Università della Svizzera italiana

Faculty of Informatics

Bachelor Thesis

June 23, 2023

Gelatin Cubes: Soft Body Simulation

Giacomo Solaro

Abstract

This project aims to build a soft body simulation which simulates a deformable object using the mass-spring system and that is able to detect collisions and bounce off some rigid, fixed in space objects, mainly spheres and planes. We will see in detail the underlying physics principles involved in such simulations and the methodologies needed to build them in a relatively simple way that will produce realistic results. The implementation of this project also provides a graphical user interface that allows the user to experiment by positioning objects and changing the physical proprieties of the gelatin cube, which in combination with the visualization of the forces through a colormap, should provide the user with a better understanding of the physics involved and the limitations of the implemented methods. As an extension of the project, we also implemented a simulation in which it's possible to define objects made of gelatin cubes in a "Minecraft" fashion by importing a simple 3D model from an obj file which is then converted to a soft body using soft cubes that share the physical proprieties of jelly.

Advisor
Prof. Piotr Didyk

Advisor's approval (Prof. Piotr Didyk): Date:

Contents

1 Introduction

Physic-based soft body simulations aim to model the realistic behavior of deformable objects according to the laws of physics. In computer graphics, these simulations are applied in a variety of fields such as video games, movies and virtual reality to create realistically-looking environments and animations of objects that can bend, jiggle, deform and retain their shape to a certain degree when reacting to forces and collisions.
There are a variety of techniques to create such simulations, such as the finite element method, position-based dynamics, finite difference method, position-based dynamics and the mass-spring system.

The focus of this project is to create a simulation using a mass-spring system, to simulate a soft cube made of gelatin that falls onto planes and spheres under the influx of the gravity force, detects the collision and reacts accordingly. The project is then extended by discussing how to jellify an imported model using gelatin cubes.

We will start by explaining how the mass-spring system works, and implement it to build a soft cube model. Next, we will discuss and implement a simulation algorithm to compute forces, velocities and update the position of the soft cube. With the main algorithm of the simulation defined, we will see in detail how to implement the various component such as collision detection, collision reaction and friction between the soft cube and solid objects fixed in space (mainly spheres and planes). A brief explanation of how to optimize the system using bounding volumes will follow, together with a brief presentation of the GUI, the features and limitations of the implemented simulation. In the last part of the project, we will see how to extend the simulation to import 3D models from obj files and recreate them using our soft cubes, with a particular focus on how to position the cube using a grid and how to connect them.

The structure of this document follows a simple intuitive step-by-step approach which is easy to follow, each section starts with a brief introduction of a topic or problem to solve, which is then followed by a brief overview of the theoretical notions, concepts and ideas needed to understand both the reasoning and the physics involved in the implemented solution. The sections are then concluded with a practical solution or implementation complemented with the necessary mathematical formulas, and algorithms. The actual results in the simulation are also shown where possible.

2 The mass spring system

The mass-spring system is the core component of our soft-body simulation. It consists of a network of interconnected masses (or particles) and springs that define the dynamics of our soft object. Thanks to Hooke's law, this very simple system can model the realistic behavior of deformable objects affected by external forces, such as gravity or that are colliding with other objects.

The first step for the implementation of the mass spring system is to compute the forces that the springs exert on each particle. To do so let us consider the simple case shown in **Figure 1** in which 2 particles P_i and P_j are connected by a single spring.

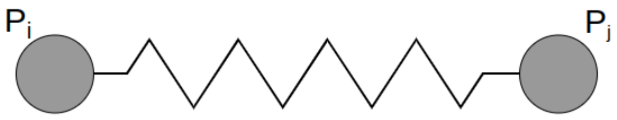


Figure 1. Two particles connected by a spring

According to Hooke's law, the force F generated by spring that has been extended or compressed by a distance x is equal to:

$$F = x \cdot k_s$$

where k_s is a constant characteristic of the spring called stiffness and the distance x can be computed as the difference between the current length of the spring at rest.

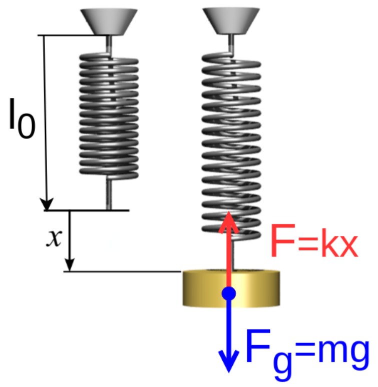


Figure 2. Hooke's law force system equilibrium [3]

By considering our simple mass-spring system at equilibrium(**Figure 3**) we can easily compute the forces acting on the particles by considering them proportional to the relative elongation of the spring as follows:

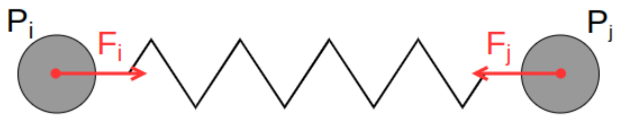


Figure 3. Force generated by a spring on two particles

$$F_i = k_s \left(\frac{\|x_j - x_i\|}{l_0} - 1 \right) \frac{x_j - x_i}{\|x_j - x_i\|}$$
$$F_j = -F_i$$

where: x_i,x_j are the vectors containing the coordinates of the two particles. and l_0 is the length of the spring at rest.

The last force we need to consider in our mass-spring system is the damping force, which takes into account the energy dissipation of our oscillating system and which depends on the relative velocity of the particles along the spring and on the damping coefficient d_s of the spring.

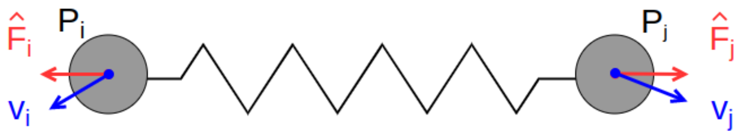


Figure 4. Spring damping force and velocities of the particles

$$\hat{F}_i = d_s \left\langle \frac{\|v_j - v_i\|}{l_0}, \frac{x_j - x_i}{\|x_j - x_i\|} \right\rangle \frac{x_j - x_i}{\|x_j - x_i\|}$$

$$\hat{F}_j = -\hat{F}_j$$

Thus the total force on a particle is:

$$F_{itot} = F_i + \hat{F}_i$$

To compute the total force of the springs acting on a particle of our cube we simply apply the above formula to each spring connected to the particle and sum all the forces affecting it.

3 The cube model

Now that we know how the mass-spring system works, we can use it to create a simple soft object to simulate, in this case, a cube. This model will be composed of two components, the "physical one and the "visual one", the former will be used for the physics calculations (forces, velocities, positions, collisions,...) that will be carried on the CPU, while the latter is used for rendering computations (colors, lights, ...) that are carried on the GPU. The latter component is the one we will visualize in our simulation.
We will start with the mass-spring system for the physical cube model, which is made of particles connected by springs. How these springs are connected will influence the overall behavior of the cube, in the next section we will discuss the most common and simple springs arrangement which allows to accurately model the behavior of a soft body.

3.1 The mass-spring system for the physical cube model

The physical cube model is composed of particles and springs. To create the cube structure we can imagine dividing a cube into sub cubes. We want to place a particle in each vertex of each sub-cube without creating duplicated particles. A basic example with indexed particles is shown in **Figure 5** (in this case the value of the particle's indexes is $i = 0, j = 0, k = 0$)

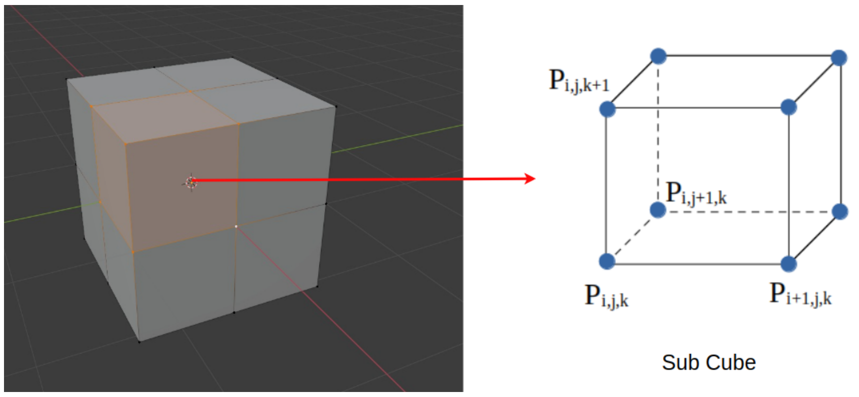


Figure 5. Cube divided in sub cubes (left), and physical sub cube with particles (right)

We then need to connect the particles with springs. To do so we consider three types of springs:

- **Structural springs:** they define the basic structure of the cube and connect each particle to its six direct neighbors if these exist. For example the particle $p_{i,j,k}$ will be connected with structural springs to the following particles:
$p_{i-1,j,k}$, $p_{(i+1,j,k)}$, $p_{(i,j-1,k)}$, $p_{(i,j+1,k)}$, $p_{(i,j,k-1)}$, $p_{(i,j,k+1)}$. In the cube showed in **Figure 5** the structural springs are the edges of the cube

- **Shear springs:** they prevent the cube from distorting, and together with structural springs they provide stability to the structure of the cube. they connect each particle to its face and body diagonal neighbors. Since it would take quite a while to describe the particle connections, an example of how to connect the particle on a 2x2x2 cube is shown in **Figure 6**, where for visual clarity the cube's body diagonal springs and the face diagonal springs have been separated in two images.
- **Bending springs:** they prevent the cube from bending, providing contraction resistance, connection wise, they are similar to structural springs but they connect each particle to its second direct neighbor. For example the particle $p_{i,j,k}$ will be connected with bending springs to the following particles: $p_{i-2,j,k}$,$p_{i+2,j,k}$,$p_{i,j-2,k}$,$p_{i,j+2,k}$,$p_{i,j,k-2}$,$p_{i,j,k+2}$). Note that the minimum cube size where the bending springs are defined is a 3x3x3 cube

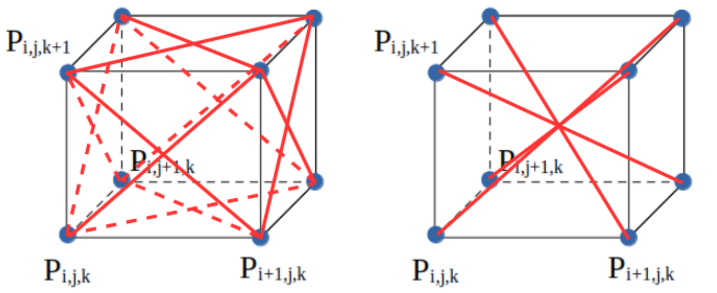


Figure 6. Cube with face shear springs (left) and body shear springs (right)

3.2 The rendered cube model

In the rendered cube model, we want to store only the vertices that correspond to the particles that are located on the faces of the cube, duplicating the vertices that are shared between the cube's faces. This will allow us to compute the lighting of the cube using the Phong model since every vertex of each face will have its own well-defined normal. (Remember that the positions of these vertices are the same as the positions of the corresponding particles located on the faces of the cube. The position of these vertices must thus also be updated if the position of said particles changes. This correspondence between particles and vertices is what connects the "physical" cube model and the "visual" cube model, the details of the computations involved will be explained later in the Simulation section).
Each face of the cube is rendered by rendering the series of sub-triangles of the sub-squares that make up the faces of the cube. This can be done by creating an array that contains the indexes of the nodes that make up the triangle, it's important to remember that the order in which these indexes are stored matters and that it must be counterclockwise.

