Modelling Soft Body Collisions in 2D

Rex Attwood

23rd June 2023

1 Introduction

Many computer simulations need to compute collisions. A "collision" occurs when two or more objects occupy the same space. Simple collisions can be simulated by assuming the objects involved are completely rigid and will not deform when colliding. However, this method of simulation only works for non-deformable objects, whereas a "soft body" allows for deformation of an object, creating a far more realistic collision. Simulating a collision that could actually occur is very useful; for example, when crash testing new vehicles. It can be very expensive to crash a car in real life, but it is much cheaper to create a model of the car and simulate the crash. To model the collision of the car bumper, a rigid body simply would not work as the bumper is designed to crumple, so a soft body model simulation would be more suitable. A soft body model works by modelling an object as several parts which can individually move about and are not rigidly connected.

My aim for this project was to create my own soft-body simulation in 2D using python. The method I have used to model soft-bodies is called a Spring-Mass model[5]. This works by simulating several particles (points in space that have some mass) connected together with springs (connections that will exert a force between particles). The simulation is 2-dimensional as it is far simpler than 3 dimensions but will still demonstrate what a soft-body is and how it works. I am using python as the programming language as it simple and quick to write code as a proof of concept as well as being understandable to the reader of this paper.

The source code to the simulation can be found at: https://github.com/REX2626/Soft-Body-Collision

2 Simple rigid collisions

2.1 Basic movement

In order to compute the movement and collisions of the simulation the time is split into "frames". In every frame of the simulation, all the objects are moved and then collisions are computed. The time that each frame takes to run is called "delta time" (Δt). To work out the new position of each object per frame, I used the following formula:

$$Position_{new} = Position_{old} + Velocity \cdot \Delta t$$

To work out the new velocity of the object I used:

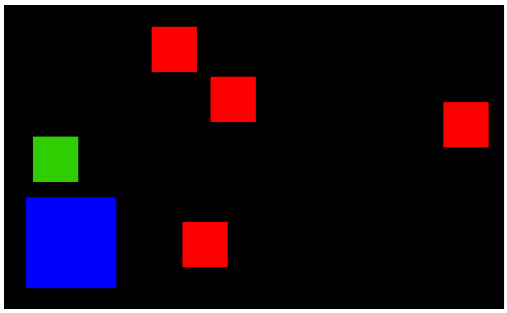

Figure 1: Example of rigid body simulation with squares. Red squares are moving, the blue square is stationary and the green square has just collided with the blue square.

$$Velocity_{new} = Velocity_{old} + Acceleration \cdot \Delta t$$

This method for updating the position and velocity of the objects in steps is called "Explicit Euler Integration"[1]. Note that this is an approximation, for small intervals of time, velocity is assumed to be constant. This requires that Δt be kept as small as possible to maintain the accuracy of the simulation. When Δt becomes too large (>20ms) the particles can jump a large distance in a single frame, as during the frame, the acceleration is assumed to be 0 and so the spring force is not taken into account. This can lead to large spring forces at the start of a frame as these particles have suddenly moved a large distance apart. These large forces will move the particle a large distance in the opposite direction leading to an increasing oscillation in position. When Δt becomes too great, the entire system quickly descends into chaos.

This movement method can be improved by changing the order in which we update the position and velocity. Using "Semi-Implicit Euler Integration"[4], the velocity is updated first and the position is updated second.

