Physics Simulations

Edoardo Takanen

Abstract

One of the subjects I am more interested in high school is without a doubt physics. We started with the concept of linear motion, then proceeding with things more "complex" and inarguably more interesting.
Since doing just some exercises for school test is not really that interesting, 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, a useful library that provides graphics-related implementations and allows us to focus only on the physics and how what we render on the screen will behave, thus attempting to implement by myself everything I need related to the subject (collision detection, motion, friction 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, but at the same time it means more hard and challenging code to write.
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 plan is to give people the tools to create something and learn, without having to start from scratch again, pretty much like the SFML library.
A good way to start is having in mind what things we need and how to structure the whole project, let's define some of the basic stuff that the engine will have:

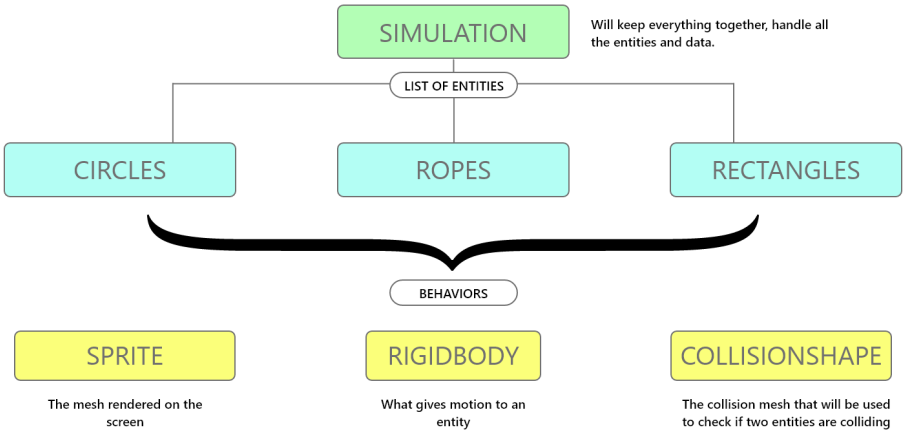
1. Simulation class
   The main class that handles entities and everything that is needed for our simulations

2. Entities
   The basic class that let us instantiate things

3. Objects
   More complex class, extends and handles specific cases for some entities (rectangles, circles, ropes etc.)

4. Behaviors
   The ones that give some abilities and properties to our entities

5. Graphs manager
   The class that helps us plotting data



We will take for granted adding vectors, which is just implementing every operator (sum, subtraction, dot product etc.).
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();

    simulation.onCreate();

    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();
        }

        // we can start drawing graphs
        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. In the very first section of the main function, we are declaring the principal variables: the window (an SFML class for rendering meshes), the graphsManager (the class for plotting graphs) and the simulation (what will handle everything, from image above 2). They will be discussed better later. Also before the while loop we prepare the simulation with the simulation::onCreate function.

Next, we are launching the simulation forever until the program is stopped.
In the loop, we:

1. First tell the simulation that a new frame is starting (simulation::onUpdate).
2. Update the entities (discussed in 4).
3. Plot our custom graphs (simulation::onDrawGraphs)
4. Render everything on the screen and inform the simulation we have drawn everything (simulation::onRender).

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, which can be extended with your code:

```
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

Event called at the very beginning of the program, before starting to build the first frame. It is usually used to create and spawn the entities used in the simulation.

3.2 onEvent

Event called when a key is pressed.

3.3 onUpdate

Event called when the program is starting to build each frame.

3.4 onDrawGraphs

Event called for plotting some data.

3.5 onRender

Event called after the frame has been fully drawn, it can be used to add additional things on the screen, like a text.

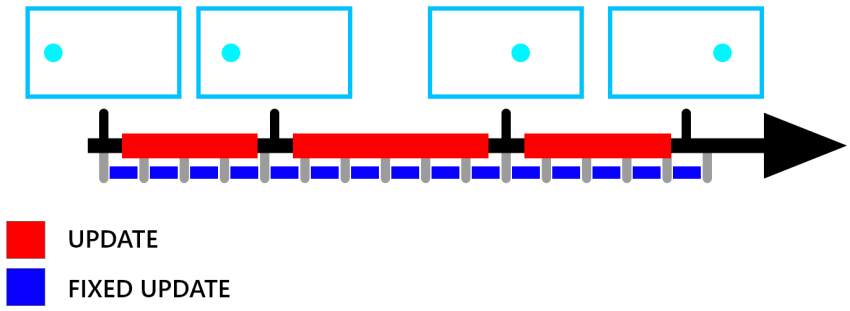
4 Entity

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, while 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. The last one ends up being more relatable because even if the program starts slowing down for a frame, the fixed update loops will always be stable. As you can see in image 4.1fixed update runs at fixed and equal intervals, unlike update.
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 the reason explained above, since update is not recommended for physics functions, the method is not used, but I wanted to add it for completeness.

4.3 fixed update

Instead, 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 method is the first called, and it is used to update variables like the velocity in the RigidBody behavior.
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

The object class is a built-in structure offered by the engine.
It already has a mesh sprite, a collision shape and a rigidbody.

