Lessons from Exploring 2D Physics Engines

Taehwan Kim

January 9, 2025

1 Integration

There are multiple options for integration.

Semi-implicit Euler Integration will be:

$$\begin{aligned} v_{n+1} &= v_n + a_n \cdot dt \\ p_{n+1} &= p_n + v_{n+1} \cdot dt \end{aligned} \tag{1}$$

Because of

$$\begin{aligned} v &= \int a \cdot dt \\ p &= \int v \cdot dt \end{aligned}$$

Verlet integration will be (Starting from 1):

$$\begin{aligned} p_n &= p_{n-1} + v_n \cdot dt \\ v_n &= \frac{p_n - p_{n-1}}{dt} \end{aligned} \tag{2}$$

Combine 1 and 2 to remove velocity v

$$\begin{aligned} p_{n+1} &= p_n + v_{n+1} \cdot dt \\ p_{n+1} &= p_n + (v_n + a_n \cdot dt) \cdot dt \\ p_{n+1} &= p_n + (\frac{p_n - p_{n-1}}{dt} + a_n \cdot dt) \cdot dt \\ p_{n+1} &= p_n + (p_n - p_{n-1}) + a_n \cdot dt^2 \\ p_{n+1} &= 2p_n - p_{n-1} + a_n \cdot dt^2 \end{aligned} \tag{3}$$

There are numerous integration methods available for calculating an object's position during discrete simulation steps, each with its own set of advantages and disadvantages. The choice of integration method can also influence the structure of the engine, making it a decision that requires careful consideration.

2 Game Loop

There are two types of game loops: Deterministic and Non-Deterministic.

The way delta time is handled determines whether a game loop is deterministic or non-deterministic.

Using Variable Delta Time (where delta time changes between frames) results in a non-deterministic game loop. This means that the simulation will not produce the same results when rerun as the time step varies.

Using Constant Delta Time (a fixed value for delta time) results in a deterministic game loop Since the time step remains consistent the simulation will produce identical results upon re-running.

3 Mass, Inverse Mass

A common way to represent "This object's mass is infinity" is by setting its inverse mass to 0. This approach is convenient because floating-point values cannot represent infinity directly, making it impossible to assign an actual infinite mass. Additionally, many equations and formulas in physics engines use the inverse mass rather than the mass itself, making it more practical to store the inverse mass. For example:

```
float m = mass;
float invMass = 1.0 / m;
```

Of course, this code is simplified.

4 Broad Phase, Narrow Phase Collision Detection

In collision detection, it is computationally expensive to loop through every object and test every edge of an object's shape. To optimize this process, collision detection is divided into two phases: broad phase and narrow phase.

Broad phase handles larger-scale checks, filtering out pairs of objects that are unlikely to collide. For example, it might use simplified shapes like circles or axis-aligned bounding boxes (AABBs) to perform quick tests. This approach drastically reduces the number of objects that require detailed examination.

Narrow phase, on the other hand, focuses on smaller-scale, detailed checks. It involves testing the actual shapes of objects, such as performing precise polygon-to-polygon collision detection by comparing edges and vertices. By combining these two phases, the system achieves a balance between accuracy and performance.

5 Contact Information

After collision detection, we gather the necessary data to resolve collisions. This data is commonly referred to as contact information. To resolve a collision effectively, we need the contact point, which is the exact point where the objects are touching or intersecting. We also need the contact normal, which is the direction perpendicular to the surface at the contact point, indicating the direction of the collision. Additionally, the penetration depth, which measures how much the objects overlap, is used to separate them appropriately.

```
Vec2 start; // collision start point
Vec2 end; // collision end point
Vec2 normal; // direction that colliding object has to go to resolve that collision
float depth; // collision depth
```

6 Constraints

Constraint is all about determining how much positional correction, impulse, or force needs to be applied to an object to manage its movement.

When dealing with positional correction, the condition is $C = 0$.

When dealing with velocity (impulse), the condition is $\dot{C} = 0$.

When dealing with acceleration (force), the condition is $\ddot{C} = 0$.

For example, a constraint for an impulse can be represented as $\dot{C} = JV + b = 0$. This equation can be used to calculate the required impulse which is then applied to the objects.

The execution order for handling these calculations is as follows:

```
for loop bodies {
    body.IntegrateForce(); // acceleration = F/M
    body.IntegrateTorque(); // angular acceleration = T/I
}
```

```
for loop constraints {
    constraint.Solve(); // impulse (velocity)
}

for loop bodies {
    body.IntegrateVelocity(); // position += velocity * dt
}
```