2.2 Rectangle objects

To simplify the simulation, I collide the soft-body objects with rigid-body rectangles. The rectangle object is rigid and can also be rotated. Due to how my collision code functions, by taking the corners into account (instead of position, width and height), the code is easily expandable to allow the use of all types of quadrilaterals for collisions instead of only rectangles.

```
465  class Rect(Object):
466      """
467      `pos` is centre of rectangle
468
469      `rotation` is in degrees
470
471      `outline` is the width of the outline, 0 is filled rectangle
472      """
473      __slots__ = ("width", "height", "_rotation", "outline", "surf", "tl", "tr", "br", "bl")
474      def __init__(self, pos: Vector, width: int, height: int, rotation: float = 0, colour: Colour = game.WHITE, outline: int = 5) -> None:
475          super().__init__(pos, colour)
476          self.width = width
477          self.height = height
478          self.rotation = rotation
479          self.outline = outline
480          self.surf = self.create_surface()
481          self.update_corners()
482
483      def __repr__(self) -> str:
484          return f"Rect({self.pos}, {self.width}, {self.height})"
485
486      def update_corners(self) -> None:
487          self.tl = self.pos + Vector(-self.width/2, -self.height/2).rotated(self._rotation)
488          self.tr = self.pos + Vector(self.width/2, -self.height/2).rotated(self._rotation)
489          self.br = self.pos + Vector(self.width/2, self.height/2).rotated(self._rotation)
490          self.bl = self.pos + Vector(-self.width/2, self.height/2).rotated(self._rotation)
491
492      @property
493      def rotation(self) -> float:
494          """Rect.rotation is degrees, Rect._rotation is radians"""
495          return math.degrees(self._rotation)
496
497      @rotation.setter
498      def rotation(self, new_rotation) -> None:
499          self._rotation = math.radians(new_rotation)
```

Figure 2: Here is the class code for the "Rect" object. This python object stores all the information for a rectangle, such as it's position, width, height, rotation, colour and outline width.

2.3 Detecting a collision

In the soft-body simulation, I use particles, which are dimensionless, that occupy a single point in space.

After moving every object, the code checks for any collisions. To do this, it loops through every object and checks if the particle is inside any rectangle in the simulation. To calculate if a particle is inside a rectangle, the code iterates through each side of the rectangle. For each side, we find a value of r, which is a ratio of a vector describing the side. If $0 < r < 1$, then we know that there is a vector which is normal to the side that will intersect the particle. If all 4 sides have an $r : 0 < r < 1$, then we know the particle is inside the rectangle.

For any given side, with corner positions of $\vec{A}$ and $\vec{B}$, and a particle position of $\vec{P}$, the value of r, can be found with the following equation:

$$r = \frac{\overrightarrow{AB} \cdot \overrightarrow{AP}}{|\overrightarrow{AB}|^2}$$

2.4 Resolving a collision

Once a value for r has been found for each side of the rectangle, I find the closest point on each side to the particle. Then the particle is moved to the closest side point so that the particle is no longer inside the rectangle. This is just an estimate of the position on the side, even of the side itself, that the particle should be at. If, for example, the particle was moving past the very corner of the rectangle, then there is a high chance the particle could be placed on the wrong side. The smaller Δt, the greater the accuracy of this re-positioning.

Next we need to correct the velocity of the particle so that it is moving away from the side of the rectangle. First, we take the side vector and rotate it through $\frac{\pi}{2}$ radians to find the normal vector. Then the new velocity will be the original velocity reflected through the normal vector.

Using the normal vector $\vec{n}$ and the previous velocity $\vec{v}$, the new velocity $\vec{r}$ can be found by:

$$\vec{r} = \vec{v} - 2(\vec{v} \cdot \hat{n})\hat{n}$$

3 Soft-body mechanics

3.1 Springs

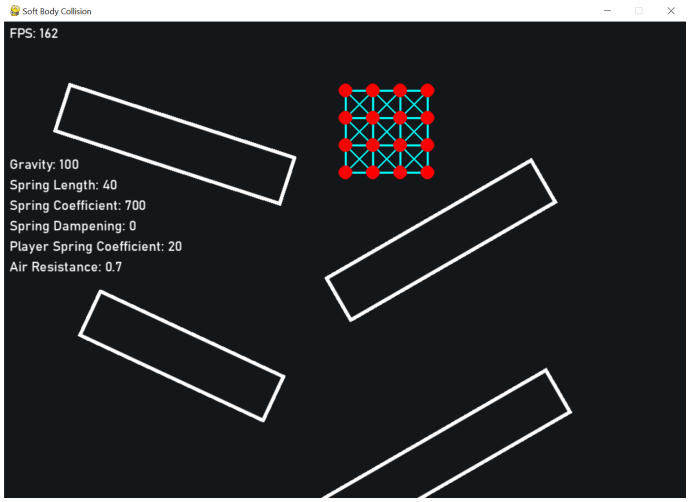

Figure 3: These static rigid rectangles demonstrate the collision of the soft body as it bounces and slides along them. There are also 4 rectangles around the border of the screen, to prevent the soft body going out of bounds.

