Team 03
Yashar G. Ahari
Eric Gonzalez
Adam Gracy
David Burris

Sphere Manipulation in Unity

This project is an attempt to provide an interactive sphere manipulation simulation to a user. The general problem/need our group wanted to solve is that we wanted to create a working object collision simulator that would allow the user to better understand mesh collision mechanics in a three-dimensional environment. The first step to accomplishing this goal is that our team wanted to use code to generate a sphere-like polyhedron to be used as an agent for collision detection purposes. Once the sphere is completed, the simulation would allow the user to take control of an object being shot towards the sphere. The object (or projectile) shape, velocity, mass, and size can be changed by the user with UI controls.. Each projectile will be launched by the user, and when the projectile makes contact with the polyhedron our goal is for simulated damage to occur. The way we represent collision damage is by re-generating vertices, edges, and faces with different vertex coordinates. The project is intended for users to see a short-duration interaction between two entities simultaneously. The result was that our group was able to produce a polyhedron using a customized data structure, but realistic damage is not currently simulated. Our implemented simulation propagates energy from the initial collision to adjacent triangles on the sphere. This simulation is still valid because it can help the user understand simple kinetic energy transfer from one object to another object.

The example section should also take up 10% (210 words).

Figure 1.0

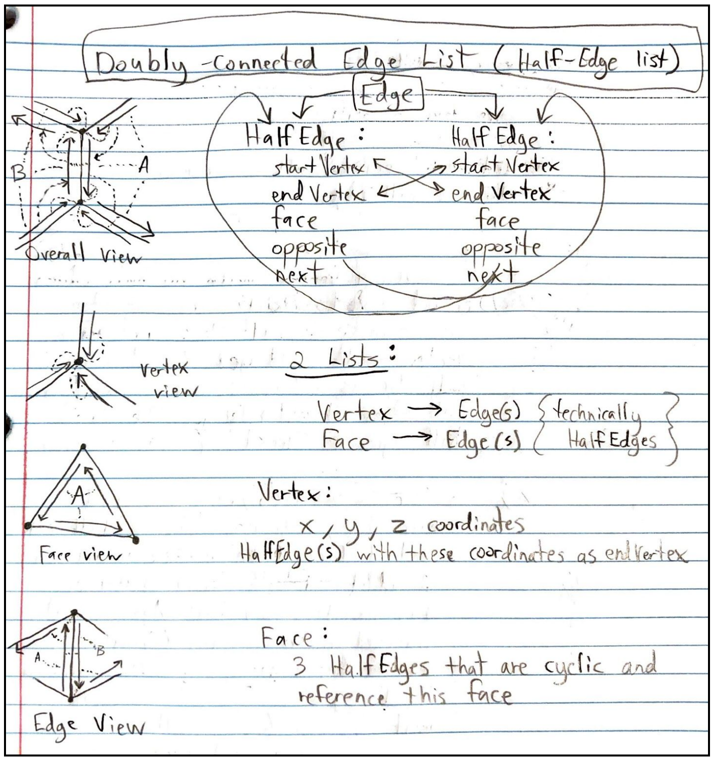


Figure 1.1

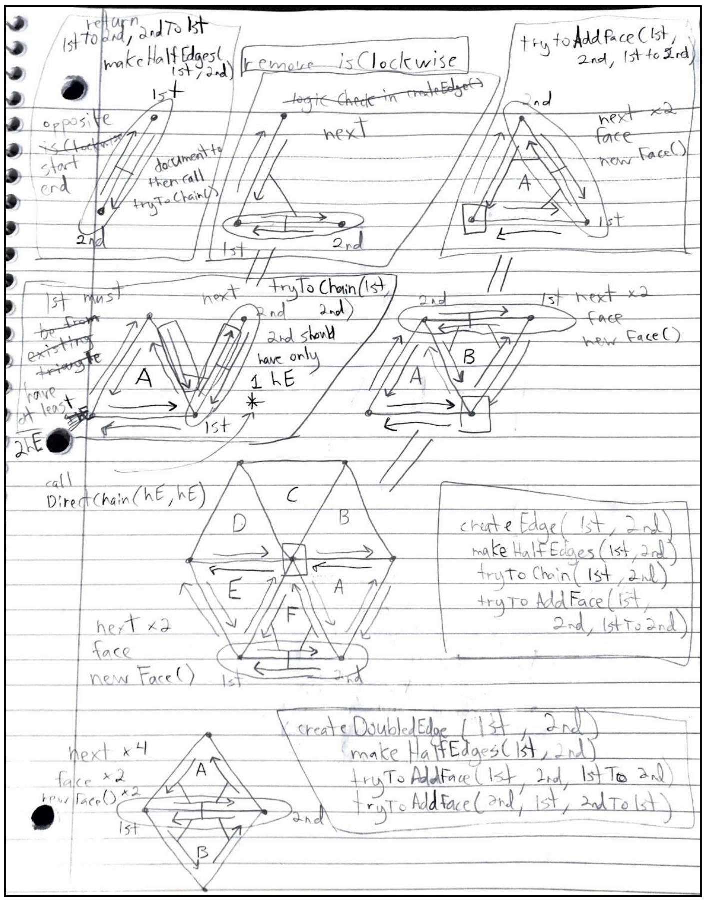
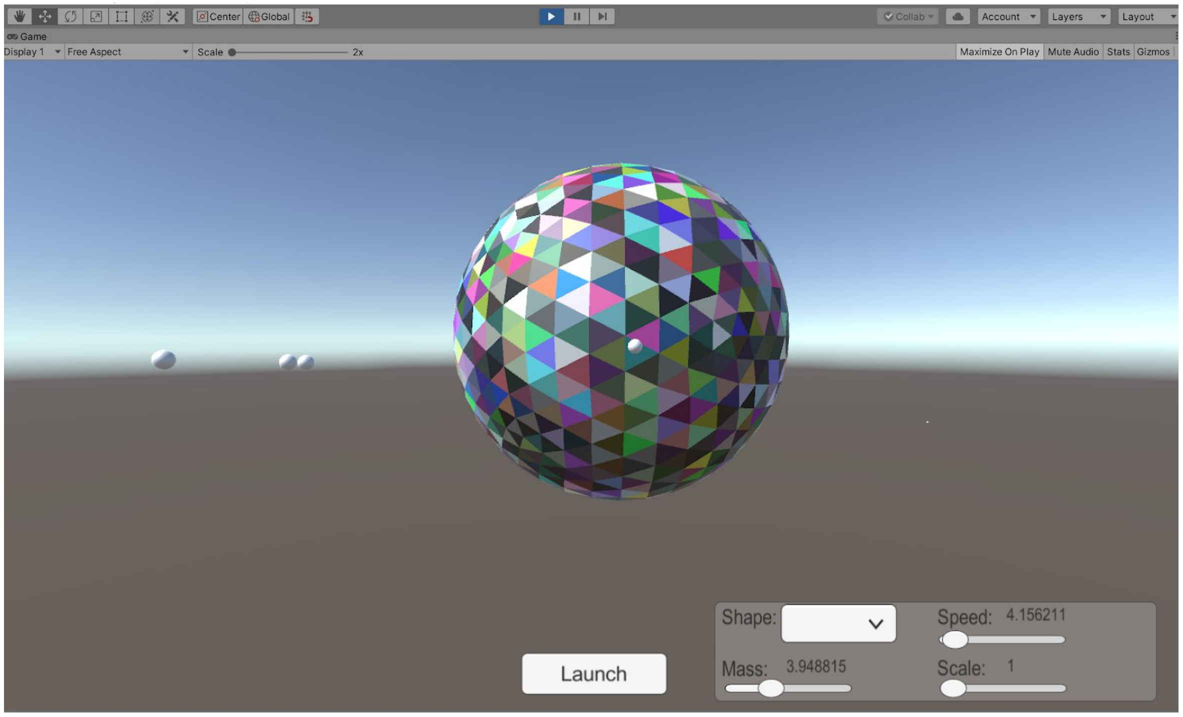


