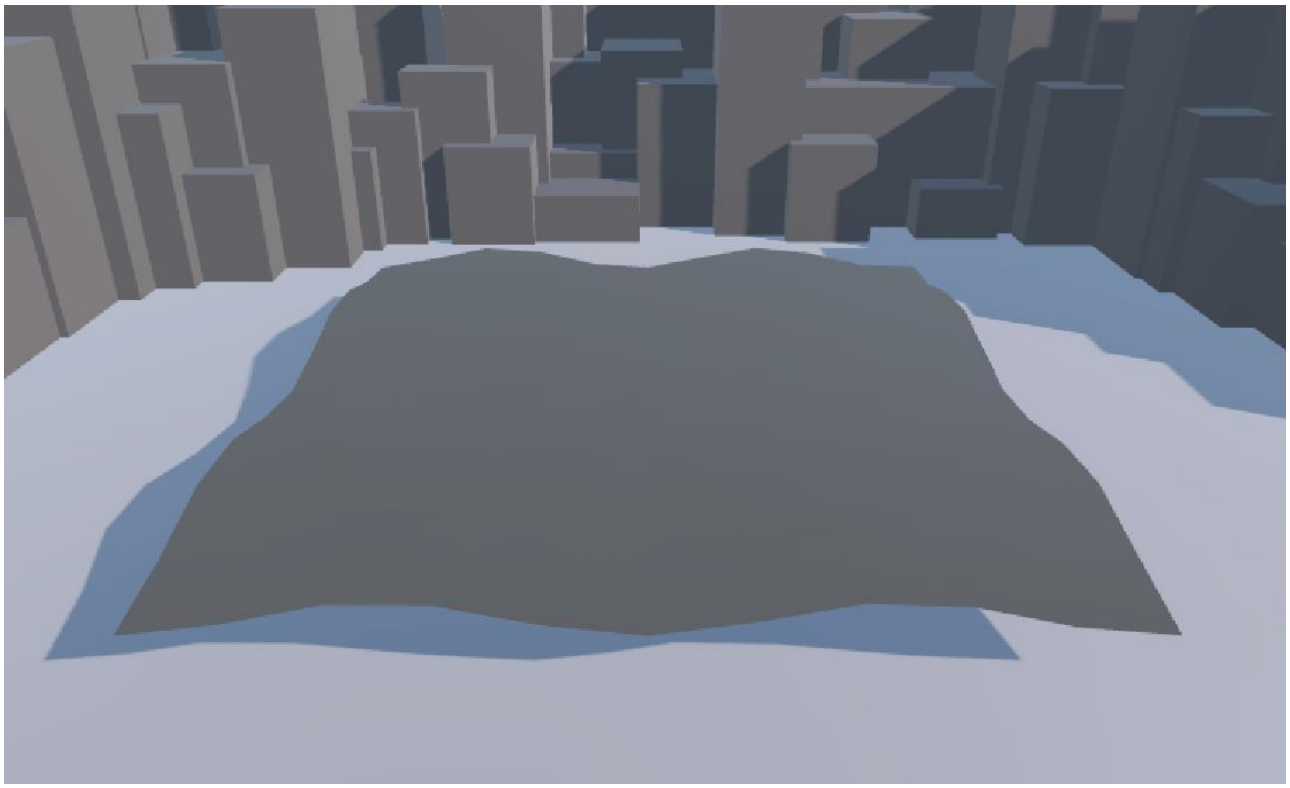
With the setup complete, we can now look at the actual planee generation itself. In Unity, a mesh is generated by creating a set of vertices and then creating tris by connecting three (3) of these vertices together. It is these triangles which will be used as a basis for the springs when the mesh is made dynamic. In order to create this mesh, the method void GeneratePlane(Vector2 Size, int Res) is defined. A list of Vector3 named Vertices is created to store the positions of each vertex in the plane. As explained previously, it is these vertices which will act as the collision detection and masses for the mass-spring system. Next two (2) variables XPerStep and YPerStep are calculated by dividing Size.x and Size.y respectively by Res (the resolution). These variables represent how much to increment along the x-axis and y-axis per step in order to evenly distribute vertices across the plane. Following this is the only two (2) uses of a nested for loop in the script which is used for the creation of the triangles and vertices. These nested loops iterate over each row (y) and column (x) from 0 up to Res+1. For each iteration, a new Vector3 is created with coordinates (x * XPerStep, 0, y * YPerStep), representing the position of a vertex on the plane. This newly created vector is then added to the Vertices list using Vertices.Add(). Another list named Triangles is created to store indices that define triangles in order to form faces on our mesh. Nested loops iterate over each row (row) and column (col) up to Res. For each iteration, an index i is calculated based on current row and column values using (row * Res) + row + col. Indices are added sequentially into Triangles according to existing vertices' positions such that they form two triangles for every grid square on our plane mesh. This creates a series of connected triangles forming multiple faces within our mesh structure.

```
1 reference
80  void AssignMesh()
81  {
82      PlaneMesh.Clear();
83      PlaneMesh.vertices = Vertices.ToArray();
84      PlaneMesh.triangles = Triangles.ToArray();
85  }
```