4 Simulation

To run the simulation we need to compute the velocities and positions of the particles according to the applied force. The simulation algorithm can be derived from Newton's second law:

$$F(t) = m \cdot a(t)$$

where F is the force, m is the mass of the particle and a the acceleration. Thus we can compute the acceleration as:

$$a(t) = \ddot{x}(t) = \frac{F(t)}{m}$$

Since the acceleration is the second derivative of the position $x(t)$ with respect to time and the velocity $v(t)$ is the first derivative of the position with respect to time, the velocity can be computed as:

$$v(t) = \dot{x}(t)$$

We can now analytically compute the velocity and position of a particle at time t as follows:

$$v(t) = v_0 + \int_0^t a(t)\,dt = v_0 + \int_0^t F(t)/m\,dt$$

$$x(t) = x_0 + \int_0^t v(t)\,dt$$

4.1 Integrators

During simulation, we want to solve the integrals mentioned above, numerically and in the simplest way possible. This is what integrators do. There are several integration schemes, each with its own advantages and disadvantages, the most popular choices are the Euler integration schemes and the Runge-Kutta Integration scheme of second and fourth degree.

The biggest advantage of Euler integration schemes is that they are first-order integrators and thus their computational complexity is lower with respect to other methods, the biggest drawback is that these integrators are unstable for big time steps due to the fact that they "step blindly into the future", by considering the force to be constant throughout the entire time step. The value of Δt is usually in the order of 10^{-3},10^{-4}. As an example, the explicit Euler schemes can be computed as follows:

$$v(t+\Delta t) = v(t) + \Delta t \cdot F(t)/m$$
$$x(t+\Delta t) = x(t) + \Delta t \cdot v(t)$$

Runge-Kutta integration schemes on the other hand are more stable for values of Δt in the order of 10^{-2}. The main drawback is the computational complexity, in the second-degree Runge-Kutta integrator we need to evaluate the forces on a particle 2 times for each time step, while in the fourth-degree Runge-Kutta, we need to evaluate the forces 4 times per time step. As an example, the second-degree Runge-Kutta integrator can be computed as follows [4]:

$$a_1 = v(t)$$
$$a_2 = F(t)/m$$
$$b_1 = v(t) + \frac{\Delta t}{2} a_2$$
$$b_2 = f(x(t) + \frac{\Delta t}{2} a_1, v(t) + \frac{\Delta t}{2} a_2)/m$$
$$x(t+\Delta t) = x(t) + \Delta t \cdot b_1$$
$$v(t+\Delta t) = v(t) + \Delta t b_2$$

where:

- a_1 is the current velocity (velocity at time t)
- a_2 is the current acceleration (velocity at time t)
- b_1 is the velocity of the next half time step (velocity at time $t + \Delta t/2$)
- b_2 is the acceleration of the next half time step (acceleration at time $t + \Delta t/2$)
- Note that $f(x(t) + \frac{\Delta t}{2} a_1, v(t) + \frac{\Delta t}{2} a_2)$ is the force computed at position $x(t) + \frac{\Delta t}{2} a_1$ considering velocity: $v(t) + \frac{\Delta t}{2} a_2$ or in other words $F(t + \Delta t/2)$

In this project, we will implement the semi-implicit Euler integrator, which is a slightly more stable version of the explicit Euler integration scheme. The choice of this integration scheme is due mainly to the high number of particles in our cube model.

If we consider a cube with n particles on its edge it means that the total number of particles will be n^3, which in turn translates in a complexity of $\theta(n^3 + m)$ (where m is the number of springs in the system) since we need to cycle through all particles and all springs to compute forces, velocities and positions.

It's also important to consider that we might want to run these computations multiple times per time step before drawing a frame. Since Runge-Kutta of 2nd/4th degree evaluates the forces 2/4 times per step, it's easy to see why semi-implicit Euler has been chosen.
in semi-implicit Euler, velocity and position are computed as follows:

$$v(t+\Delta t) = v(t) + \Delta t \cdot F(t)/m$$
$$x(t+\Delta t) = x(t) + \Delta t \cdot v(t+\Delta t)$$

4.2 Simulation algorithm

The pseudo code for the simulation algorithm using the Euler semi-implicit integration scheme is as follows:

Algorithm 1 Semi-implicit Euler Integration

```
$vardeltaT = 0.001$;                                   ▷ time step for simulation
$varframeR = 60$;                                      ▷ Frame rate in FPS
$num_substeps = 1.0/60/deltaT$;                        ▷ number of computed steps before updating the scene
for each particle p do
    $initialize\_velocity$
    $initialize\_position$
    $initialize\_mass$
end for
$generate\_springs$
while Simulating do
    for (int $i = 0$; $i < num\_substeps$; i++) do

        for each particle p do
            $initialize\_forces\_on\_particle$          ▷ gravity force + external forces
        end for

        for each spring s do
            $compute\_spirng\_force$                    ▷ compute spring force exerted on the particles
            $updeate\_connected\_particles\_forces$     ▷ add spring force to particles connected by it
        end for

        for each particle p do
            $compute\_collision\_force(p)$              ▷ compute collision force on particle if any
        end for

        for each particle p do
            $p.vel = p.vel + \Delta t \cdot p.force/p.mass$   ▷ update particle velocity
            $p.pos = p.pos + \Delta t \cdot p.vel$            ▷ update particle position
        end for
    end for
    $draw\_frame()$
end while
```

5 Collisions

Now that we have the basis for our simulation, it's time to implement collision. Collision is usually a two-step process, the first one is called collision detection, in which we want to find out if two objects are intersecting/colliding. The second step is determining the collision response (or solving the collision).

The collision detection method might differ depending on the shapes that are intersecting, we could in theory implement collision considering only triangles and create a universal collision detection algorithm since all our objects are made up of triangles. The problem with this approach is that it becomes computationally expensive quite fast depending on the number of triangles that make up the colliding objects.

There are several ways to implement collision response, mainly:

Projection methods: resolve the collision by modifying the position of the intersecting objects.
Impulse methods: resolve the collision by modifying the velocity of the intersecting objects.
Penalty methods: resolve the collision by modifying the acceleration/force of the intersecting objects.

When simulating a mass-spring system like our gelatin cube the most popular method and the one that we will implement is a penalty method that uses springs to modify the colliding particle's force based on how deep the particle has penetrated into the object. Using a projection or an impulse method in our case can cause problems since modifying directly the velocity or position of the colliding particles without considering the acceleration or the forces can lead to the introduction of unrealistic forces due to the springs connecting the particles, this can potentially increase the instability of our system.

5.1 Plane collision detection

The main idea of plane collision detection is to check if the distance between the particles of our cube and the plane is less or equal to a threshold value.
This is because if we check for the exact collision of the particle with the plane, we might miss it (**Figure 7**). If for example at time t a particle P is moving towards a plane with a high velocity $\vec{v}$, it is possible that at time $t + \Delta t$ (where Δt is the minimum time step of the simulation), the particle will already be on the other side of the plane and we won't be able to detect the collision. Using a Threshold distance value between the particle and the plane to detect collision will allow us to mitigate this problem.

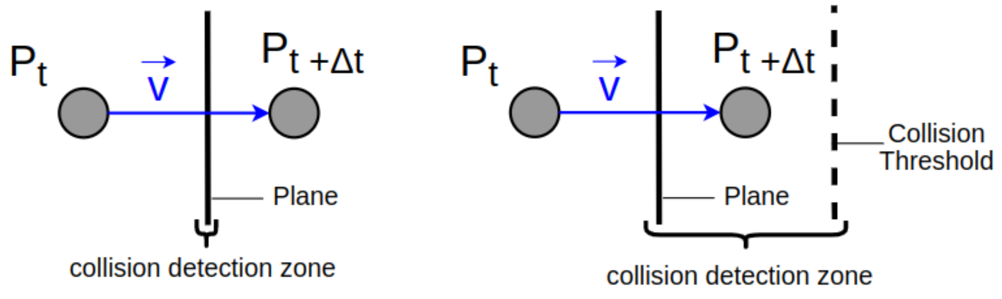


Figure 7. Plane collision without distance threshold(left) and with distance threshold (right)

The first step in collision detection is to create a ray that originates from the particle P with direction $\vec{d}$, where d is the inverse of the plane's normal. The ray $y = t\vec{d} + P$ where t is an unknown scalar, intersects the plane at point X iff $\langle X - Q, \vec{n} \rangle = 0$. We can then replace point X with the ray equation and find the value of the scalar t.

$$\langle t\vec{d} + P - Q, \vec{n} \rangle = 0$$

$$t = \frac{\langle q - p, n \rangle}{\langle d, n \rangle}$$

Before continuing, it's important to note that if the numerator $\langle q-p, n\rangle$ is equal to 0 it means that the particle P is already on the plane, and if the denominator $\langle d, n\rangle$ is 0 then it means that the direction d is perpendicular to the plane normal $\vec{n}$, meaning that the ray doesn't intersect the plane unless the nominator is 0. The sign of the scalar t also offers a very useful bit of information that we will use later, that is if $t < 0$ it means that the particle is past the plane and if $t > 0$ the particle is over the plane (this is true only if $\vec{d} = -\vec{n}$).
Now that we have the value of t we can find the intersection point X with the ray equation. Since a plane in our simulation is finite, before computing the distance between the particle and the plane ($distance = ||x-p||$), we need to check if the intersection point X is inside or outside the finite plane region. To do so we can simply check if the coordinates of X are bigger or equal to the minimum coordinates of the plane and less than or equal to the maximum coordinates of the plane. (The minimum coordinates of the plane are the smallest possible coordinates values (x,y,z) of a point that is on and inside the finite plane, similarly the maximum coordinates of the plane are the highest possible coordinates values (x,y,z) of a point that is inside and on the finite region of the plane. Each of these two points corresponds to a corner of the finite plane).
If X is in the plane we can compute the vector $\vec{PX}$ by simply plugging the value of t in the equation $y = td + P$, thus finding the value of $\vec{y}$. The length of $\vec{y} = \vec{PX}$ is the distance of our particle from the plane.
Thus a collision is detected if:

- The ray-plane intersection point exists.
- The point is within the region of the finite plane.
- The distance between the particle P and the intersection point X is less or equal to threshold value.