Figure 1.2



Scale 2x

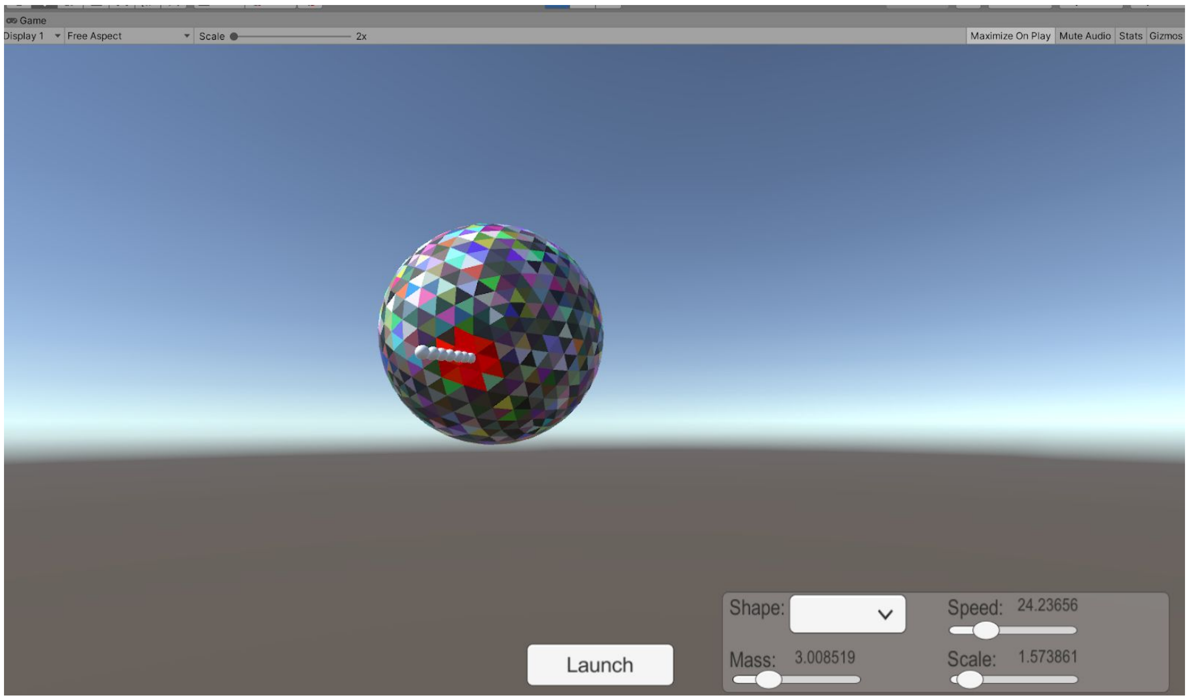
Shape:

Speed: 4.156211

Mass: 3.948815

Scale: 1

Figure 1.3



Shape:

Speed: 24.23656

Mass: 3.008519

Scale: 1.573861

An example of how our product works is through figures 1.0 and 1.1. First, figures 1.0 and 1.1 describe the logic of how the icosahedron triangulation procedure is generated. This polyhedron is accomplished in 3 high-level phases: Creating the top section, fanning outward from the "north pole" of the polyhedron, from theta = 0; adding on downward-facing triangles within the middle section; and adding the bottom section whilst generation of the middle section is completed, weaving into the "south pole". After the icosahedron is complete, a set of all Edge objects in the icosahedron is created, and for each Edge, that edge is "split" so that a midpoint is created and pointers are modified to point to the new midpoint. After all edges are split, the triangle faces can be triangulated. This is done by storing references to the midpoints of each triangle face, dereferencing all existing triangle faces, and connecting the stored midpoints to create the triangulated faces within the "original" triangle face. This procedure turns one triangle face into four triangle faces. Once triangulation has been performed, the vertex coordinates need to be reset to correct spherical radius length. This is done simply by converting the coordinates of the vertices from cartesian to spherical, adjusting the radius parameter of the spherical coordinates, and then re-converting back to cartesian coordinates.