The purpose of this AssignMesh function/method is to update/render changes made to vertex positions and triangle indices onto the actual Mesh used by this game object (PlaneMesh). It clears any existing data from PlaneMesh using Clear(), then sets/assigns new values for vertices and triangles based on Vertices and Triangles lists respectively. By calling this function after modifying or updating mesh-related data, we can apply those changes visually by updating, rebuilding, and refreshing the mesh representation.

```
1 reference
87  void LeftToRightSine(float Time)
88  {
89      for (int i = 0; i < Vertices.Count; i++)
90      {
91          Vector3 Vertex = Vertices[i];
92          Vertex.y = Mathf.Sin(Time + Vertex.x);
93          Vertices[i] = Vertex;
94      }
95  }
96
    1 reference
97  void RippleSine(float Time)
98  {
99      for (int i = 0; i < Vertices.Count; i++)
100     {
101         Vector3 Vertex = Vertices[i];
102         Vertex.y = Mathf.Sin(Time + (Vertex.x + Vertex.z));
103         Vertices[i] = Vertex;
104     }
105 }
106 }
```

The purpose of these LeftToRightSine and RippleSine functions/methods is to apply a sine wave effect to the y-coordinates of vertices stored in the Vertices list. The LeftToRightSine function modifies vertex positions based on a left-to-right motion along x-axis by using Mathf.Sin(Time + Vertex.x.