```
class Object : public Entity {
public:
    Mesh mesh;
    CollisionShape* shape;
    RigidBody rb;

    Object(const Mesh& mesh, CollisionShape* collisionShape);

    explicit Object(Vector2 size);

    Object();

    ...

    inline void update() override {}

    void beforeFixedUpdate() override;

    void checkCollisions() override;

    void fixedUpdate() override;

    void render(sf::RenderWindow* window) override;

    ...
}
```

Notice that the class have a Mesh property and a CollisionShape one. This means a single object can have a sprite rendered on the screen different from the collision shape.

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;
};
```

I wanted to use the same approach used by unity, so behaviors can be added to entities using the **AddBehavior()** function, where the new value is stored in an array of Behavior pointers, and can be retrieved with the **TryGetBehavior()** function. (see 4)

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.

```
bool 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?

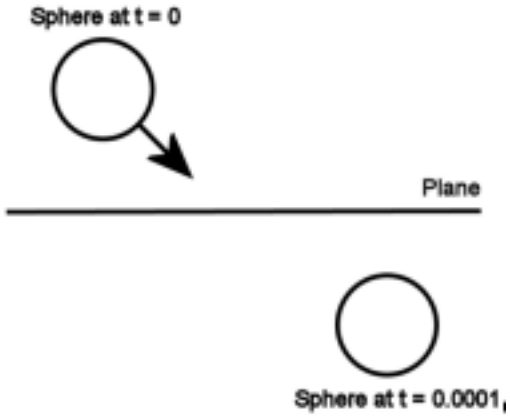


Figure 1: The tunneling problem

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

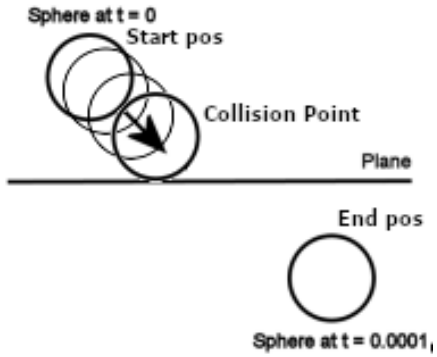


Figure 2: Continuous collision detection

Continuous collision detection tries to check for every position between start and end, so that we know the exact position where two object are colliding.

```
float precision = 0.1f;
int max = (int)(1 / precision);

for (int i = 0; i <= max; i++) {
    float j = (float)i / (float)max;

    Vector2 pos = startPos + j * direction;