Figures 1.2 and 1.3 describe the simulation portion of our project. Figure 1.2 displays the user UI wherein the user may select from a number of options that edits the projectile. The user may opt to select a sphere, cylinder, cube, and capsule. The mass, speed, and scale of the projectile can also be edited by the user. Figure 1.3 demonstrates how mesh collision detection is simulated. When a projectile has collided with the icosahedron, each triangulated face turns red to represent a collision. Each amount of triangle face that has been collided is also determined by the projectiles mass, speed, and scale. The larger these values hold, the more triangle faces will change colors to represent collision detection.

We opted to use the Unity game engine. Unity provided an easy and fast platform and plenty of tools for implementing our project. In order to finish the project, we divided the project into four steps: Creating and defining the necessary infrastructure to create a base for development, making the necessary user interface for testing, creating and refining the sphere generator algorithm, and creating the collision interaction and triangle manipulation. The first step involved creating the necessary classes and infrastructure to make the development possible. The earliest work was making the visualization possible to provide a platform for developing the sphere generator. The main collision object created that creates an Unity game object for any given triangle. The second step was also a crucial step for making the later development and testing possible. For this, the camera movement implemented and also the projectile manipulation and lunch UI created. Initially for the third step, we planned on using an existing sphere generator which is made by Yashar for his HCI project. The problem with that algorithm was that the generated spheres were made of the different sized triangle, with high concentration of small triangles in the poles and low concentration of big triangles in the equator in which was not the best shape for collision simulation. So we worked on creating a sphere generator that created a sphere with equilateral triangles. The fourth step was the most important, but unfortunately the latest one resulted in multiple setbacks, including moving the classes online therefore reduction of the performance of the team. We were expected to start working on the collision detection much earlier, however, we faced some complications on creating the sphere generator which pushed the work on collision to after spring break. Moving classes online caused a huge reduction in our team's performance as we used to meet and discuss the work at least once a week. Therefore we could not fully implement the collision deformation. As mentioned in before, we changed our plan to use a different technique on generating the sphere.

We wanted a minimal but easy-to-use interface that requires little training or knowledge of the program. Unity has a great user interface system that helped accomplish this. We were able to manage all of the user interface items that the user sees on the screen in one game object called “Canvas”. In the object, we hold the projectile launch button, shape selection dropdown, and the three sliders that set the properties of the projectile. The three sliders manipulate the projectile model (described below) from within our *Controller* script.

When the user interacts with the settings panel to adjust the projectile’s values, the *Controller* script, which has an instance of the projectile model class called *Model*, will detect the adjustment using built-in event handlers, and update the model accordingly.

The *Controller* script will also handle what happens when the user clicks the *launch* button. When the user clicks launch, the *Controller* will load the selected, prefabricated shape, and set it in motion.

We also implemented camera control to be able to move around the scene. We downloaded a premade script called *H4CameraOperate* from the Unity asset store to make our lives easier. The code is properly cited in the source file named *CameraOperate.cs.*

As described on the progress reports, we realized that for the best possible outcome, we needed our sphere to be filled by the equilateral triangles. For that reason we opted to create an algorithm to generate and sphere based on icosahedron. The algorithm creates a base icosahedron which is scalable in size. Later, the algorithm divides each triangle face of the icosahedron to three smaller triangles which are also equilateral and then pushes the middle vertices of the triangles to match the sphere radius. Continuing this process for three times, creates a great looking sphere which is made out of equilateral triangles.

The data structure to contain the information about the sphere, as described in greater detail in the progress report, are made out of “face” objects in which they contain “edge” and “half edge” objects that describe the vertices of the triangle. The unique properties of these objects are they are all linked to each other, which means that from an “half-edge” object, we can access the opposite half-edge and from half-edge, the face and so on, therefore, making it easy to access the neighboring triangles components. Also, each of the faces gets an unique id which is useful to find and manipulate each of the faces.

Each of the face objects that are created and stored in the general data structure, will turn into an unity game object, we named them Triangles. Triangles are individual game objects that all share one parent which we named it “Colid” as for the collider. Triangles are meant to be the link between abstract data structure and visualization and they are responsible for hit detection, and acting upon detecting the hit.