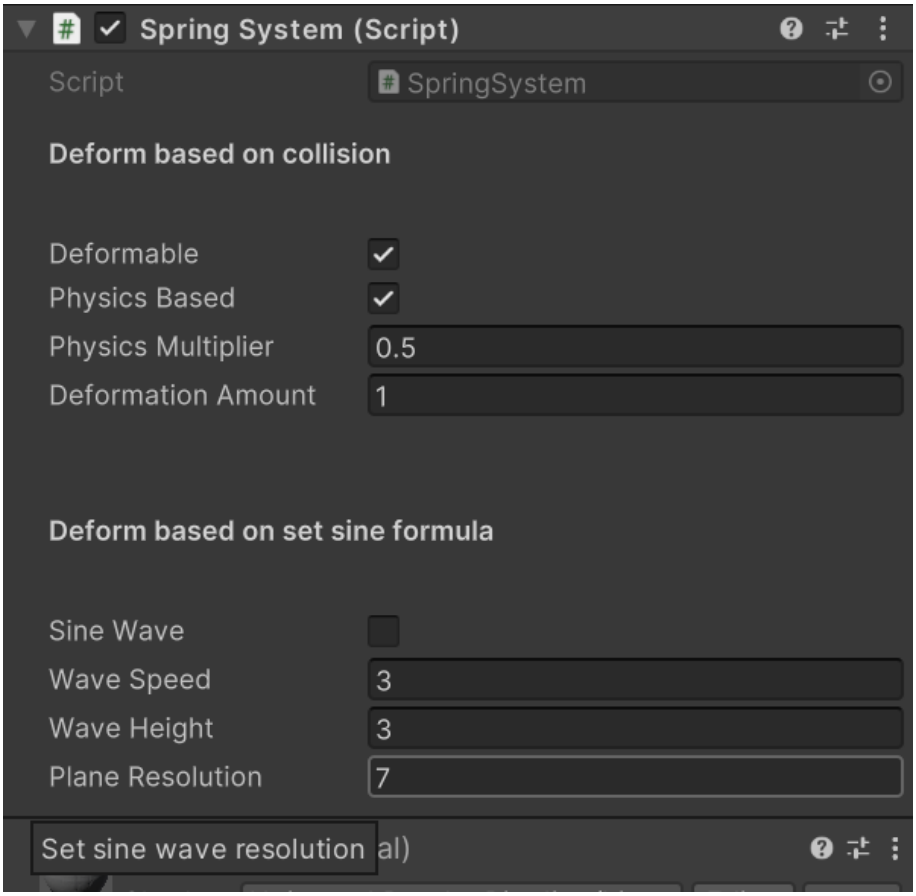


- *SpringSystem.cs*

With all this known we can now begin to apply these changes to an already existing mesh. This is a significant difference to creating our own as there is know way of knowing the layout, resolution, and shape of the pre-existing mesh.

```
4   [RequireComponent(typeof(MeshFilter))]
5   [RequireComponent(typeof(MeshCollider))]
6
    Unity Script (4 asset references) | 0 references
7   public class SpringSystem : MonoBehaviour
8   {
9       //Public
10      //UI Elements
11      [Header("Deform based on collision")]
12      [Header("")]
13      [Tooltip("Set object to be deformable by external forces")]
14      public bool Deformable;
15      [Tooltip("Set deformation ammount to be based on the physics of the colliding object")]
16      public bool PhysicsBased;
17      [Tooltip("Set multiplier for physics based deformation formulas")]
18      public float PhysicsMultiplier = 1.0f;
19      [Tooltip("Set deformation amount if not physics based")]
20      public float DeformationAmount = 1.0f;
21
22      [Header("")]
23      [Header("Deform based on set sine formula")]
24      [Header("")]
25      [Tooltip("Set deformation to sine wave")]
26      public bool SineWave;
27      [Tooltip("Set sine wave speed")]
28      public float WaveSpeed = 1;
29      [Tooltip("Set sine wave height")]
30      public float WaveHeight = 1;
31      [Tooltip("Set sine wave resolution")]
32      public float PlaneResolution = 1;
33
34
35      //Private
36      //Mesh Variables
37      private MeshFilter MeshFilter;
38      private MeshCollider MeshCollider;
39      private List<Vector3> SourceVertices;
40      private Vector3[] Normals;
41      private Mesh mesh;
```

Similar to before, the first thing to do is to define the variables to be used in the script. The difference in this case is the fact that this is intended to the be the final product of the project. This means that there are some more requirements to be met such as including an easier UI for the user to interact with. It is for this reason that this script implements use of the header and tooltip functions within Unity. In Unity, the [Header] attribute adds a header label in the inspector UI for a group of related properties or fields. It helps organize and visually separate different sections of properties by providing a title or descriptive text. The [Tooltip] attribute adds a tooltip that appears when hovering over the name of a variable/property in the inspector UI. It provides additional context or explanation about what that variable represents or how it should be used. These attributes allow for an easy to read and understand UI which save developers time in learning the system.




```
Unity Message | 0 references
47  private void OnValidate()
48  {
49      if (Deformable)
50      {
51          SineWave = false;
52      }
53      if (SineWave)
54      {
55          Deformable = false;
56      }
57  }
```

The OnValidate() function is a yet another special method in Unity that is automatically called whenever a value of an exposed variable (serialized field / public variable) in the Inspector changes. It can be used to perform actions or validations when those variables are modified. In this case it is being used to ensure that the user does not have both the SineWave and Deformable booleans selected at the same time. This is important because these booleans both have conflicting behaviours.

```
Unity Message | 0 references
59  private void Start()
60  {
61      //Get Component
62      MeshFilter = GetComponent<MeshFilter>();
63      mesh = GetComponent<MeshFilter>().mesh;
64      Normals = mesh.normals;
65      MeshCollider = GetComponent<MeshCollider>();
66      SourceVertices = new List<Vector3>();
67  }
```

The Start function for this script is also used to establish base values for many of the variables within this script. For the most part these are self explanitory, however, one which is worth explaining is the use of mesh.normals. this method is used in order to create a list of Vector3 variables which will store the rotation data for each vertex within the mesh. This is important to get as it will be used for the translation to decide the direction of the deformation.

```
Unity Message | 0 references
73  private void Update()
74  {
75      if (SineWave)
76      {
77          RippleSine();
78      }
79
80      //Update Mesh Collider
81      MeshCollider.sharedMesh = MeshFilter.sharedMesh;
82  }
```