In order to simulate the deformation for a soft-body model, I connect several particles together with springs. When these springs are stretched or compressed, they exert a force on the particles they are attached to. This allows the soft-body to maintain it's general shape whilst still being able to be deformed. By increasing or decreasing the spring constant, I can adjust the stiffness of the springs. For very large spring constants, the soft-body will be rigid and for very small spring constants, the soft-body will not maintain it's shape, having no discernible structure.

In my simulation, there are some constraints on the stiffness of the spring. If the spring coefficient is too great, the inaccuracies due to our integration approximation will cause the springs to become extended significantly. As the spring coefficient is so great, the force will be massive, which will cause the particle to be moved a great distance, resulting in the system descending into chaos.

For any spring between 2 particles with positions $\vec{p_1}$ and $\vec{p_2}$, with a natural spring length l and with spring coefficient k, the spring force is calculated as follows:

$$\vec{F} = k\vec{x}$$

```python
def update_springs(self, delta_time: float) -> None:
    """Accelerate this Particle with the Force from the springs connected to it's neighbours"""
    for neighbour, length in self.neighbours:
        distance = self.pos.distance_to(neighbour.pos)
        extension = distance - length
        force = game.SPRING_COEFFICIENT * extension
        force = self.dampen(neighbour, force)
        acceleration = neighbour.pos - self.pos
        acceleration.set_magnitude(force)  # Acceleration = Force, as mass == 1
        self.velocity += acceleration * delta_time
```

Figure 4: Python code that applies a force to every particle based on the position of the particles that it is connected to.

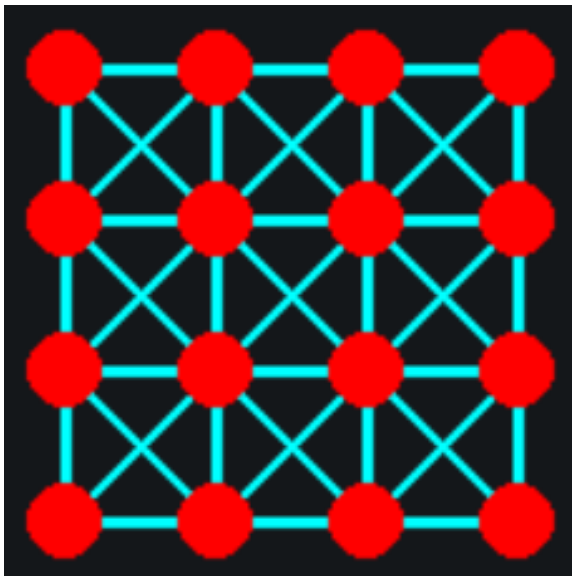

Figure 5: Red circles are the particles, blue lines are the springs.

3.2 Connecting the particles

Whilst the particles and the springs for the soft-body are now functioning, they still need to be connected. To simplify the shape of the soft-body, I used a rectangle, with given width and height. To generate the body, particles are placed in a rectangular shape a certain distance apart. Initially, each particle was connected to up to 4 other particles (up, down, left, right).

However, this led to a several issues as particles on the outside, especially at the corners, could be pushed inside the soft-body fairly easily. To fix this, I added diagonal springs so that particles could be connected to up to 8 particles. Note, the natural length of the diagonal springs must be $\sqrt{2}$ longer than the length of the cardinal springs.

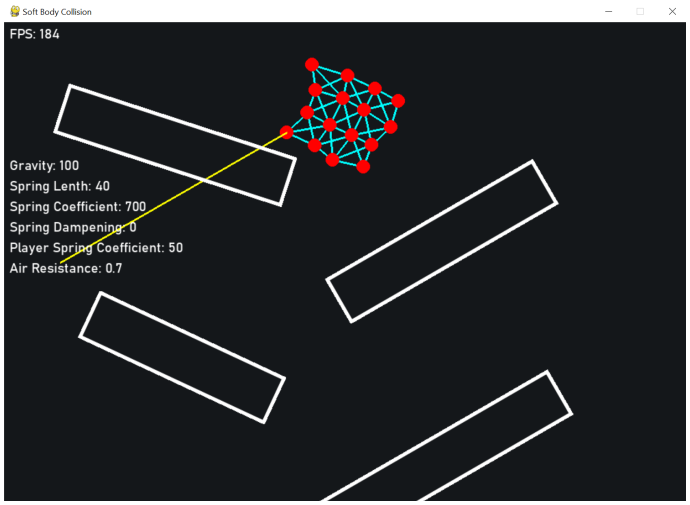

Figure 6: Example of the soft body deforming, the yellow line is a spring that pulls the soft body.

