PhysicsSimulations

Edoardo Takanen

Abstract

One of the subjects I'm more interested in high school, is 100% physics, we started with the concept of linear motion, then proceeding with things more "complex" and inarguably more interesting. Thus, I decided to start developing physics simulations, to challenge myself and see how many things I learn at school I can implement here.

Contents

1 Introduction

These simulations will just be focusing 2D.
I chose to develop this using the C++ programming language and the graphics library SFML, attempting to implement by myself everything I need (collision detection, vectors etc.). C++ offers a very good standard library to start from as well as low level and memory management, these tools will be very useful for the simulations to run fast.
I will also be using a library for rendering real time plots of our simulations, which can help us to analyze data while the program is running. The library I am talking about is ROOT, a powerful open-source data analysis framework coming from the CERN.

2 The Engine

The first question obviously is: how can we structure our simulations' engine? I would like to follow the idea of how the Unity engine works. The idea I have in mind is not only to create this physics engine, but I also want to build an API from which people can extend and experiment the engine. So the idea is to give people the tools to create something and learn, pretty much like the SFML library that I will be using.
Now, let's define some basic things the engine will have:

1. Entities
   The basic class that let us instantiate things
2. Objects
   More complex class, extends and handles specific cases for some entities (rectangles, circles, ropes etc.)
3. Behaviors
   The ones that give some abilities and properties to our entities
4. Simulation class
   The main class that handles entities and everything that is needed for our simulations
5. Graphs manager
   The class that helps us plotting data

We will take for granted adding vectors, which is just implementing every operator.
Now we are going to look at the main function that runs a simple (and empty for now) simulation, let's write it with pseudo-code for simplicity.

```
function main() {
    auto window = new RenderWindow();
    auto graphsManager = new GraphsManager();

    auto simulation = new Simulation();

    while (true) {
        simulation.onUpdate();

        // update on fixed delta time
        // (we will talk about this later)
        for (entity in entities) {
            entity.beforeFixedUpdate();
        }

        for (entity in entities) {
            entity.checkCollisions();
        }

        for (entity in entities) {
            entity.fixedUpdate();
        }

        simulation.onDrawGraphs();

        // calculations are done, now let's draw everything

        window.clear();

        for (auto entity: GlobalVars::entities) {
            entity.render(window);
        }

        simulation.onRender(window);

        graphsManager.render();

        window.display();
    }
}
```

This is a very simplified version of what we are doing every frame, I will discuss later about the choice of this approach.

3 Simulation

As I said above, I would like to build a usable API out of this engine.
To make this, I created the Simulation class:

```
class Simulation {
public:
    double time = 0.f; // run time
    GraphsManager graphsManager;
    bool start;
    bool paused;

    sf::Clock runTimeClock;
    sf::Clock deltaTimeClock;
    sf::Clock clock;
    sf::Time elapsedTime;

    Simulation() : start(false), paused(false) {}

    // the events are fired from the main loop
    virtual void onCreate() {}
    virtual void onEvent(sf::Event) {}
    virtual void onUpdate() {}
    virtual void onDrawGraphs() {}
    virtual void onRender(sf::RenderWindow*) {}
};
```

For this paper we will focus on the event functions you can see above.

3.1 onCreate

3.2 onEvent

3.3 onUpdate

3.4 onDrawGraphs

3.5 onRender

4 Entity

The entity class is the primary structure for instantiating and managing things in the simulation.

```
class Entity {
public:
    int id;

    Entity();

    template <class T> T* TryGetBehavior() {
        for (auto behavior : behaviors) {
            T* a = dynamic_cast<T*>(behavior);
            if (a != nullptr) {
                return a;
            }
        }
        return nullptr;
    }

    void AddBehavior(Behavior* b) {
        behaviors.push_back(b);
    }

    virtual void update() = 0;
    virtual void beforeFixedUpdate() = 0;
    virtual void fixedUpdate() = 0;
    virtual void checkCollisions() = 0;
    virtual void render(sf::RenderWindow*) = 0;
protected:
    std::vector<Behavior*> behaviors;
};
```

These are all the methods every entity needs. I will talk about behaviors later, now, before looking at the 3 main phases of each entity, I want to discuss the difference between the two events update and fixed update.

4.1 update vs fixed update

Basically, update runs once per frame. fixed-update can run once, zero, or several times per frame, depending on how many physics frames per second are set in the time settings, and how fast/slow the frame rate is.
For this reason fixed update should be used when applying physics-related functions, because you know it will be executed exactly in sync with the physics engine itself.
Meanwhile, update can vary out of step with the physics engine depending on how much it takes to render the frame. If it was used for physics, it would give different results every time!

4.2 update

For this reason, since update is not recommended for physics functions, the function is not used, but I wanted to add it for completeness.