After updating the necessary effects and calculations related to sine waves (if SineWave is true), there's an update to MeshCollider component. Here, we set sharedMesh of MeshCollider (MeshCollider.sharedMesh) to match the sharedMesh of MeshFilter (MeshFilter.sharedMesh). This ensures that as changes are made to mesh via MeshFilter (which holds current object mesh data), those modifications are also reflected in associated MeshCollider so that collisions and physics interactions remain accurate with respect to modified geometry.

```
1 reference
88  void RippleSine()
89  {
90      var meshFil = MeshFilter.mesh;
91      var Vertices = meshFil.vertices;
92
93      PopulateSourceVertices(); //Set Initial Values
94
95      //Wave Loop
96      for (int i = 0; i < Vertices.Length; i++)
97      {
98          Vector3 CurrentNormal = Normals[i];
99
100         float sineTerm = Mathf.Sin((Time.time * WaveSpeed) + (Vertices[i].x + Vertices[i].z) * PlaneResolution);
101         Vector3 NewPos = SourceVertices[i] + CurrentNormal * (sineTerm * WaveHeight);
102
103         Vertices[i] = NewPos;
104
105     }
106
107     meshFil.vertices = Vertices;
108 }
```

The ripple sine function for the final version of the system is overall almost identical to the one from the spring plane demo. This is with the exception of a few changes. The first of these is the inclusion of the PopulateSourceVertices() function, which is used to set the starting points and base for all deformations. This version also makes use of the normals which were previously explained. They are implemented into the sine wave function in order to have the translation take place along that rotated axis defined by that normal. This deformation is then performed based upon the source vertices defined before.

```
2 references
void PopulateSourceVertices() // Create a list of initial vertex positions
{
    var mesh = MeshFilter.mesh;
    var Vertices = mesh.vertices;

    for (int i = 0; i < Vertices.Length; i++)
    {
        SourceVertices.Add(Vertices[i]);
    }
}
```

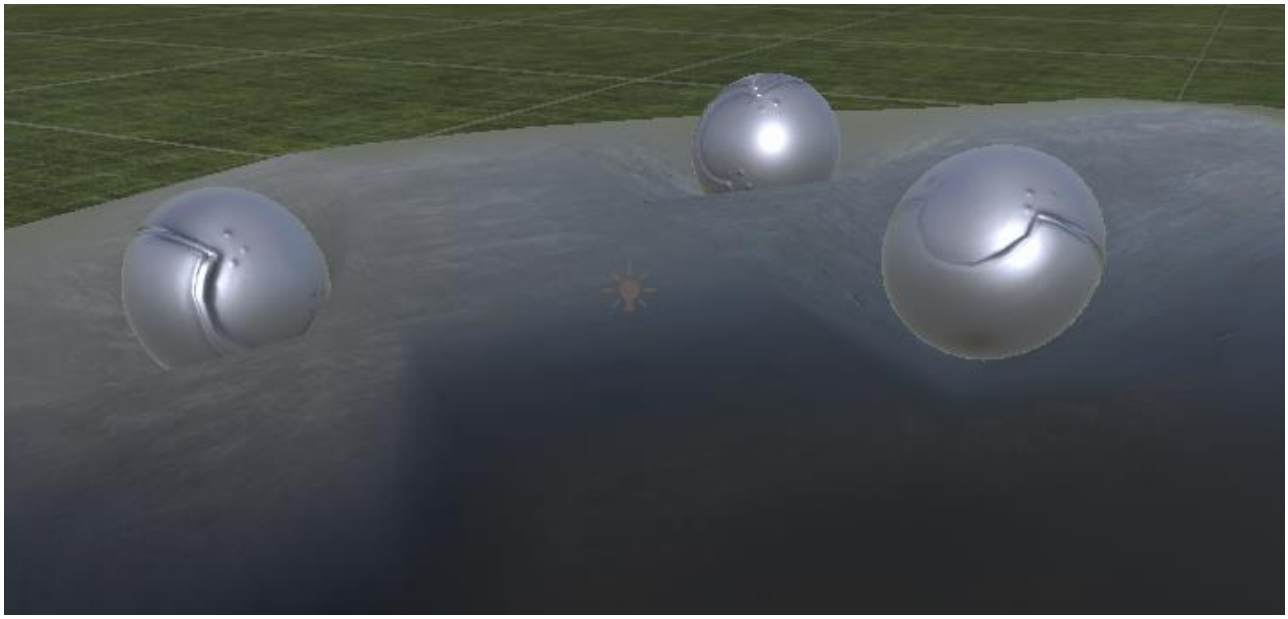
This function is a simple sub-function which is used to create a set of baseline vertex locations to be used as reference for other functions. Its main purpose is to turn the mesh.vertices call from an array to a list.

```
// Returns the nearest point to the colliding object
1 reference
public Vector3 NearestVertex(Vector3 point)
{
    // Convert point to local space
    point = transform.InverseTransformPoint(point);

    float minDistanceSqr = Mathf.Infinity;
    Vector3 nearestVertex = Vector3.zero;

    // Scan all vertices to find nearest
    foreach (Vector3 vertex in mesh.vertices)
    {
        Vector3 diff = point - vertex;
        float distSqr = diff.sqrMagnitude;

        if (distSqr < minDistanceSqr) // Check if current vertex is closer than the last
        {
            minDistanceSqr = distSqr;
            nearestVertex = vertex;
        }
    }

    return nearestVertex; // Return the closest vertex
}
```