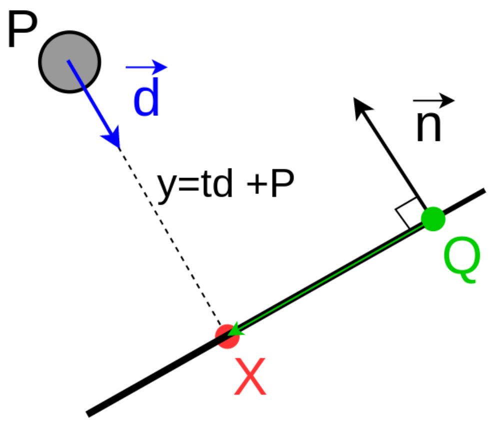


Figure 8. Ray-plane intersection

5.2 Plane collision response

As anticipated in the previous sections the collision response is modeled with the force produced by a spring where the elongation is the distance from the plane to the particle that has penetrated it (see **Figure 9**). The maximum distance is the collision detection threshold value. The stiffness of the spring must be set with trial end error, if it's too stiff the object will bounce and if it's not stiff enough the object might fall partially through the plane and then resurface or fall completely through it.

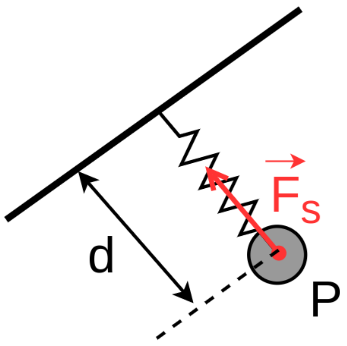


Figure 9. Ray plane intersection

The collision spring force magnitude will thus be $\vec{F_s} = k \cdot d$. If we consider an infinitely rigid plane, the direction of the collision spring force will be the normal of the plane. Thus the collision spring force vector is: $\vec{F_s} = k \cdot d \cdot \vec{n}$ (where the plane normal $\vec{n}$ is a normalized direction). To this force we still need to add the component of the particle's force ($\vec{F_p}$) that is normal to the plane. If we don't do so the cube might partially penetrate the plane and it will also take a very long time to reach equilibrium and stop oscillating. The total reaction force vector will thus be:

$$\vec{F_R} = (k \cdot d + \|\langle \vec{F_p}, \vec{n} \rangle\|) \cdot \vec{n}$$

It's important to notice when exactly we want to apply the reaction force and its components:

- F_R is computed and applied only when a collision is detected and the particle is either inside ($t < 0$) or on the plane ($t = 0$) (see chapter 5.1)
- $\langle \vec{F_p}, \vec{n} \rangle$ is applied only if the component of the particle's force that is normal to the plane is going into the plane which means that $F_p \cdot \vec{n} < 0$

5.3 Plane collision detection and response implementation results

As shown in **Figure 9** implementing the methods for plane collision detection and response discussed in the previous sections leads us to some nice collisions.

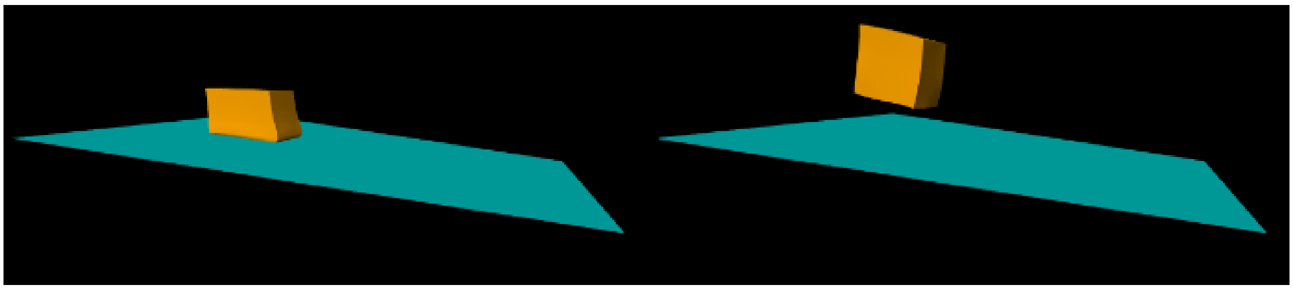


Figure 10. Plane collision detection and response implementation results: cube dropped on the plane getting squished as it lands on it (left), cube bouncing off the plane after collision (right)

5.4 Sphere collision detection

Sphere collision detection is easier than a plane but it works in a similar fashion. All we have to do is check if a particle of the cube is inside the sphere. We can do so by computing the length of the vector that connects the center of the sphere with the particle of the cube. If the length of the vector is less or equal to the ray of the sphere we know the particle is colliding with the sphere.

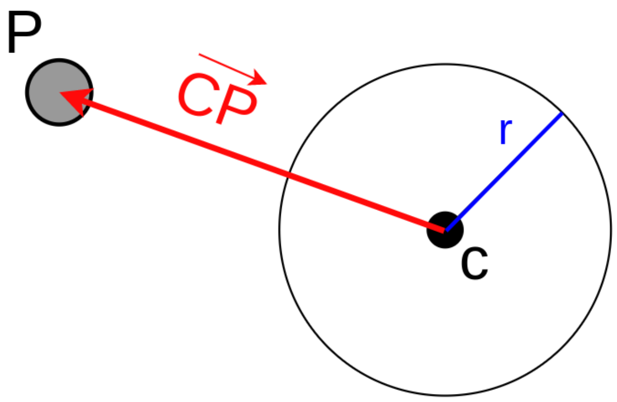


Figure 11. Sphere collision detection

Thus we collide if:

$$\|\vec{CP}\| = \|P - C\| <= r$$

5.5 Sphere collision response

The sphere collision response can be implemented in a very simple yet effective way, which is similar to the plane response.

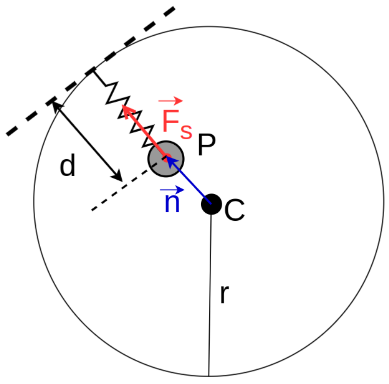


Figure 12. Sphere collision detection

The total reaction force $\vec{F_r}$ to apply to the colliding particle is computed in the same way as for the plane:

$$\vec{F_R} = (k \cdot d + \|\langle \vec{F_p}, \vec{n} \rangle\|) \cdot \vec{n}$$

The only thing that changes is how we compute the distance d and the normal direction vector $\vec{n}$. For the normal direction, we want to find the normal of the sphere surface at the point on which the particle collided. To compute it we need to store the position of the particle before the collision P_{prev} and then we can use the ray sphere intersection formula to find where the particle collided on the surface of the sphere. The direction of the ray will simply be the vector that goes from the particle at the previous step P_{prev} to the current position of the colliding particle P. For clarity, the situation is illustrated in **Figure 13**:

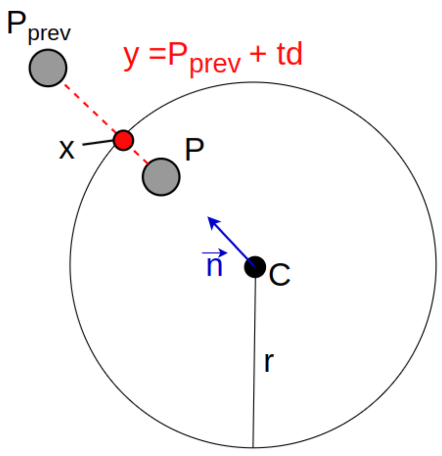


Figure 13. Ray Sphere intersection

We can use the ray-sphere intersection formula to find the collision point x:

$$\|P_{prev} + t\vec{d} - c\|^2 - r^2 = 0$$

where $\vec{d} = P - P_{prev}$. This equation can be solved for t with the 2nd degree equation formula using the following parameters:

$$a = \langle \vec{d}, \vec{d} \rangle$$

$$b = \langle 2\vec{d}, (P_{prev} - c) \rangle$$

$$c = \|P_{prev} - c\|^2 - r^2$$

The possibles values of t will then be:

$$t_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

In the case that t has more than one possible solution we want of course keep the smallest positive t value. The collision point on the surface will then be:

$$x = P_{prev} + t\vec{d}$$

The collision normal direction can be found as follows:

$$\vec{n} = \frac{x - c}{\|x - c\|}$$

The distance of the particle from the surface can be found by solving another ray sphere intersection in which the ray starts from P and has direction n we won't repeat the math since the process is the same. It's important to notice that here the distance d is not the shortest possible distance of the particle P from the surface but it's the distance from the surface that the particle has to cover along the direction of the when "pushed" by the collision reaction force

5.6 Sphere collision detection and response implementation results

Figure 14 shows the results obtained by implementing sphere collisions as explained in the previous sections.