    if (checkDiscreteCollision(pos)) {
        return pos; // We now have the precise position
    }
}
```

Pros and Cons

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

5.1.5 Swept volume

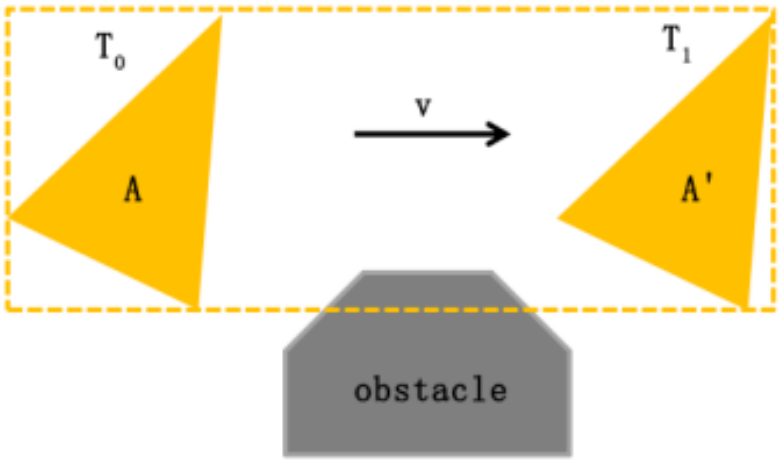


Figure 3: Swept volume

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.

5.1.7 Collision response

The last thing (but not the least important) the rigidbody does for us is collision response. What happens the an object collides with another object?
This is what the **checkCollisions** function does, it checks for collisions and

applies impulses to separate these objects.

Summing up, therefore, we apply:

1. traslational impulses
2. rotational impulses
3. friction

For this we will obviously assume we have detected that two objects are colliding and we have all the information we need, summed up with these structures:

```
struct CollidingPoints {
    Vector2 pointA;
    Vector2 pointB;
    int contactCount = 0;
};