The NearestVertex() function is the primary functionality which allows the collision detection to work correctly. This function essentially works as a distance between two (2) points formula. It takes in a point as a Vector3, this will represent the colliding object. A foreach loop is then used to cycle through all vertices and a distance between two (2) points formula is used to determine which vertex is closest. This is then returned by the function to be used.

```
150     // Detect collisions on deformable object
        Unity Message | 0 references
151     void OnCollisionEnter(Collision collision)
152     {
153         if (Deformable)
154         {
155             //Get Data From Colliding Object
156             Vector3 HitVert = NearestVertex(collision.transform.position); //Get closest Vertex to colliding object
157                                                                            // Calculate overall speed
158             float ColliderSpeed = Mathf.Sqrt(Mathf.Pow((collision.relativeVelocity.x), 2) + Mathf.Pow((collision.relativeVelocity.y), 2) + Mathf.Pow((collision.relativeVelocity.z), 2));
159             float ColliderMass = collision.rigidbody.mass; // Colliding object mass
160
161             //Dimentions used for calculations on planes only
162             float F = ColliderMass * ColliderSpeed;
163             float L = transform.localScale.y;
164             float A = transform.localScale.x * transform.localScale.z;
165
166             if (PhysicsBased)
167             {
168                 DeformationAmount = ((F / A) / (1 / L)) * PhysicsMultiplier; // Planar Deformation Formula Found in "Millington Game physics engine development"
169             }
170
171             var meshFil = MeshFilter.mesh;
172             var Vertices = meshFil.vertices;
173
174             PopulateSourceVertices();
175
176             for (int i = 0; i < Vertices.Length; i++)
177             {
178
179                 Vector3 CurrentNormal = Normals[i];
180
181                 if (Vertices[i] == HitVert) //find matching vertex from list to the one which was hit
182                 {
183                     Vector3 NewPos = SourceVertices[i] + CurrentNormal * (DeformationAmount - (DeformationAmount * 2));
184                     Vertices[i] = NewPos;
185                 }
186
187             }
188
189             meshFil.vertices = Vertices;
190         }
191     }
```

The OnCollisionEnter() function is a the last special method in Unity that is used in this project, it is automatically called when a collision occurs between the GameObject that has this script attached to it and another GameObject with a collider. In order to prevent wasted code excecution while the object is not set to be deformable, all code within the OnCollisionEnter() function is also held within a Deformable flag. Upon detecting a collision, the transform.position of the object is then passed into the NearestVertex() function, which will return the nearest vertex to the collision. The deformation can then be performed on this returned vertex. With this we can then begin calculating the overall deformation amount by first getting the speed of the colliding object. This is done through the use of a speed calculation which makes use of the .relativeVelocity function for all three (3) axes. This section also makes use of the most Mathf functions which is the standard math functionallity which can be included in C# scripts. We also can make use of the .rigidbody.mass function in order to get the mass of the colliding object. With the speed, mass, and nearest vertex defined the calculations can now be performed in order to determine the deformation amount. This is done by using the Planar Deformation Formula which is calculated and then applied to the vertex which matchis with the returned nearest vertex.