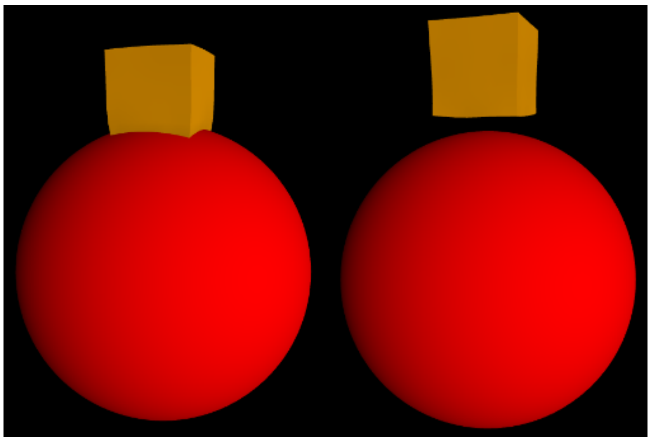


Figure 14. Sphere collision detection and response implementation results: cube dropped on the sphere getting squished as it lands on it (left), cube bouncing off the sphere after collision (right)

6 Friction

Now that collisions have been implemented we might notice that our cube behaves unrealistically. For example, while colliding with the XZ plane, the cube particles have a hard time to stop oscillating along the x and z directions. Another example would be that no matter what we do if we position our gelatin cube on an inclined plane, it will always slip and fall off the plane. These problems are due to the lack of friction in our simulation. In this section, we will see how to implement friction in order to obtain a more realistic simulation.

6.1 Inclined plane friction

Let us consider an object (our cube's particle) on an inclined plane, subjected to a generic force $\vec{F}$, this force might be for example the sum of all forces acting on our particle ($\vec{F} = \vec{F}_{springs} + \vec{F}_{gravity} + ...$). We can then decompose the force $\vec{F}$ into two components such that:

$$\vec{F} = \vec{F}_p + \vec{F}_n$$

where $\vec{F}_p$ is the force component that is parallel to the surface of the inclined plane and $\vec{F}_n$ is the force component that is perpendicular to the plane surface. The described scenario is shown in **Figure 15**.

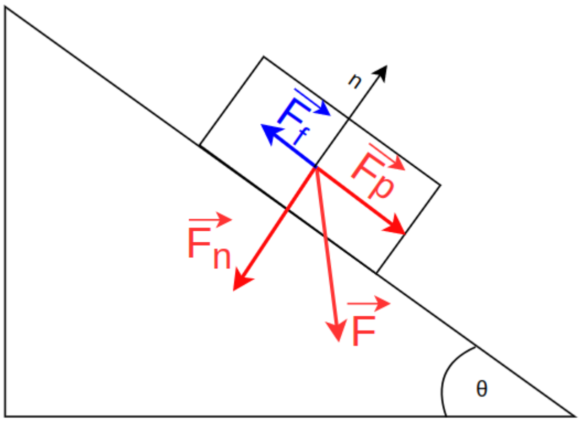


Figure 15. Friction of an object on an inclined plane

Given that the force $\vec{F}_n$ is directed towards the plane, if the object is moving with velocity $\vec{v}$, the friction force $\vec{F}_f$ acting on our object can be computed as follows:

$$\vec{F}_f = F_N \cdot \mu_d \cdot (-\vec{v}_{p\_dir})$$

Where:

- $F_N = \langle \vec{F}, \vec{n} \rangle$ is the magnitude of the force component acting on the object that is normal to the plane (this force must be directed towards the plane $\langle \vec{F}, \vec{n} \rangle < 0$).
- μ_d is the dynamic friction coefficient between the plane and the object, his value depends on the material of the plane and the object and on the condition of the surfaces that are in contact with each other (eg. roughness).
- $\vec{v}\_{pdir}$ is the normalized velocity vector of the object which is parallel to the surface of the plane. The inverse direction $(-\vec{v}_{p\_dir})$ is the direction in which dynamic friction affects the object. we can compute the velocity component parallel to the plane as follows:
$$\vec{v}\_{pdir} = \frac{\vec{v} - (\|\langle \vec{v}, \vec{n} \rangle\| \cdot \vec{n})}{\|\vec{v} - (\|\langle \vec{v}, \vec{n} \rangle\| \cdot \vec{n})\|}$$

Where $\vec{v}$ is the velocity of the object

Note that what we have just computed is **dynamic friction** which affects the object only when said object is moving and there is a force applied to it which has a component directed towards the plane. If we consider the same example showed in **Figure 15** but with a still object ($\vec{v} = 0$), we can similarly compute the friction force:

$$\vec{F}_f = F_N \cdot \mu_s \cdot (-\vec{F}_{p\_dir})$$

Where:

- μ_s is the static friction coefficient between the plane and the object, its value depends on the material of the plane and the object and on the condition of the surfaces that are in contact with each other (eg. roughness) and it's usually higher than his dynamic counterpart.
- $\vec{F}_{p\_dir}$ is the normalized force component vector of the object that is parallel to the plane, and the inverse direction is the direction in which static friction affects the object

Generally, if we consider an object on a flat surface, under the influx of gravity, when a force that would induce a movement of the object is applied, we have static friction. The friction force is thus computed using the static friction coefficient and his direction is the one opposed to the component of the applied force that is parallel to the surface. When the object starts moving, the friction coefficient that must be considered is the dynamic one while the direction is the inverse of the velocity component that is parallel to the surface.

When implementing friction it's important to notice that we want to compute and apply it only to the cube's particles that are colliding with an object.

6.2 Inclined plane friction implementation results

Our cube is now able to stop slipping on the plane thanks to friction. However this is difficult to show in a Figure, thus we must ask the reader to trust the fact that the cube laying on the plane showed in **Figure 16** is not moving nor slipping thanks to friction. Alternatively the reader is welcome to test this himself.

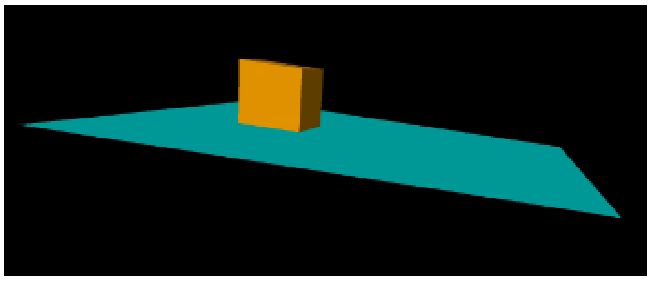


Figure 16. Inclined plane friction implementation: the cube stopped slipping on the plane thanks to friction

6.3 Sphere friction

Friction on the sphere surface works exactly in the same way as for an inclined plane, thus in this section, we will explain the main idea of why that's the case, but we won't repeat all the computations involved since they are the same shown in **Section 6.1**.

To compute friction the only geometric-dependent information we really need to know is the normal direction of the contact surface. If we consider a perfect sphere colliding with an object with perfect surfaces, we will have only one point of contact, this point is the point on which our cube particle collide with the surface of the sphere. We can thus compute friction as we did for the plane, the only difference is that we have to compute the normal direction at every step during collision since it depends on where the particle is colliding with the sphere.

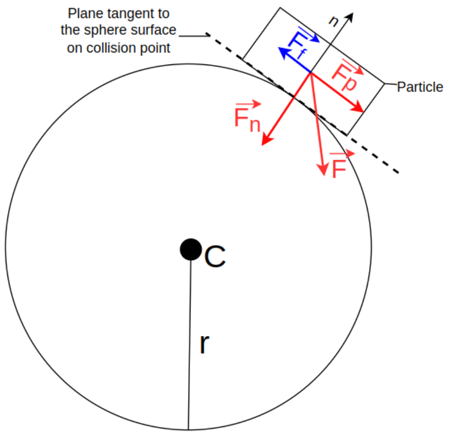


Figure 17. Friction of an object on spherical surface

6.4 Sphere friction implementation results

By implementing friction we can see that the cube doesn't slide uncontrollably anymore and the colliding particle is affected by the friction force.

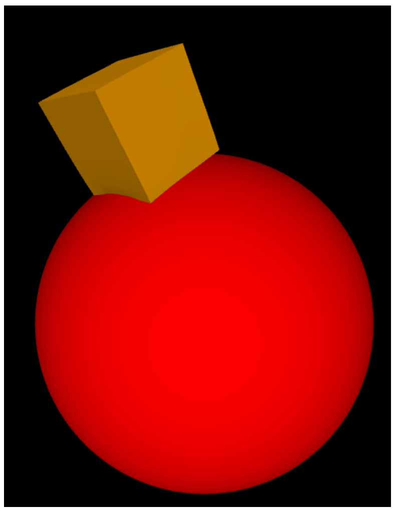


Figure 18. Sphere friction implementation results: the cube's left side deforms and the cube starts to rotate due to friction as it falls off the sphere

7 Bounding volume (BV)

Checking for collisions by computing the distance between each cube particle, and every other object at each simulation step is computationally very expensive since it would mean that we need to compute distances at least: $\theta(n^3 * k)$ times, where n is the number of particles that make up our cube and k is the number of objects we have in our simulation (jelly cube excluded). But do we need to compute distances to check for collisions at every step?
The answer is of course not, we only need to check for example when the jelly cube is in the proximity of another object. To do so we use a BV.

The main idea of BV is to consider invisible primitive solids (sphere, cube, cone,...) in which our objects will reside. (It's important to note that the primitive solids orientation in the space doesn't change even if the object is rotating or stretching and their volume is usually slightly bigger than the object/s it contains). We then check for collisions between the primitive solids or bounding volumes first. If we detect a collision between the cube's BV and another object's BV, we check again for collisions between the particles of the cube and the object. If no bounding volume collision is detected, we can instead update the cube's particles directly, without running a collision check for every single particle, since we know there are none. **Figure 19** shows a graphical example of collision detection using bounding volumes. On the left we check for collisions between bounding volumes and since there are none, we don't need to make another check using the cube's particles. On the right we check for collisions using bounding volumes and since the two volumes intersect each others (collision between BVs detected), we make an additional collision check between the particles of the cube and the plane.