4.3 fixed update

Unlike update, fixed update is very used since everything here is related to physics. I decided to break this out and differentiate the fixed update process into 3 phases:

4.3.1 beforeFixedUpdate

This function is the first called, and it is used to update variables like the velocity.
For example, we make sure gravity is applied to our velocity each time.

4.3.2 checkCollisions

We then check for collisions for each entity in the simulation and resolve them. If a collision happens, we may want to apply some impulses to separate those entities.

4.3.3 fixedUpdate

Last of all, we update everything like the position or rotation.

4.4 render

The render function is the last function called every frame. With this we just render our entities on the screen.

4.5 Object

5 Behavior

Behaviors, as I said in the introduction, are the components that give some properties to our entities. This is the abstract class for a behavior:

```
class Behavior {
public:
    Entity* entity;
    explicit Behavior(Entity* parent) : entity(parent) {}
    virtual ~Behavior() = default;

    virtual void update() = 0;
};
```

Behaviors can be added to entities using the **AddBehavior()** function and can be retrieved with the **TryGetBehavior()** function

5.1 RigidBody

The RigidBody component is an implementation of physics for our entities. The two main functions are the followings:

5.1.1 Collision Detection

Collision detection is used to check if an object is colliding with another object This part will not cover the two collision detection algorithms (aabb/sat). I will talk about them in 5.2.
The two types of collision detection I have implemented are:

1. Discrete
2. Continuous

5.1.2 Discrete

Discrete collision detection simply uses aabb/sat algorithm and detects if there is a collision on the position where the object wants to go.

```
function checkDiscrete(final_position, other_obj) {
    this.setPosition(final_position);
    return checkSAT(this, other_obj); // AABB works as well
}
```

Pros and Cons

1. **Pros**
   Low computational cost
2. **Cons**
   Tunneling

5.1.3 What's tunneling?

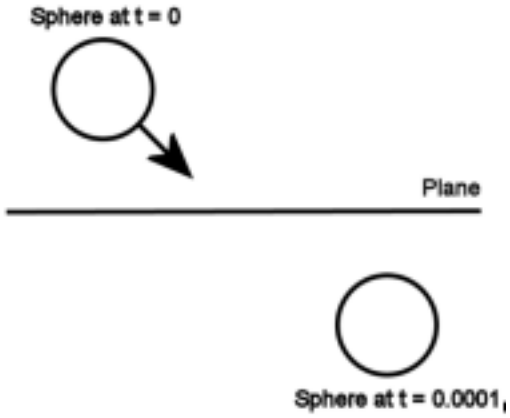

Figure 1: The tunneling problem

Tunneling is when an object moves so fast from one frame to another that he completely passes another object.

1. Make sure an object doesn't move too fast
2. Increase frames per second
3. Continuous collision detection
4. Swept volume

5.1.4 Continuous

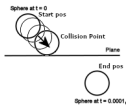

Figure 2: Continuous collision detection

Continuous collision detection checks every position between start and end position, so that the computer knows the exact position where two object are colliding.

Pros and Cons

1. **Pros**
   Precise collision detection
2. **Cons**
   Higher computation cost

5.1.5 Swept volume

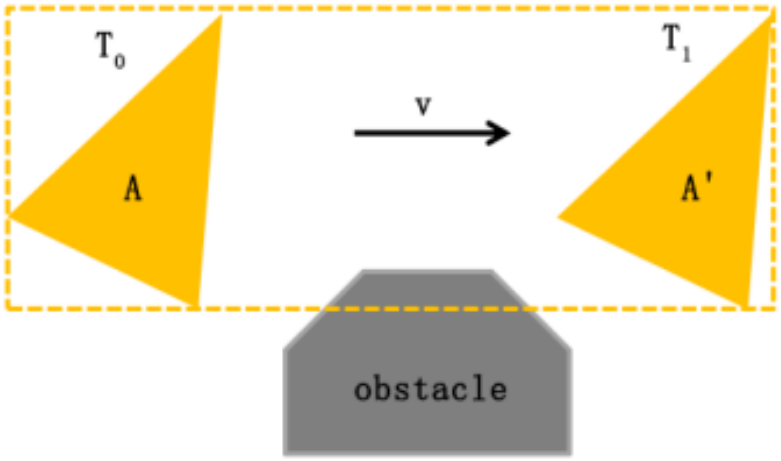

Figure 3: Swept volume

Swept volume is the volume of the object from one frame to the next. With this method, we simply check the collisions considering the whole area.

Pros and Cons

1. **Pros**
   Low computation cost
2. **Cons**
   We do not know the precise position where the collision happened

5.1.6 Velocity management

Velocity is used to calculate the next position of an object.
Note: velocity can be assigned from its variable or can be updated using the **addForce()** function.
Velocity is updated on fixed update, this also includes gravity.

5.2 CollisionShape

6 GraphsManager