User manual

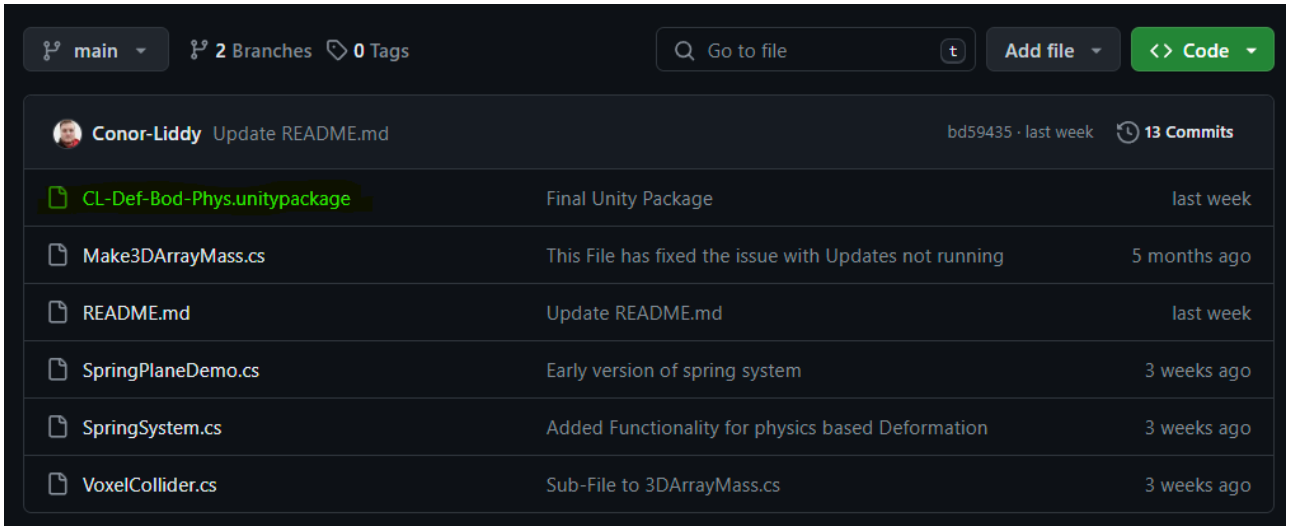
Contents:

- Make3DArrayMass.cs
- VoxelCollider.cs
- SpringPlaneDemo.cs
- SpringSystem.cs
- three (3) test scenes

Installation:

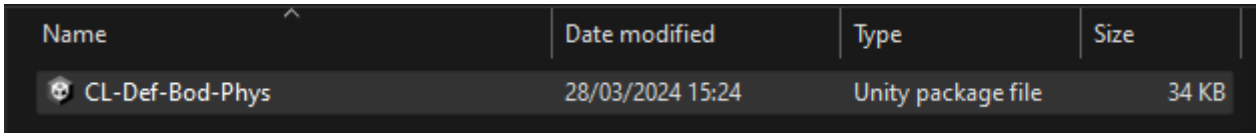
The Unity package can be found at: https://github.com/Conor-Liddy/CL-Deformable-Bodies-In-Unity

Once on the GitHub page you can then download the "*CL-Def-Bod-Phys.unitypackage*" file.

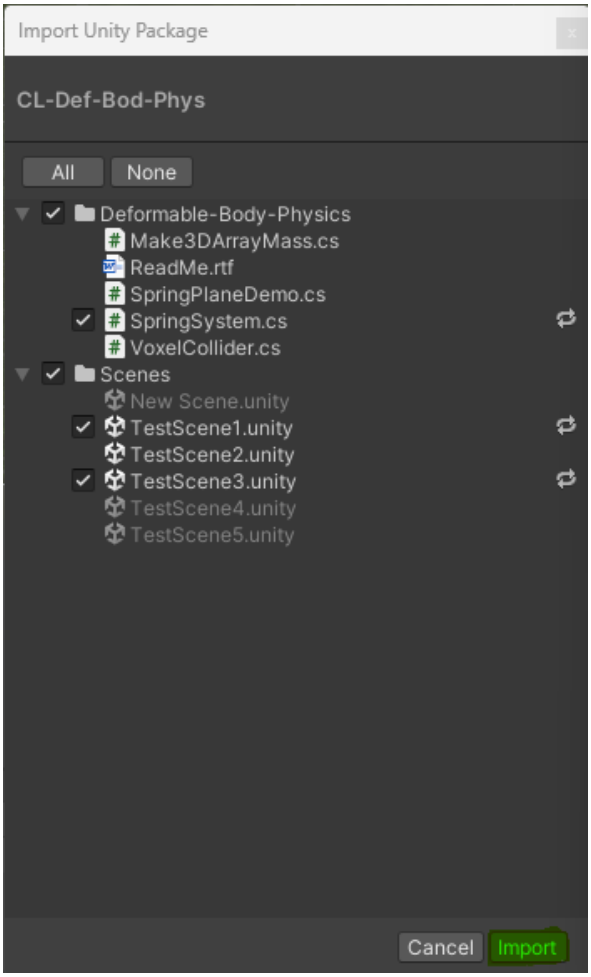


With this downloaded you can then start Unity and open the project you wish to use it in.

In file explorer double click on the package file which was downloaded.



This will open the “Import Unity Package” window, select “Import”.