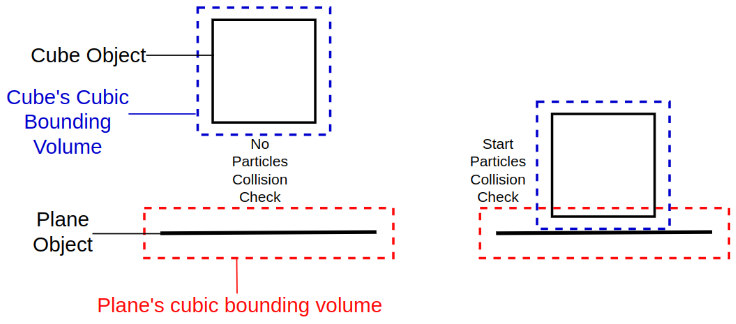

Cube Object
Cube's Cubic Bounding Volume
No Particles Collision Check
Plane Object
Plane's cubic bounding volume
Start Particles Collision Check

Figure 19. Bounding volumes collision check example

7.1 Bounding volume implemetation

In our case, since the gelatin cube is a moving deformable object, that can stretch and contract, we use a relatively large bounding box, to guarantee that the cube will be always inside it. We also put bounding volumes (or boxes since they are cubical volumes), on all the other undeformable objects.

Here is how we use bounding volumes in our simulation:

1. At each simulation step before updating the cube particles positions, we clear all the bounding volume arrays of all our solid objects. Note that this BV array contains all the indexes of the jelly cube's particles that are inside the object's bounding volume. We also clear the B.V array of the gelatin cube. This bounding volume array works a little bit differently from the others since it stores the solid objects that are intersecting the bounding volume of the gelatin cube.

2. After we compute the new positions of the cube's particles, and update the cube's bounding volume location, we check if the cube's bounding box is intersecting any other object's bounding box. If that's the case we add the object to the cube's bounding volume array.

3. The next step is to loop over the content of the cube bounding volume. For each object inside it, we check if there are cube particles that are inside the rigid object's bounding volume. If that's the case, we add the particle's index to the object's bounding volume array.

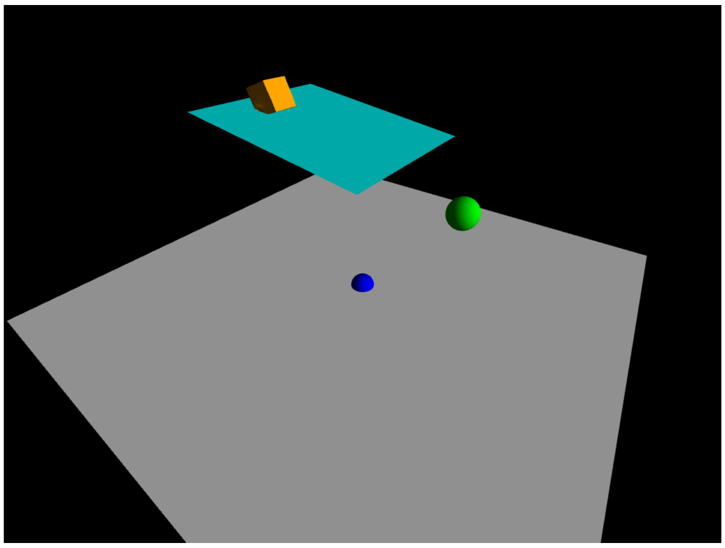
4. Finally, when computing the forces acting on the cube's particles, we check for collision only the particles that are inside an object's bounding volume array. If a collision is detected, we compute the collision reaction forces as we did before.

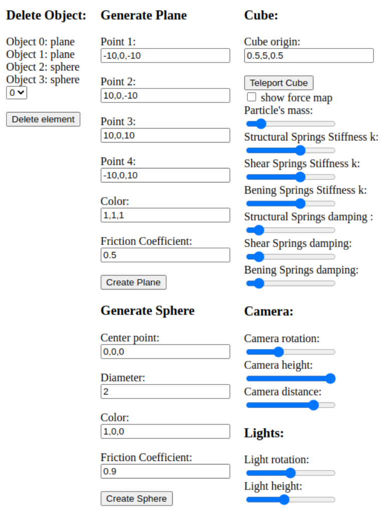
It's easy to see that this implementation of bounding volumes allows us to speed up collision computations quite a bit by checking object proximity using primitive solids instead of computing distances between all the n^3 particles and all the other objects when the cube and the objects are far apart.
Once the objects are in close proximity to the cube, we evaluate collisions in more detail using particles and distances. We then compute the collision reaction forces considering only those particular particles that are colliding with the objects that are in the cube's proximity.
This leads us to a complexity of $\theta(n^3 \cdot k_{near})$, where k_{near} is the number of objects that are in close proximity to the cube. At first glance, it might not seem like a big improvement, but indicatively, before using bounding volumes, our system would struggle to handle a cube made of 512 particles (8x8x8) and six rigid objects. Now, it can handle a cube made of 1000 particles (10x10x10) with the same number or higher of rigid objects.
This system works quite well with the condition that there aren't too many objects close to each other and to the cube at the same time.

8 GUI

The GUI of this simulation is divided into three columns, and it allows the user to modify the cube proprieties and build their own simulation as follows:

- The first chunk of the 3rd column starting from the left contains the cube's controls. In this section, we can provide an origin for the cube and then teleport it by pressing the "teleport cube" button. Additionally, we can enable the cube force map or/and modify some of the cube proprieties such as the particle's mass, the stiffness and the damping of Structural, Shear and bending springs.
- In the same column, underneath the cube controls, we can find some controls for adjusting the camera and the directional light.
- The second column is dedicated to the generation of rigid objects that can be added to the simulation. To generate an object, the user provides the indicated parameters and presses the corresponding create object button. Some default values for each object are provided, to show how the parameters work
- The first column to the left is for deleting objects from the simulation. It shows a list of the objects that are currently in the simulation and the user can select the index of the object to delete from the drop-down menu and then press delete object in order to delete it. (The list of objects is automatically updated when an object is deleted)





Delete Object:

Object 0: plane
Object 1: plane
Object 2: sphere
Object 3: sphere
0

Delete element

Generate Plane

Point 1: -10,0,-10

Point 2: 10,0,-10

Point 3: 10,0,10

Point 4: -10,0,10

Color: 1,1,1

Friction Coefficient: 0.5

Create Plane

Generate Sphere

Center point: 0,0,0

Diameter: 2

Color: 1,0,0

Friction Coefficient: 0.9

Create Sphere

Cube:

Cube origin: 0.5,5,0.5

Teleport Cube

☐ show force map

Particle's mass:

Structural Springs Stiffness k:

Shear Springs Stiffness k:

Bening Springs Stiffness k:

Structural Springs damping :

Shear Springs damping:

Bening Springs damping:

Camera:

Camera rotation:

Camera height:

Camera distance:

Lights:

Light rotation:

Light height:

Figure 20. Simulation's Graphical User Interface

It's important to note that this simulation allows the user to experiment with parameters to intuitively understand the limitations and drawbacks of the methods that have been implemented. It should also offer the possibility to understand how the parameters of the springs combined with the cube's particle mass will affect the behavior of the cube.

9 Cube force color map

To get a better understanding of the forces within the cube, we can visualize them with the help of a color map. The implementation is quite simple, we already have the value of the force affecting each particle, we can simply convert those value to a color, store it and pass it to the fragment shader to be visualized. This needs to be done at each step of our simulation in order to update the cube colors in real-time. The color map is showed in **Figure 21**

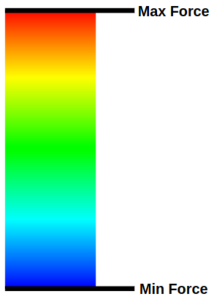


Figure 21. Cube force map while colliding on sphere

The implemented color map applied to the gelatin cube is shown in **Figure 22**

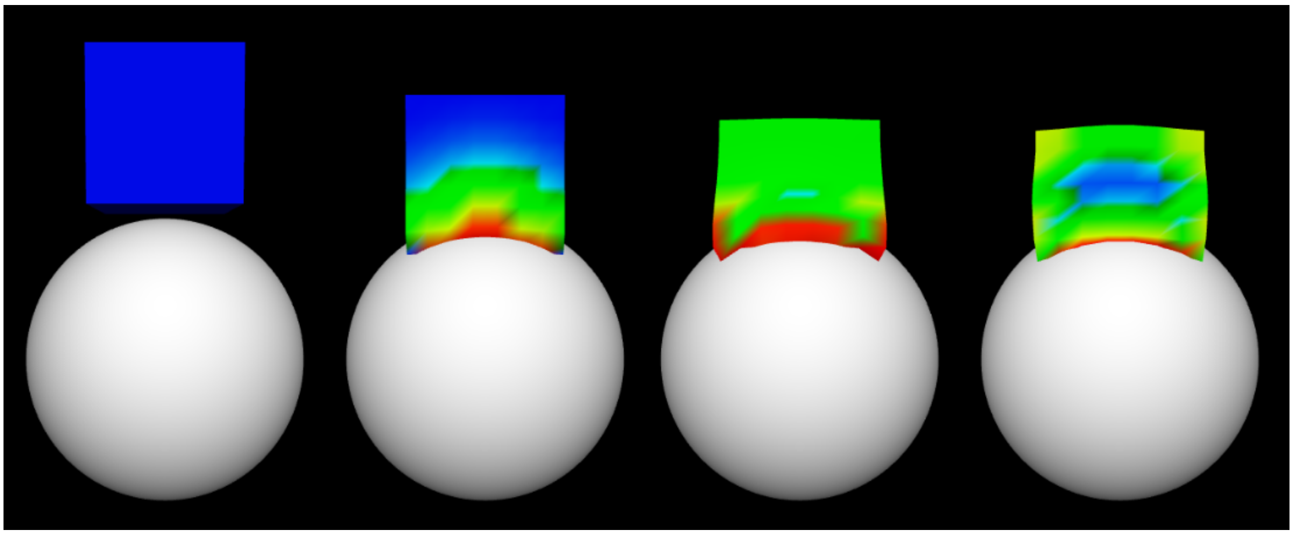


Figure 22. Cube force map while colliding on sphere

10 Limitations

As already mentioned in the previous sections, this simulation has its limitations. The most noticeable is that our gelatin cube may break in different ways. This is due to a couple of reasons. The first one is the use of the Euler integration scheme which can sometimes cause the simulation to become unstable if the simulation's time step is too big to accurately approximate the physical computations. In the simulation this happens only if the values of the damping coefficients of the springs are high with respect to the mass of the particles. The issue can be solved either by picking a lower time step (this will make the simulation slower) or by simply increasing the particles mass. An example of this instability is shown in **Figure 23**.