4 Simulation

4.1 Parameters

To model the soft-body, the simulation has several different parameters. The following parameters are required for basic simulation:

1. FPS: Frames Per Second, equal to $\frac{1}{\Delta t}$.
2. Gravity: The acceleration due to gravity.
3. Spring Length: The natural length of a spring, i.e. when the spring is at this length, no force is applied.
4. Spring Coefficient: The stiffness of the spring. The greater this value, the greater the force applied for a given extension.
5. Spring Dampening: When a spring force is applied, it can be reduced by a certain amount. This was set to 0 as it did not have an effect and air resistance suited the model better.
6. Player Spring Coefficient: The stiffness of the user's spring (yellow line), to move the soft-body around the map. The value needs to be increased when the soft-body is larger.
7. Air Resistance: Value from 0 to 1. The greater this constant, the greater the velocity of objects are reduced. Air resistance is especially important for this simulation, as without it, particles would continue to oscillate indefinitely, and the system is extremely likely to descend into chaos.

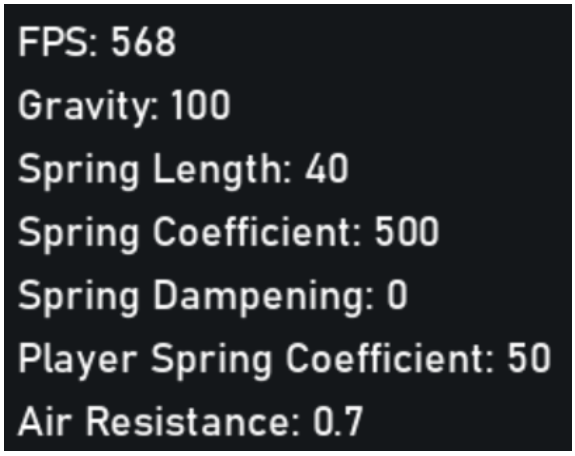

Figure 7: Different parameters and their values, which the user can modify.

4.2 User perturbation

Before beginning the simulation, the user can set the start position and size of the soft-body, as well as modify simulation parameters. However, after running the simulation, the user was unable to affect the soft-body. To improve the simulation, I wanted it to respond to real time changes and to see how it would handle more extreme situations. I first added the ability to add in particles by clicking at a given position, which was useful for testing the physics of the particles and seeing how they collided with rectangles. After this, I allowed soft bodies to be spawned in, which was useful for testing the performance of my code. For example, when several soft-bodies were added (>10), the Δt became very high and the soft-bodies began to break. The particles moved apart at tremendous speeds due to the huge spring forces because of the diminished accuracy of the simulation.

Later, I wanted the user to be able to move the soft-body around, so I added a "player spring" which is shown by a yellow line connecting the mouse to one of the soft-body particles. This player spring acts in the same way as a normal spring (with natural length 0), trying to pull the soft-body towards the user's mouse. I also added a "player pusher", shown as a grey circle, which applies an outward pushing force from the user's mouse so the user can push the soft-body around and into rectangles.

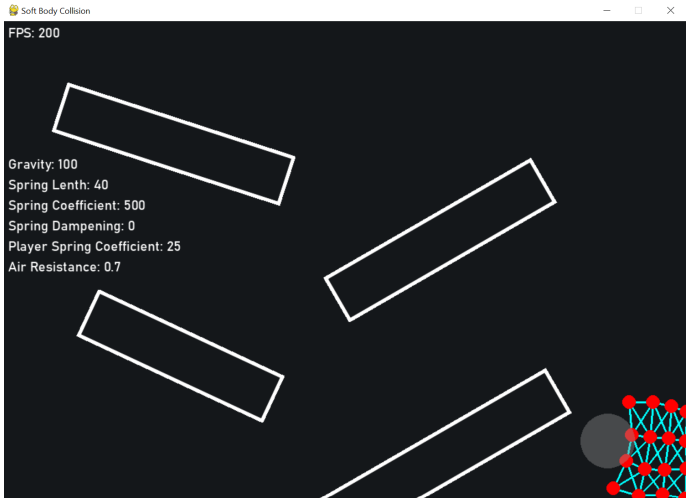

Figure 8: Grey circle is a push force from the user's mouse.