This will then give you the “Deformable Body Physics” folder in the “Assets” folder.

Usage:

MASS SPRING SYSTEM / DEFORMABLE BODY PHYSICS SYSTEM (WIP)

1. Click “*Add Component*” in the “*Inspector*” window for the object within Unity that you wish to be "*Deformable*", and search for the “*SpringSystem.cs*” script, and select it.

2. Ensure the object also has the following components:
    - Mesh Filter
    - Mesh Collider
    - Rigidbody (If Physics are wanted)

3. If a sine function is what is wanted the select the "*Sine Wave*" boolean in the Unity Inspector window. You can then adjust the wave "*Speed*", "*Height*", and the "*Plane Resolution*" (*this functions as the wave density*) (*Note: Using a Sine function diasables the collision detection functions*)

4. If you want to make the object "*Deformable*" then select the boolean to do so.

5. Select if you want the deformation to be "*Physics Based*" or a set amount each time.

6. Non-physics based deformation can be adjusted by the "*Deformation Amount*" float.

7. The strength of physics based deformations can be adjusted by the "*Physics Multiplier*" Float (*Note: The strength of physics based deformation can also be affected by the mass of the colliding object*)

3D ARRAY MASS SYSTEM (WIP)

1. Click "*Add Component*" in the "*Inspector*" window for the object you wish to have "*voxelized*", and search for the "*Make3DArrayMass.cs*" script, and select it.

2. Set the desired dimentions for the 3D Array Mass in the Unity inspector window.

3. Set the booleans for the object (*E.g. Has Physics, Crumble*)

4. Apply a "*Material*" to the "*Voxels*" (*Note: If no material is applied then the child voxels will use the material of the parent object*)

5. Hit "*Play*" in the Unity editor to run the code.

SPRING PLANE DEMO

1. Create an "*Empty*" object within unity.

2. Ensure that the "*Empty*" object has both the "*Mesh Renderer*", and "*Mesh Filter*" components.

3. Click "*Add Component*" in the "*Inspector*" window for the "*Empty*" object, and search for the "*SpringPlaneDemo.cs*" script, and select it.

4. In the "*Inspector*" window we can now define the "*X*" and "*Z*" values for the size of the plane that will be generated. (*Note: This plane will be generated outward from the position of the "Empty" object*)

5. In the “*Inspector*” window you can now decide what type of sine wave is wanted. A “*Sine Wave*” will go from one (1) edge of the plane to the opposite, a “*Ripple Wave*” will go from one (1) corner of the plane to the opposite. You can then adjust the wave "*Speed*", "*Height*", and the "*Plane Resolution*" (*this functions as the wave density*)

6. Hit "*Play*" in the Unity editor to run the code.

Critical analysis and Conclusions

Throughout the implementation and testing of the deformable body physics system in Unity, several key aspects were examined to determine its effectiveness and usability. The following critical analysis highlights the strengths, limitations, and overall conclusions drawn from this project. One of the primary goals of this project was to create a deformable body physics library that could be universally applied to any object within the Unity editor. This goal was successfully achieved by using existing mesh data as a basis for deformation calculations, allowing developers to apply the system to any imported or created object. Another important aspect was providing an intuitive user interface within the Unity editor for adjusting parameters related to deformation, wave effects, and other properties. By utilizing attributes like [Header] and [Tooltip], the UI became more organized and informative for users. Considerable effort went into optimizing computational efficiency to ensure real-time performance even when simulating complex deformations on multiple objects simultaneously. Techniques such as de-rendering non-visible voxels or updating only necessary calculations improved overall performance. The current implementation primarily focuses on planar deformation rather than volumetric deformation due to performance challenges associated with using 3D array masses or voxels extensively in Unity scenes. While collision detection is implemented for deformable bodies interacting with rigid objects or other deformable bodies accurately preserving physical accuracy, there are still limitations in terms of advanced physics interactions like friction or fluid dynamics, which are all features that I would like to add in the future. he final implementation of a deformable body physics library for use in Unity provides an effective solution for game developers wanting realistic soft-body simulations without having extensive knowledge about complex physical modeling techniques themselves.By leveraging existing mesh data instead of creating individual objects (voxels), it allows easy integration into any Unity project while maintaining high performance. The user-friendly parameter adjustment interface and compatibility with the latest versions of Unity further enhance its usability.The limitations, such as limited physics interaction or the focus on planar deformation, can be addressed in future iterations by exploring advanced techniques like using 3D array masses with optimized algorithms for improved performance or incorporating more sophisticated collision detection and response methods.Overall, this deformable body physics library offers a valuable tool for game developers, VR developers, simulation engineers, educational institutions, indie developers, hobbyists/enthusiasts, and researchers to create immersive experiences and conduct experiments within Unity.