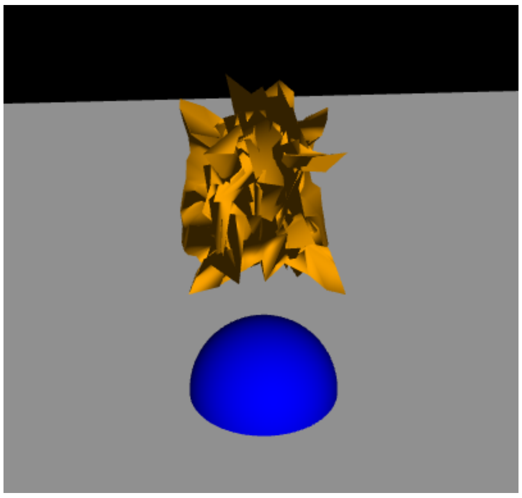


Figure 23. Instability of the Euler method when the damping coefficient is too high with respect to the mass of the cube's particles

The second one is due to the fact that there is nothing that restrains the displacement and rotation of both the particles and the springs, thus if the stiffness of the spring is not high enough is it possible to break the cube. This issue can also be mitigated by adjusting the parameters of the cube such as springs stiffness, damping and particles mass. This limitation is shown in **Figure 24**:

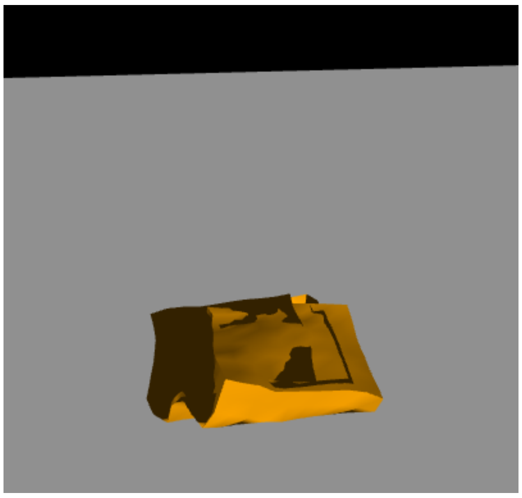


Figure 24. Broken cube due to springs with a too low stiffness

11 Extending the cube model

So far, we have obtained a fairly accurate simulation of the physical behavior of our gelatin cube, but wouldn't it be great to be able to build entire objects made of gelatin cubes?
In this section, we will see a simple approach that will allow us to connect multiple gelatin cubes together in order to form different "Minecraft" style shapes. The first thing we need is to convert what we coded so far into classes, in order to create multiple gelatin cubes, since this process is quite long and tedious we will skip this part and focus more on how to connect the cubes.

11.1 Cubes connections

There are several ways to connect the two cubes, we will adopt a convenient approach that takes advantage of the cube structures and minimizes the number of particles and springs. The main idea is simply to merge the particle of the two connecting faces together. The only problems that remain are that we have to connect the particles of the cube whose face particles were replaced with springs, but we also have to connect some particles between the two cubes. The first problem is easily solved by the cube class, our cube is made of particles and faces, and once we have replaced the particles of one face, we can regenerate the springs of the cube whose particles were replaced. This will reconnect all the particle of the cube, and by proxy, it will also connect the two cubes, since now the particles of the connected faces are exactly the same. This solved our problem, but a new one arises, now we have duplicated the springs on the shared face, to fix this we can simply remove the springs that connect 2 shared particles in one of the two cubes.
The only thing that remains to do is to generate the bending springs that connect the particles that are on the "inner" faces of the two cubes. **Figure 25** shows how the cubes are connected:

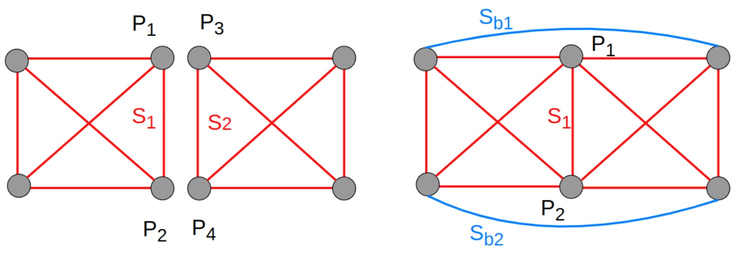


Figure 25. Cubes with particles and springs before merging (left) and after merging (right)

Notice that in order to merge the two cubes, we had to delete particles P_3 and P_4 from the cube on the right, but we also had to remove spring S_2 and add the two bending springs S_{b1} and S_{b2}. This is the simplest merging case possible but keep in mind that for two cubes with more particles, duplicated shear and bending springs must also be deleted from one of the two cubes.

11.2 Cube Grid

So far, we have seen how to connect the cubes but how do we know if we have to connect two cubes and which faces we have to connect the answer is simple we use a 3D grid. We can define a grid in space by defining two points, G_{min}, G_{max} and the number of desired subdivisions, obtaining the following grid (showed in 2D just for the sake of simplicity):

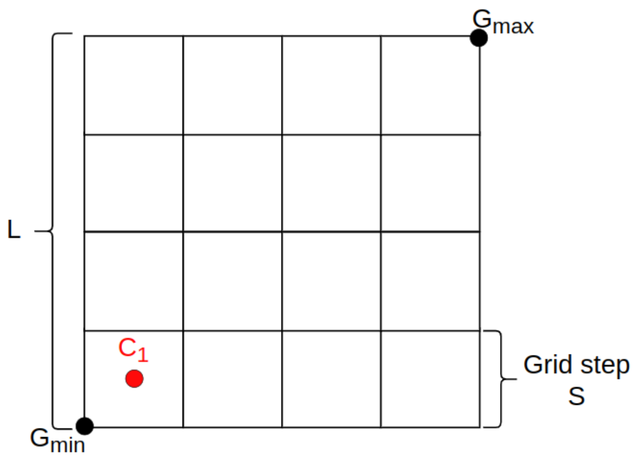


Figure 26. Grid

This grid definition allows us to define the size and positions of multiple cubes in a "Minecraft" style using the size of the grid step S and the centers of the subdivisions to position our cubes. We can then easily check if we have to merge them, by simply checking if a cube has a neighbor in one of the adjacent cells (left, right, top and bottom in the 2D case). We can also create objects made of cubes in an easier way by using grid indices instead of coordinates. For example, the index of the cube which corresponds to a cube with size S (the grid step) whose center is located at the center of of the grid subdivision C_1 would be $index = (1, 1)$. We can convert indices to subdivisions center points as follows:

$$coord = G_{min} + \frac{S}{2} + index \cdot S$$

11.3 Cube grid and connected cube implementation results

The results of implementing the grid and connecting the cubes merging their faces are shown in **Figure 27**:

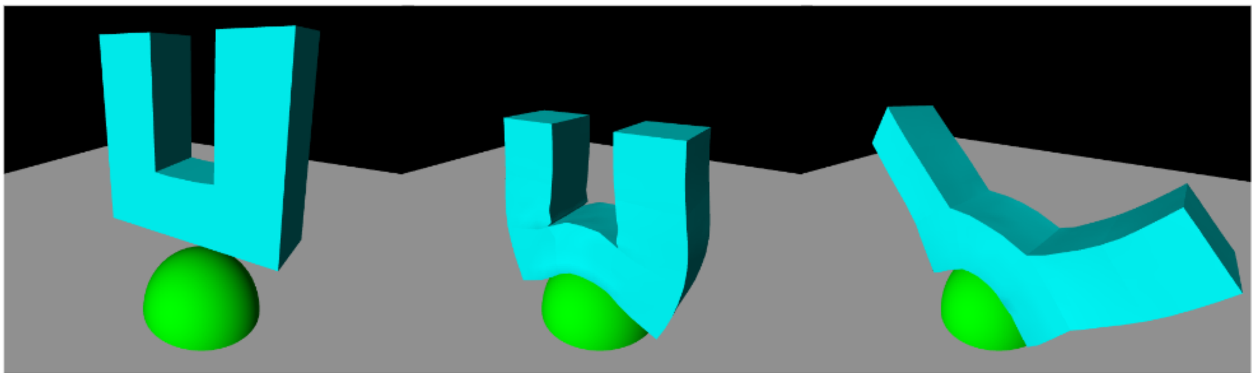


Figure 27. Cube grid and merging implementation results: the connected cubes form and behave like a single "U" shaped soft object that deforms as it falls on a rigid sphere

12 From 3D to jelly cubes structure

Now that we can build jelly cubes objects using a grid, the next step would be to import a 3D model from an obj file and use the grid to recreate it as a "Minecraft" style object made of gelatin cubes. This can be done with the ray triangle intersection in combination with a 3D grid, but we first need to parse the obj file and extract vertices and triangles. The obj parser function is provided in the code of the project, but it will not be explored in detail since it's not this project's focus.