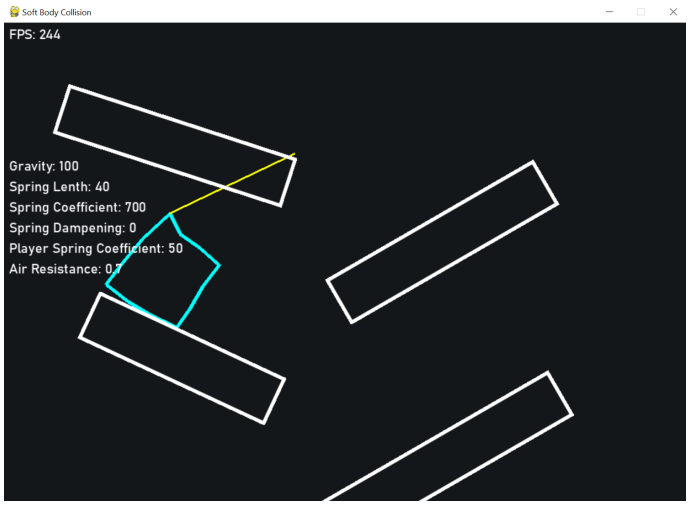

Figure 9: Blue straight lines connected together to show a soft-body square.

4.3 Outline

So far, the soft-body has been shown by drawing red circles to show the particles, and light blue lines to show the springs. However, often a soft-body is used to simulate a real object that needs a graphic representation, e.g. when modelling a wheel, to show it as a circle with a spokes, tyre etc, instead of particles and springs. This is why I added a feature to the simulation which removes the particles and springs, instead just showing the outline of the shape. This shape is drawn by choosing which particles are on the edge of the shape and drawing a straight line between them.

4.4 Circular soft-bodies

In the previous examples, the soft-bodies have been rectangles, primarily because they are simple to generate. I decided to add circular soft-bodies because circles are a commonly used shape to simulate, for example, modelling tyre deformation for a car. Generating a circle is substantially more complicated than a rectangle as it has to be approximated by generating a polynomial with a large number of sides.

To create a circular soft-body, the required inputs are: position, number of particles per ring, number of rings and distance between each ring. To generate a circle, my code completes the following steps:

1. Place a central particle at a given position (this is also the particle the player spring is connected to).

2. Place a ring of particles around the central particle, with springs connecting the ring to the centre.

3. Continue to place rings until the required amount has been placed.
4. Attach springs to the neighbour of every particle per ring.
5. Attach springs from each particle to the closest particle on the adjacent rings, as well as the neighbours to that closest particle.

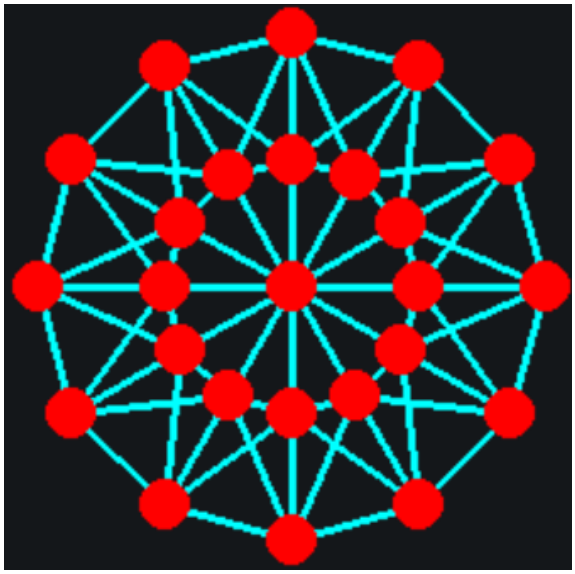

Figure 10: Here you can see the structure of a circular soft-body, with 2 rings and 12 particles per ring. I tried various different arrangements and this method of criss-crossing springs in between rings proved to be the most stable.