struct Colliding {
    bool collision;
    double penetration;
    double overlap;
    Vector2 normal;
    CollidingPoints collidingPoints;
};
```

I will talk more about this in 5.2.

Traslational impulses

Assuming that there is no friction, the impulse we will generate will be entirely in the normal direction. This impulse will be only calculated if the two objects are trying to overlap, we check this calculating the relative velocity along the normal:

$$v^{AB} = (v^{AP} - v^{BP}) \cdot n \tag{1}$$

Where A and B are the two objects, P is the point of contact and n is the normal of the collision. If the dot product is greater than 0, that means the velocities are already making the objects go away from each other, so we do not need to calculate the impulse.
We can express the impulse with a scalar j times the normal of the collision. Newton's Third Law of equal and opposite forces says that the impulse for A is $j\mathbf{n}$, while the impulse felt by B is simply -$j\mathbf{n}$, the same and opposite impulse.
The formula for calculating j is the following:

$$j = \frac{-(1+e)v^{AB}}{n \cdot n(\frac{1}{m^A} + \frac{1}{m^B})} \tag{2}$$

e is our restitution value, it is a number clamped between 0 and 1, where 0 is when two objects just crash, and 1 is when they bounce. How can we decide what value we have to pick? This restitution value is based on the material of the object, so every object will have its own variable. When resolving a collision

we will just get the least value between the twos.
m^A and m^B are the masses of the two objects, notice how in this equation we need the inverse of the mass. That brings us to a consideration I actually discovered while doing my researches and that many physics engines did. How can we simulate big masses like the Earth?
We could just put a big floating number like $2^{31} - 1$, but with floating numbers imprecisions, that would not turn up to be perfect. So people came up with this idea, giving a mass of 0, so that every impulse or force applied, will be multiplied by 0, resulting to no impulse at all. The last problem to solve is the inverse of the mass, 1 over 0 will result in an error.
Physics engines figured out they could just save two variables for each rigidbody, the mass and the inverse of the mass, adjusting these values manually for masses of 0, so assigning zero to both variables.

Now that we know j, we can just update the velocities:

$$v_f^A = v_i^A + \frac{j}{m^A}\mathbf{n} \tag{3}$$

$$v_f^B = v_i^B + \frac{j}{m^B}\mathbf{n} \tag{4}$$

Adding rotations

Looking at how we can add rotations too, it really looks very similar to the previous equations. The same way we are calculating linear impulses that will change the velocities, for rotations, we will need to calculate angular impulses that will change the angular velocities.
First, we will have to change the equation for the relative velocity, since we will need to also include the initial angular velocities.

$$v^{AB} = ((v^{AP} + v_a^B) - (v^{BP} + v_a^A)) \cdot n \tag{5}$$

Having:

$$v_a^A = r_\perp^{AP}$$
$$v_a^B = r_\perp^{BP}$$

Where the two r are the orthogonals of the directions **AP** and **BP**. We then update our expression to calculate j:

$$j = \frac{-(1+e)v^{AB}}{n \cdot n(\frac{1}{m^A} + \frac{1}{m^B}) + \frac{(r_\perp^{AP} \cdot n)^2}{I^A} + \frac{(r_\perp^{BP} \cdot n)^2}{I^B}} \tag{6}$$

Where I^A and I^B are the two momentums of inertia.
These are calculated when the rigidbody is created and a mass is given. Obviously the same as for the mass happens (for the same reason) for the moment of inertia, which means that there are two variables: the intertia and the inverse

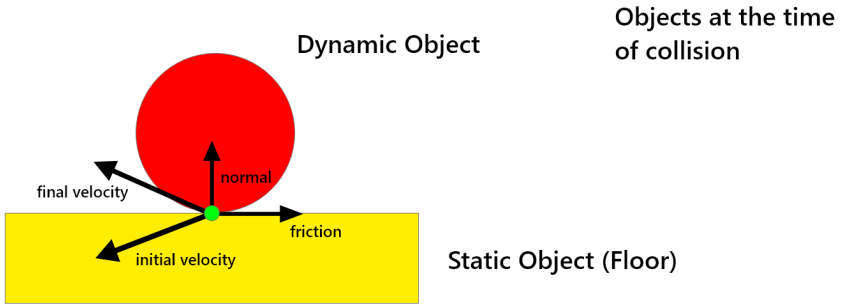
of the inertia.
Lastly, as for the velocities, we update the angular velocities:

$$\omega_f^A = \omega_i^A + \frac{r_\perp^{AP} \cdot jn}{I^A} \tag{7}$$

$$\omega_f^B = \omega_i^B + \frac{r_\perp^{BP} \cdot jn}{I^B} \tag{8}$$

Friction

If we actually tried to run our simulations now, we would see that the objects spawned keep moving slowly, like they are on an icy surface. That is because we have not implemented friction yet.
But first, what is friction?
Friction is the force that is resisting the motion of two objects sliding against each other. For our simulations this force will make our entities stop after some time. Let's see what will be the direction of our friction first.



Here we have a situation where two entities have just collided, let's assume the red circle is a dynamic object (the one that is moving) and the yellow square is a static object like a floor.
Before hitting the floor, the circle was moving towards it with an initial velocity. After the collision, the direction of the velocity is flipped according to the normal. The friction instead will be tangent to the normal direction and opposite to the final velocity.
Therefore we calculate the tangent:

$$t = v^{AB} - (v^{AB} \cdot n)n \tag{9}$$

note that the relative velocity v^{AB} is the final velocity in the image, that is because friction is calculated after separating the two entities apart

Similar to the previous calculations for impulses, we now calculate the j scalar for friction:

$$jt = -\frac{v^{AB} \cdot t}{(\frac{1}{m^A} + \frac{1}{m^B}) + \frac{(r_\perp^{AP} \cdot t)^2}{I^A} + \frac{(r_\perp^{BP} \cdot t)^2}{I^B}} \tag{10}$$

Very similar to the previous one, except that we are using the tangent instead of the normal.
note the minus at the beginning, because the impulse will be opposite to the velocity
One last thing we need to do before calculating and applying the final friction impulses is using Coulomb's law that clamps a friction force that is too big. Coulombs's law says that the friction force F_f must be less or equal than the normal force F_n times a static friction constant

$$F_f \leq F_n \cdot \mu \tag{11}$$

And with our variables we can rewrite this as:

$$jt \leq j \cdot sf \tag{12}$$

Where jt is the scalar we have calculated before and j is the scalar calculated back in (6) for the motion impulse.
That also leads us to declaring two important values: the static friction constant and the dynamic friction constant. In real life, these values really depend on the materials of the two objects.
In the future, I would like to create a list of some materials with their real constants for better results. However, for now I will just have a custom static and dynamic friction constant for each RigidBody (RigidBody::staticFriction and RigidBody::dynamicFriction) and just calculate the average.

$$sf = \frac{SF_A + SF_B}{2} \tag{13}$$

$$df = \frac{DF_A + DF_B}{2} \tag{14}$$

So if condition (12) is met, the friction impulse will be

$$i = jt \times t \tag{15}$$

Otherwise, Coulomb's law states that the friction will be

$$i = -j \times t \times df \tag{16}$$

Finally, we can add these impulses to the velocities and angular velocities of the two objects, like we did before.

5.2 CollisionShape

The CollisionShape class has the information about the shape for collisions and the algorithms of collision detection. There are different algorithms based on the type of shape we have, that is why the CollisionShape class is abstract and has different implementations.