Each triangle represents a face which is created by the sphere generator and contains the information about the face it represents. This information includes a unique identification (UID) which is the same as the unique id of the face, the UID of neighbor triangles (sharing same edge), the face object, which includes the vertices for the mesh. Also it is important to mention that the name of each triangle in the Unity game object hierarchy is the same as the UID of that triangle. The triangles upon the creation provide the vertices to the mesh renderer and mesh collider components. Those components are provided by the Unity that can be attached to every game object. Given the vertices in the right order (clockwise), mesh renderer will render the triangle. For purposes of making each triangle visible and distinguishable, we gave each triangle a random color. The mesh collider provided for each triangle will cover the whole surface of each triangle and provides an event based capability to act for each collision detection.

Our initial approach to explore the collision and deformation was to figure out the proper way of propagating the events throughout the triangles. Given that each triangle stores the UID of their neighbors, we retrieved the game object of the neighbor triangles by calling a find function that goes through the game objects in the Unity game object hierarchy and retrieves the game object based on the provided name. This proved to be a highly powerful technique on accessing the neighbor triangles.

For purposes of making the calculations easy, we assumed that the colliding object loses all of its energy and gives it to the sphere. Therefore we can calculate the energy of each collision event as the kinetic energy of the colliding object, which is equal to $E = \frac{1}{2} \times m \times v^2$. Here, “m” is the mass of the object and “v” is the velocity of the object. When each triangle energizes, there are two possible actions it can perform. One of them is to pass that energy to the neighbor triangles, or consume the energy. The consumption of energy for each triangle means two things, one of them is that some of the energy turns into heat and dissipates as infrared radiation, and the rest would be spent on deformation. We could not manage to implement the deformation, therefore to make the energy exit from the system, an arbitrary energy loss implemented to reduce the energy from the triangles on each frame. Also , each triangle is implemented with a limit on maximum energy that they can contain, and if the energy level is higher than that, they divide and pass the leftover energy to their neighbors. For the purpose of graphical visualization of that energy interaction, triangles turn red as they gain energy, and return to their normal color as they lose their energy.

At the beginning of the semester, our core project idea revolved around the simulation of damage on a programmatically generated sphere being caused by a launched projectile. Unfortunately, we were unable to implement the simulated damage to the generated sphere as we had previously planned. Time is a critical factor in missing this key feature. We would like to see this damage simulation feature added in the future.

We know that having only the option to test a spherical object is boring! We would like to add the ability to test other objects, such as polyhedra. As the shapes become more complicated, so will our understanding of the algorithms. Eventually, we could be able to simulate damage to real-world objects such as vehicles, buildings, et cetera.

Another nice feature would be to simulate the generated object, and projectiles, being made from different materials, and test how they interact with one another at high speeds. For example, the generated sphere could be made from diamond, while the projectile is made of tungsten carbide. It might be an interesting experiment to see how these two strong materials interact.

Currently, once the projectile impacts the generated sphere is destroyed by Unity. It would make for a more robust and realistic experience if we implemented the physics necessary to simulate the reflection and direction of travel of the projectile after impact.

In conclusion, it is a bit disappointing that we were not able to implement the original plan we presented at the beginning of the semester because we encountered a few stubborn bugs that really ate up a lot of time. Perhaps, in future projects, we will be more cognizant of our time management to avoid missing deadlines.
However, not all is lost, we were able to simulate visualization of energy dispersion when a projectile hits a sphere. Though far from perfect, we believe that the work we did get done is still useful, interesting, and a great base for future projects that need spherical model manipulation.

Adam worked on the user interaction portion of the project. This includes implementing a camera script to move around the scene, a projectile launcher, as well as the user interface. Yashar worked on creating the initial infrastructure and some base classes, visualization of the triangle, dealing with the hit detection and energy propagation.

Eric was responsible for overseeing work being done by the team, mitigating roadblocks found during work, user testing and documentation.

David handled the icosahedral triangulation design, programming the icosahedron, and programming the triangulation procedure.

Yashar guided the team towards the basic understanding of the technology of Unity, and helped research data structures and algorithms capable of basic mesh collisions.