Bibliography/ References

1. DAVID H. EBERLY, CHAPTER 4 - DEFORMABLE BODIES, EDITOR(S): DAVID H. EBERLY, GAME PHYSICS (SECOND EDITION), MORGAN KAUFMANN, 2010,

PAGES 155-212, ISBN 9780123749031,HTTPS://DOI.ORG/10.1016/B978-0-12-374903-1.00004-9. (https://archive.org/details/gamephysics0000eber_w3s3/page/n5/mode/2up)

2. IAN MILLINGTON, 12 - COLLISION DETECTION, EDITOR(S): IAN MILLINGTON, GAME PHYSICS ENGINE DEVELOPMENT (SECOND EDITION), MORGAN KAUFMANN, 2010, PAGES 253-289, ISBN 9780123819765, HTTPS://DOI.ORG/10.1016/B978-0-12-381976-5.00012-7. (https://www.sciencedirect.com/science/article/pii/B9780123819765000127)

3. SINGH, N.P., SHARMA, B. AND SHARMA, A. (2022). PERFORMANCE ANALYSIS AND OPTIMIZATION TECHNIQUES IN UNITY 3D. [ONLINE] IEEE XPLORE. DOI:HTTPS://DOI.ORG/10.1109/ICOSEC54921.2022.9952025.

4. DR. EDWARD LAVIERI (2015). GETTING STARTED WITH UNITY 5 : LEVERAGE THE POWER OF UNITY 5 TO CREATE AMAZING 3D GAMES. [ONLINE] EBSCOHOST. BIRMINGHAM, UK: PACKT PUBLISHING. AVAILABLE AT: HTTPS://EDS.S.EBSCOHOST.COM/EDS/DETAIL/DETAIL?VID=11&SID=32AA4FA4-0380-4CEAA9B3EE41FDEF2D36%40REDIS&BDATA=JNNJB3BLPXNPDGU%3D#AN=1000532&DB=E000XWW

5. Palmer, G. (2005). Physics for Game Programmers. Apress eBooks. doi: https://doi.org/10.1007/9781430200215.%40REDIS&BDATA=JNNJB3BLPXNPDGU%3D#AN=1000532&DB=E000XWW

6. David Bourg, B.B. (2013). *Physics For Game Developers 2nd Edition*. [online] *Internet Archive*. Available at: https://archive.org/details/physics-for-game-developers-2e

7. Christer Ericson. (2005). Real-Time Collision Detection. eBook. http://www.r-5.org/files/books/computers/algo-list/realtime-3d/Christer_Ericson-Real-Time_Collision_Detection-EN.pdf

8. Verth, J.M.V. and Bishop, L.M. (2004). *Essential Mathematics for Games and Interactive Applications: A Programmer's Guide*. [online] *Google Books*. Taylor & Francis. Available at: https://books.google.ie/books/about/Essential_Mathematics_for_Games_and_Inte.html?id=19SpFYj82owC&redir_esc=y

9. mathfor3dgameprogramming.com. (n.d.). *Mathematics for 3D Game Programming and Computer Graphics*. [online] Available at: https://mathfor3dgameprogramming.com

10. Catto, E. (2005). *Iterative Dynamics with Temporal Coherence*. [online] Available at: https://www.gamedevs.org/uploads/iterative-dynamics-with-temporal-coherence.pdf

11. Redon, S., Kheddar, A. and Coquillart, S. (2002). Fast Continuous Collision Detection between Rigid Bodies. *Computer Graphics Forum*, 21(3), pp.279–287. doi:https://doi.org/10.1111/1467-8659.t01-1-00587

12. Szauer, G. (n.d.). *Game Physics Cookbook*. [online] Available at: http://ndl.ethernet.edu.et/bitstream/123456789/24446/1/Gabor%20Szauer_2017.pdf

13. Erleben, K., Sporring, J., Henriksen, K. and Dohlmann, H. (n.d.). *Physics-Based Animation*. [online] Available at: https://iphys.files.wordpress.com/2020/01/erleben.ea05.pdf.

14. Introduction to 3D Game Programming with DirectX® 12. (n.d.). Available at: https://terrorgum.com/tfox/books/introductionto3dgameprogrammingwithdirectx12.pdf.