```
enum ShapeType
{
    CIRCLE, RECTANGLE
};

class CollisionShape {
sf::Shape* sprite{};
    ... // static methods used for SAT collision detection
public:
    // abstract methods to implement
    virtual ShapeType getType() = 0;

    virtual bool aabbCollision(CollisionShape&) = 0;

    virtual Colliding satCollision(CollisionShape&) = 0;

    virtual sf::Shape* getBounds() = 0;

    virtual Vector2 getSize() = 0;
}
```

These are the methods to override for extending the class.

1. **getType**
   returns the type of shape (circle or rectangle for now)

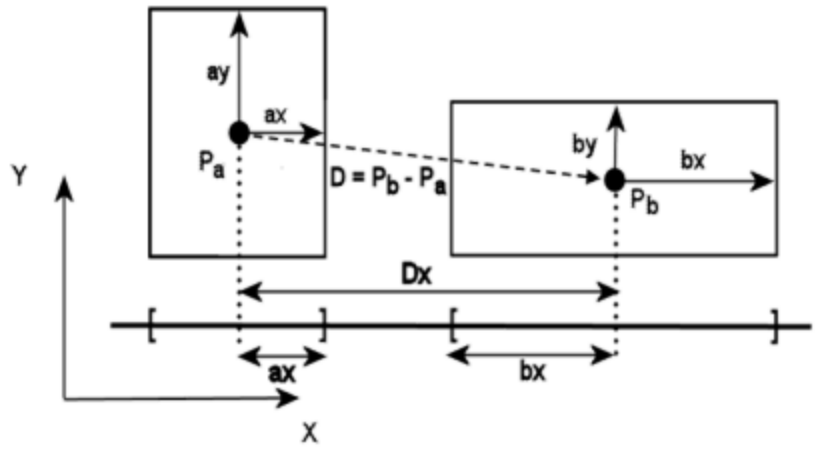
2. **getBounds**
   returns the SFML shape

3. **getSize**
   returns the size of the shape, needs to be interpreted based on the shape.
   For circles the radius is the length of the vector.
   For rectangles the x and y sizes are just the vector's components

4. **aabbCollision**
   contains the logic for the AABB algorithm, returns a boolean whether the two objects collide or not.

5. **satCollision**
   contains the logic for the SAT algorithm, returns a structure containing all the information of the collision.

5.2.1 AABB algorithm

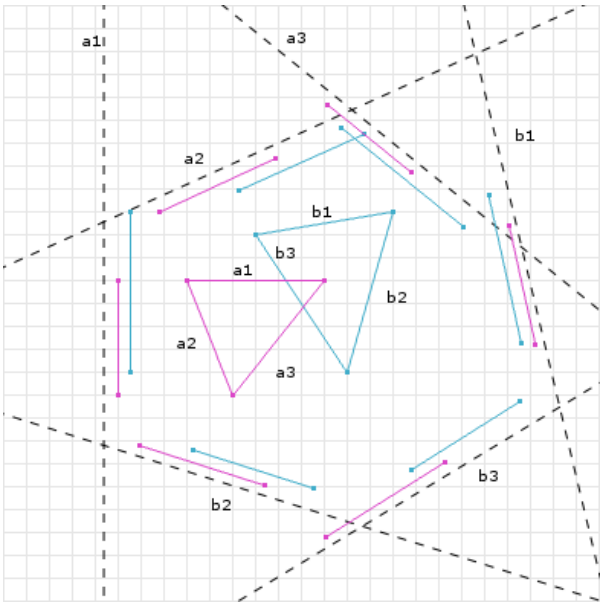


AABB stands for Axis-Aligned Bounding Box.
This method only works with rectangles or squares and only if they're axis-aligned.
If two objects are colliding, all of these conditions will be true:

$$\begin{aligned} other.left &\leq object.right \\ other.right &\geq object.left \\ other.top &\geq object.bottom \\ other.bottom &\leq object.top \end{aligned} \tag{17}$$

note that this algorithm is implemented in the project but never used, because since we are also dealing with angular velocities and rotations, this method will not work

5.2.2 SAT algorithm

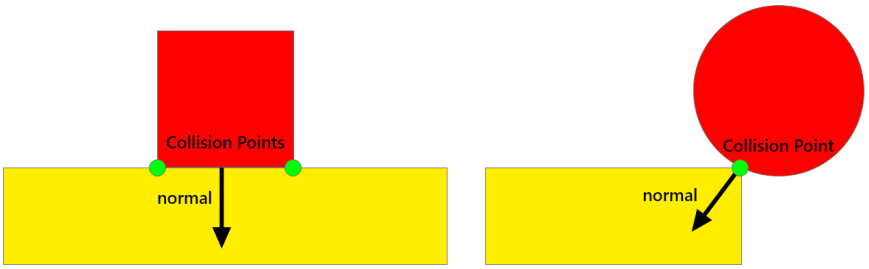


SAT stands for Separating Axis Theorem.
This is a much better algorithm (but also more expensive) than AABB because it works with **every convex** shape and **every rotation**.

What we basically do is make a projection of both shapes on every axis. If all the projections overlap, the objects are colliding.
As we saw in 5.2.2, collision resolution needs some more information rather than just a boolean value that says if a collision happened.