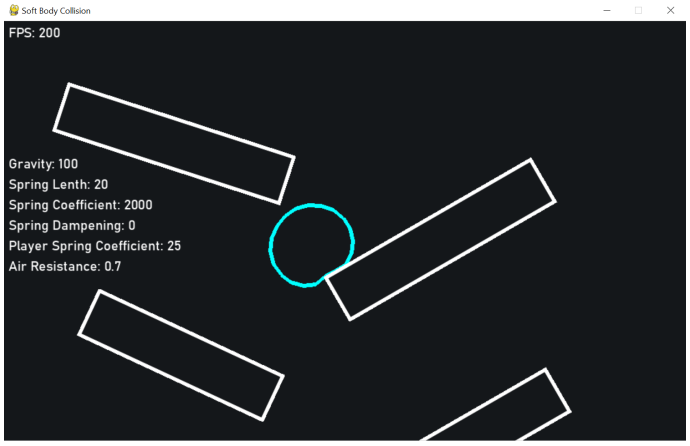

Figure 11: Example of a circle with 20 particles per ring, which is a reasonably accurate model for a circle. The sag of the circle can be seen as gravity pushes it into the rectangle.

4.5 Improving model accuracy

One of the main problems with the simulation occurs when the Δt becomes too large and the accuracy of the approximation breaks down. To rectify this, I set a maximum Δt so that if there were many objects, instead of the Δt becoming too large, the simulation itself slows down so that Δt is capped below a given value.

This limit of Δt greatly improved the simulation. Previously the system could suddenly vibrate intensely and then break down. This did not only occur when there were several objects or very large objects but could also occur when there was a "latency spike" (a large Δt for a frame).

4.6 Future improvements

The Spring-Mass model is a fairly simple yet mostly reliable model for soft-bodies. However, through extensive testing I have found several drawbacks. The system can easily break down when the time between frames becomes too great as the particles are held together purely with springs, which will become over-extended. Another drawback occurs if a great enough force pushes the body into a rectangle, some of the particles can become stuck inside the soft-body itself (the springs prevent the particle from freeing itself).

I came up with two solutions:

1. Add a rotation factor to the springs: Currently, the particles and springs are directionless, which means that the rotation of the particles and springs is irrelevant. By adding a rotation to every particle, there can be a "correct" angle of the springs relative to each particle. This will mean that the particles will resist being pulled around other particles, preventing the particles from getting stuck inside the soft-body.
2. Switch to a pressurized model[2]: This works by only generating the particles and springs for the outer layer of the soft-body. A simulated gas fills the area of the soft-body, which ensures that the area is kept constant by following the ideal gas law. This will push the particles from the centre of the body, similar to how a pressurized balloon works - when one end is pushed, the air inside pushes the other side of the balloon. The pressure inside will ensure that none of the particles can get wrapped up or stuck inside the soft-body.

5 Final thoughts

A spring-mass model for a soft-body is simple to implement and produces some great results. Both rectangular and circular shaped soft-bodies deform realistically. Some of main downsides come from the method of calculating the position and velocity of the particles. Semi-Implicit Euler integration is poor when the Δt between frames becomes too great. There are better ways of finding more accurate positions and velocities such as the Runge-Kutta method[3] which I will investigate in the future. There are also ways to improve the spring-mass model, such as adding a rotation effect to particles and springs to better maintain the shape of a soft-body. Pressurized models could also help to solve some of the problems with particles becoming trapped inside the soft-body itself. I am pleased to say that I have surpassed my original expectations of creating my own soft-body simulation in 2D using python, as I have not just created a working soft-body but also added many improvements and considered possible future improvements.

References

[1] Euler method — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Euler_method.

[2] Pressure model of soft body simulation. https://arxiv.org/ftp/physics/papers/0407/0407003.pdf.

[3] Runge–kutta methods — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Runge-Kutta_methods.

[4] Semi-implicit euler method — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Semi-implicit_Euler_method.

[5] Soft-body dynamics — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Soft-body_dynamics.