12.1 Grid definition from model coordinates

Once we successfully imported the data we need from the obj file (points and triangles), we need to use it to define the grid. We can define the grid by going through all the x, y, z coordinates of all the points of the object, saving both the highest and the lowest coordinates we can find into two points: P_{max} and P_{min}. Once we have a point containing the maximum coordinates values of the imported object $P_{max} = (x_{max}, y_{max}, z_{max})$ and a point containing the minimum coordinates values of the imported object $P_{min} = (x_{min}, y_{min}, z_{min})$, we can use them to define the cubical grid as follows:

We compute the center of the grid:

$$G_{center} = \frac{P_{max} + P_{min}}{2}$$

We then define the length of the edges of our cubic grid by choosing the maximum dimension from the difference between the two points:

$$P_{diff} = P_{max} - Pmin$$

$$L = max(x_{P_{diff}}, y_{P_{diff}}, z_{P_{diff}})$$

We can then compute the starting point of our grid as:

$$G_{start} = G_{center} - \vec{D}$$

Where

$$\vec{D} = (L/2, L/2, -L/2)^T$$

The last thing that we need is the grid step which depends on how much detail we want in our soft object, for simplicity we will define this as a constant that can be adjusted manually should we need a finer or bigger grid step. We can for example set it at 8 which corresponds to 512 subdivisions. We thus defined a grid around our imported object as shown in **Figure 28** (G_{start} corresponds to G_{min} in the figure)

12.2 Voxelization using ray casting

Now that we have defined the grid and the subdivision, we have to decide in which of the subdivision we should put a jelly cube. In order to do so we use the ray triangle intersection to check if the center of a subdivision is inside or outside the object. This can be done by casting a ray from the center of the subdivision in a random direction and checking for intersections with the triangles of the model imported from the obj file. If the ray intersects an even number of triangles, it means that the grid subdivision's center point is outside the object and thus we don't want to place a cube there; if instead the ray intersects an odd number of triangles the center of the grid subdivision is inside the object and we want to place a cube there, **Figure 28** shows the general idea of the algorithm. we can see that the ray that starts from point C_9 intersects 2 triangles of the object, this means that C_9 is not inside the object and we don't want to place a cube there. The ray starting from C_1 intersects just one triangle of the object, this means that C_1 is inside the object and we want to place a cube there.

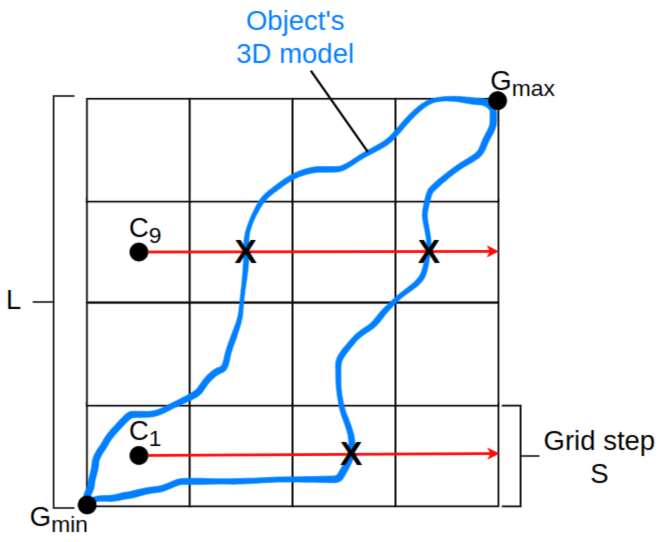


Figure 28. Grid subdivision's ray intersections with object model

Now in order to check the intersection point, we can create our ray equation in the following way:

$$y = C_i + t\vec{d}$$

Where C_i is the center point of the subdivision we want to check, and d is a random direction, we can for example pick $\vec{d} = [1, 0, 0]^T$. To find the intersection point with a triangle, we simply apply the ray-plane intersection with the plane on which the triangle lies on, and then if the intersection point exists, we check if it's inside or outside the triangle using barycentric coordinates. The ray-plane intersection won't be repeated since the process is the same as presented in section 5.1, we just need to find the plane normal which is the same as the triangle normal, that can be computed in the following way:

$$triangle\_nomral = \frac{(P3 - P1) \times (P2 - P1)}{||(P3 - P1) \times (P2 - P1)||}$$

Where the three points $P1, P2, P3$ are shown in **Figure 29**

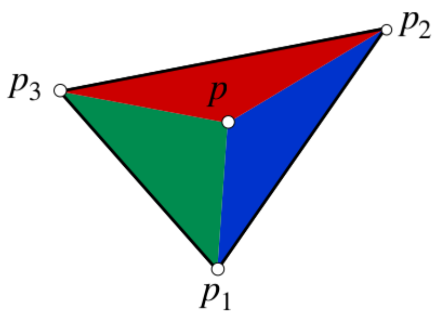


Figure 29. Triangle barycentric coordinates

It's important to notice that, when we solve the ray-plane intersection, if the intersection point exists, it's a valid intersection point only if the value of t is positive, meaning that the direction of the ray was the original one and not its opposite.

We can then check if our intersection point P is inside the triangle, to do so we need to compute the signed area W of the triangle $P1, P2, P3$:

$$W = \frac{1}{2}||n||$$

Where n is the cross product between the vector representing the edges of the triangles that generates a vector that is normal to the triangle's plane:

$$n = (P3 - P1) \times (P2 - P1)$$

The signed areas of the sub-triangles:

$$w_i = \frac{1}{2}||n_i|| \cdot sign(\langle n_i, n \rangle)$$

Where

$$n_i = (P_{i+1} - P) \times (P_{i-1} - P)$$

We then know that the intersection point P is inside the triangle only if these two conditions are respected :

1. All the signed areas w_1, w_2, w_3 are positive

2. The sum of the barycentric coordinates is 1:

$$\frac{(w_1 + w_2 + w_3)}{W} = 1$$

If one of these conditions doesn't hold the intersection point is outside the triangle and thus we don't want to put a jelly cube in that grid subdivision.

It's important to note that in the case our ray intersects a shared edge between two mesh triangles, the intersection will be counted twice (one per triangle) even if in reality we have just one intersection point. This is very likely to happen, especially for models with regular shapes like cubes. To fix this it's important to keep track of all the intersection points when checking a subsection center point with every triangle so that we can detect if we already encountered an intersection point or not and we will count only the intersections that happen in never seen before points.

12.3 From 3D model to jelly cube structure implementation results

In order to test our implementation and avoid copyrights infringements I used Fusion 360 to make a couple of 3D models that have been then exported as obj files. The first object is a simple structure made of cubes and the second is an elastic coupler. The implementation results of the frist object are shown in **Figure 30** and **Figure 31**

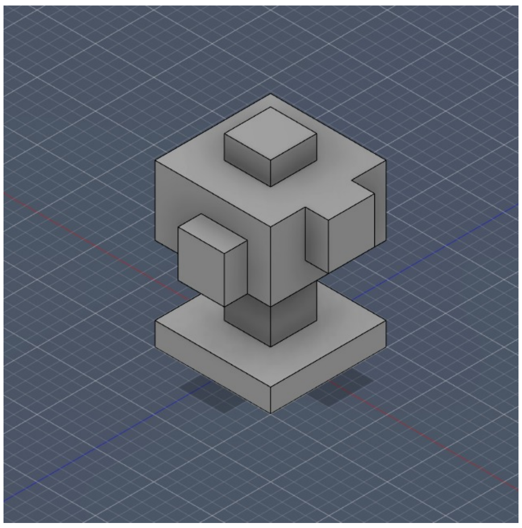


Figure 30. minecraft_tree.obj model in Fusion 360

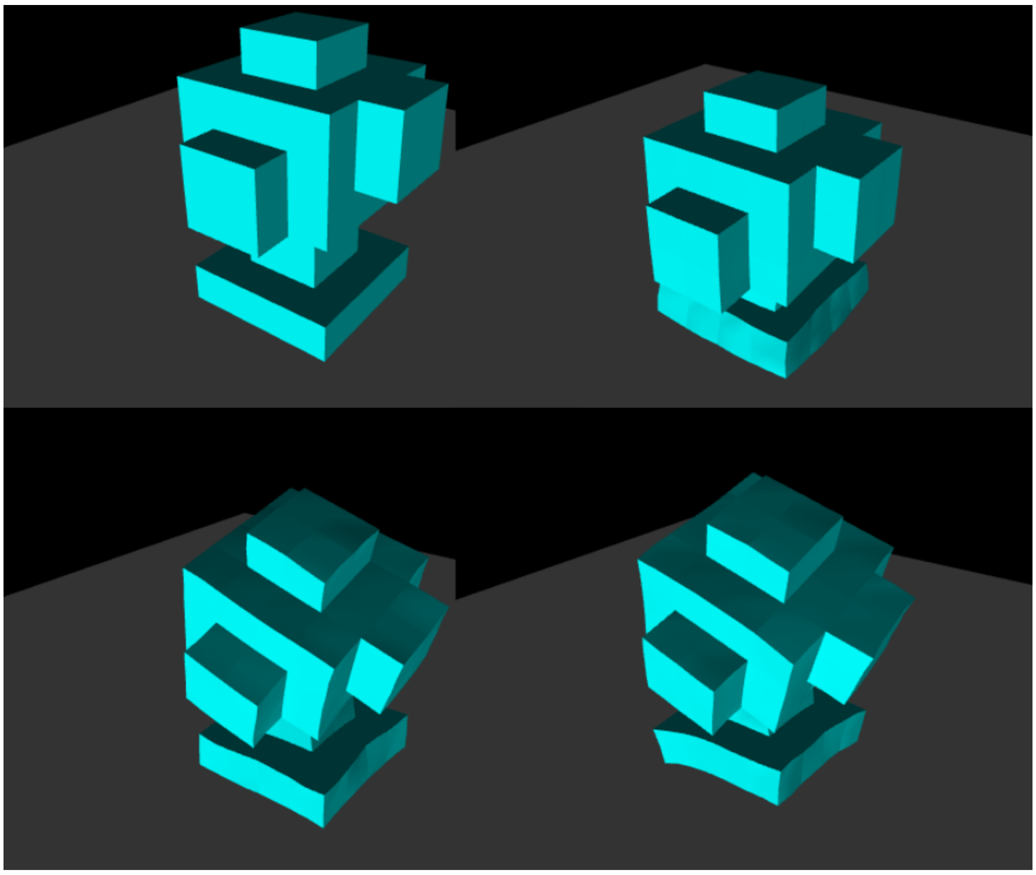


Figure 31. minecraft_tree.obj object recreated with multiple connected soft cubes that deforms as it falls on a plane

Cubic shaped models are not the only type of object that our implementation can "jellify" we can also do the same thing with more complex models like this elastic coupler that I modeled in fusion 360:

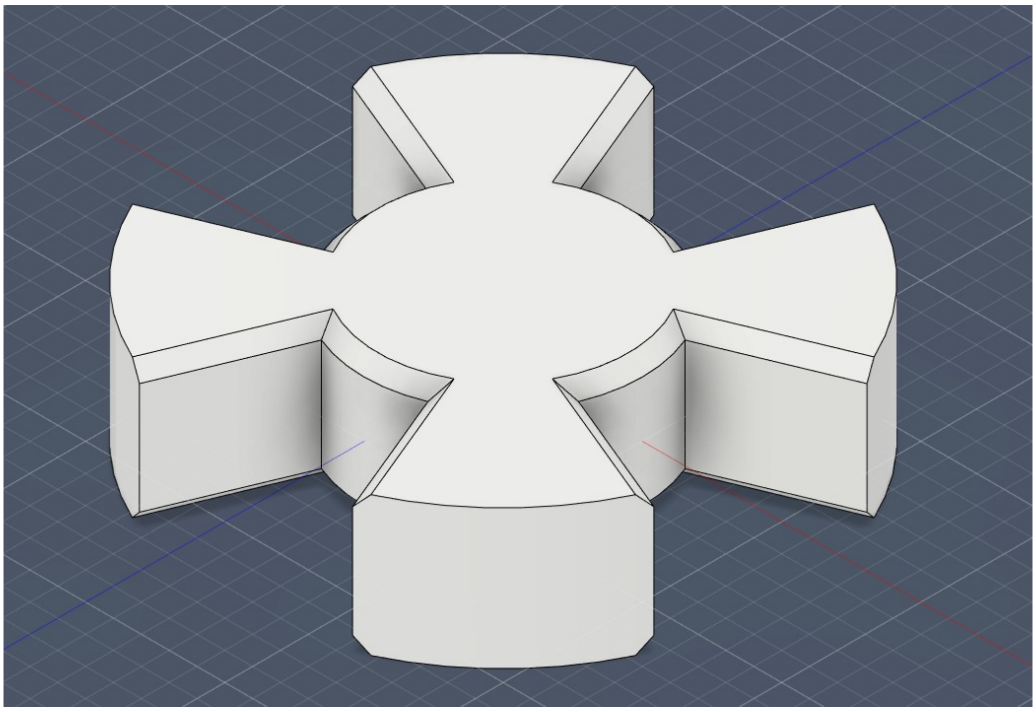


Figure 32. coupler.obj model in Fusion 360

The quantity of detail of the jellified object will, of course, depend on how many subdivisions the grid has. The following figures show different grid densities applied to the coupler.obj model.

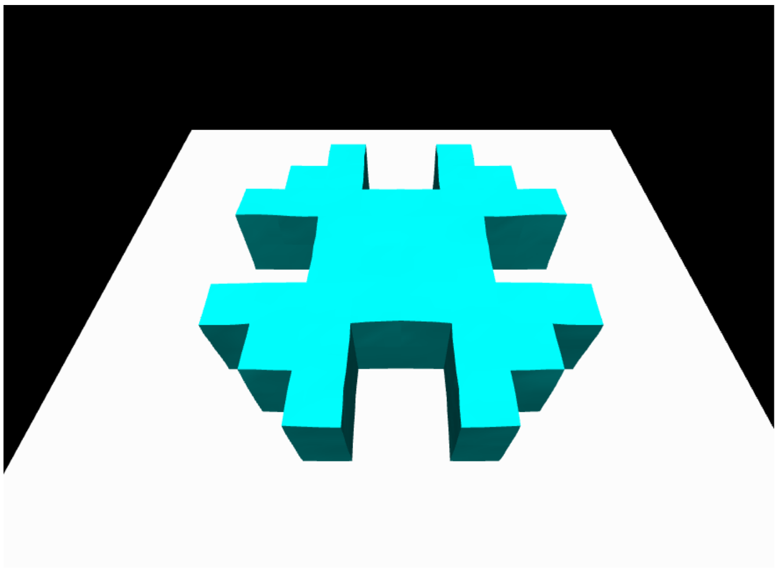


Figure 33. coupler.obj in a grid with 7x7x7 subdivisions

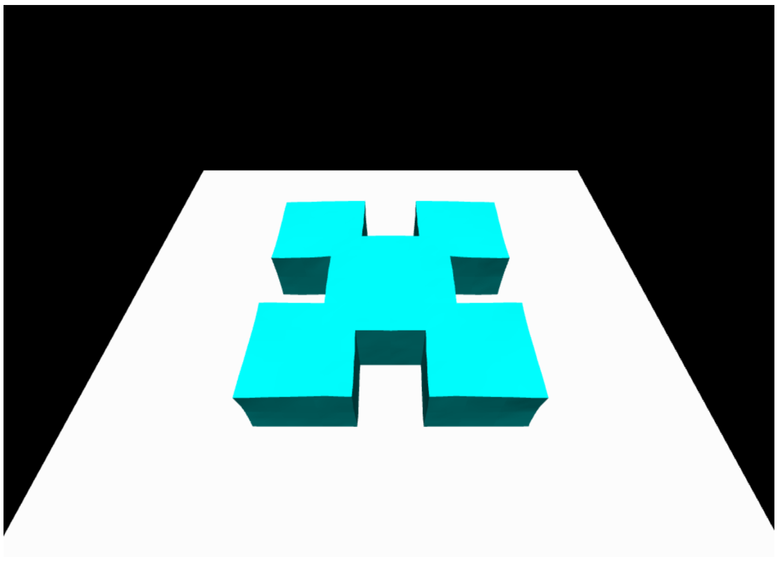


Figure 34. coupler.obj in a grid with 8x8x8 subdivisions

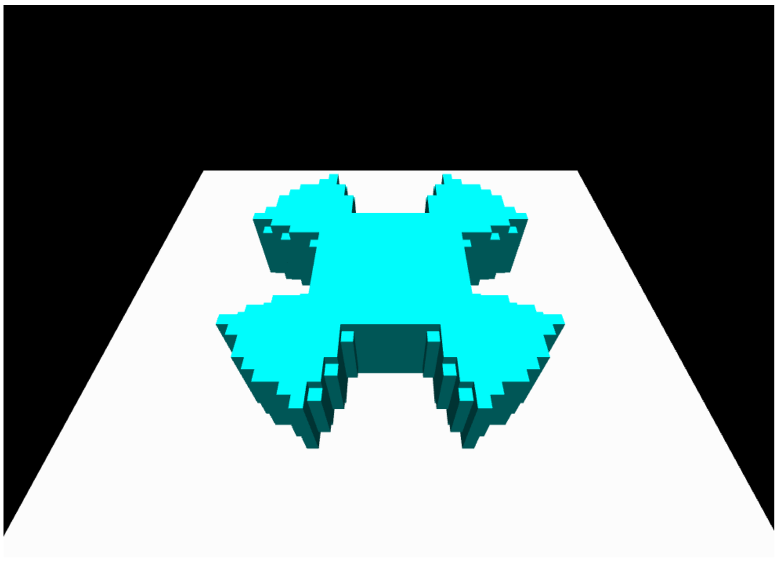


Figure 35. coupler.obj in a grid with 20x20x20 subdivisions

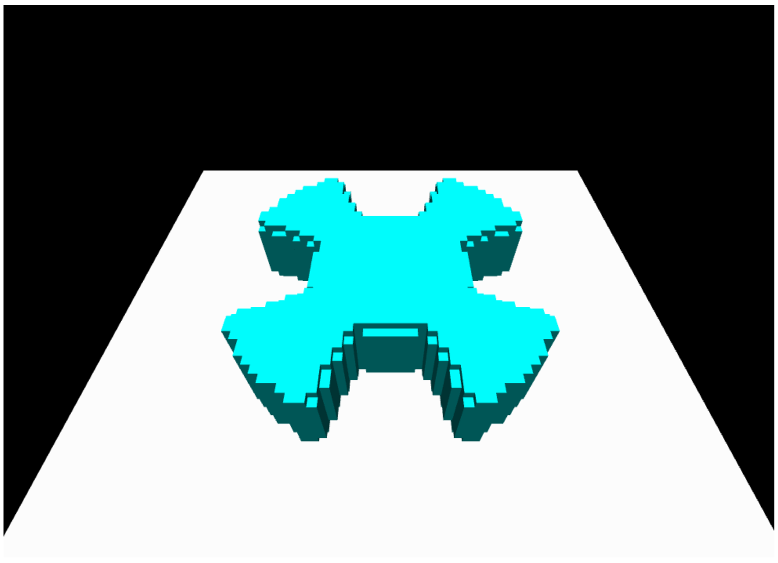


Figure 36. coupler.obj in a grid with 40x40x40 subdivisions

13 Future Work

Due to the lack of time, it was not possible to implement the controllers presented in section 8 (GUI) in the simulation in which we recreate an imported 3D model out of jelly cubes. A possible future development would be to implement the missing controllers.

Other possible future developments are:

- Further optimize the collisions and particles update computations.
- Create a more advanced parser for obj files and other file formats.
- Implement collisions between soft objects.

14 Conclusion

The main objective of this project was to simulate a gelatin cube that is able to collide and slide on rigid stationary objects, using the mass-spring system. This was not the only thing it was achieved, thanks to a very intuitive graphical user interface, the system allows not only to build a custom simulation but also to tweak the physical parameters of the objects such as friction coefficient, the springs stiffness, and more. This in combination with the possibility of visualizing the forces affecting the gelatin cube in real time using colors, allows the user to easily gain some insights on how these proprieties might affect the behavior of the cube but also on the important limitations of the methods implemented in order to build this simulation. The scope of the project was then expanded by creating a simulation in which it's possible to import a 3D model from an obj file and recreate it in a "Minecraft" fashion using soft cubes. This last simulation is without a shadow of a doubt a very interesting application of the methodologies explored in this project.

References

[1] Blender 3.4 reference manual. https://docs.blender.org/manual/en/latest/index.html. Accessed: 10.03.2023.

[2] Carnegie Mellon University physical simulation for animation case study: The jello cube. http://www.cs.cmu.edu/~barbic/jellocube_bw.pdf. Accessed: 10.03.2023.

[3] Hooke's law. https://en.wikipedia.org/wiki/Hooke%27s_law. Accessed: 10.03.2023.

[4] Autodesk Doug James Cornell University Nils Thürey ETH Zurich Matthias Müller, NVIDIA Jos Stam. *Real Time Physics Class Notes.*

[5] William Strunk and E. B. White. *The Elements of Style.* Longman Publishers, 4th edition, 1899.