```
struct CollidingPoints {
    Vector2 pointA;
    Vector2 pointB;
    int contactCount = 0;
};

struct Colliding {
    bool collision;
    double penetration;
    double overlap;
    Vector2 normal;
    CollidingPoints collidingPoints;
};
```

Penetration is the greatest overlap encountered, while overlap is the least overlap, these values are used to correct the visible bug of two objects being one over the other.



The normal is the normalized direction of the collision.

5.2.3 Circle-Polygon algorithm

To check if a circle is colliding with a convex polygon, we iterate through every vertex of the polygon and get the least distance between a vertex and the center of the circle. If the least distance is less or equal than the circle's radius, the two objects are colliding.

5.3 Mesh

The Mesh class is a wrapper for SFML's shapes, it represents the sprite that will be rendered on the screen (NOT the collision bounds).

```
class Mesh : public Behavior {
public:
    sf::Shape* shape;

    explicit Mesh(sf::Shape* shape)
        :   Behavior(nullptr),
            shape(shape)
    {}

    void update() override {}

    inline void draw(sf::RenderWindow* window) const
    {
        window->draw(*shape);
    }

    static Mesh RectangleMesh(Vector2 size, sf::Color color);
    static Mesh CircleMesh(float radius, sf::Color color);
};
```

6 GraphsManager

The GraphsManager is a wrapper for CERN's library: ROOT.
It simplifies the process of plotting real-time graphs for analyzing data from the simulation, whether for debugging or not.

```
class GraphsManager {
private:
    std::vector<Graph*> graphs;
    TApplication rootApp;
    std::unique_ptr<TCanvas> canvas;

    static Option_t* getGraphMode(GraphMode m);
public:
    inline GraphsManager() :
        rootApp("Graphs", nullptr, nullptr),
        canvas(std::make_unique<TCanvas>("canvas", "Physics
    simulations"))
    {}

    void setCanvasSize(sf::Vector2i size);

    void setCanvasPosition(sf::Vector2i pos);

    void addGraph(
        const char* title,
        graph_modifier func = [](TGraph* _){},
        GraphMode mode = GraphMode::DEFAULT
    );

    void build();

    void addPoint(int i, double x, double y);

    void render();
};
```

Here's an example of building some graphs
note that the simulation class already has a graphManager property

```
// inside a class that extends the Simulation structure
void onCreate() override {
    // sets the window size
    graphsManager.setCanvasSize(sf::Vector2i(1550, 700));
    // sets the window position on the computer
    graphsManager.setCanvasPosition(sf::Vector2i(-2048, 0));

    // adds a plotting space to the window
    graphsManager.addGraph("Test");
    graphsManager.build();
}
```