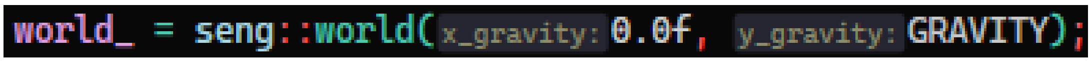
2D Physics Engine Manual

Introduction

The library allows you to define a physical world with data by your specific needs and then create objects within it. The library automatically detects collisions using an efficient spatial management system and resolves them either with predefined collision managers or you can customize the resolution by providing your own resolution method.

World creation

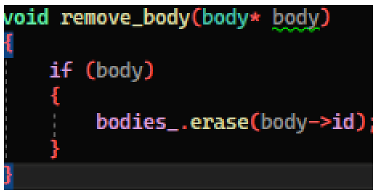
To create a world, first include the header '<2DPhysicsEngine.hpp>' and create a world object like this: 

```
world_ = seng::world(x_gravity: 0.0f, y_gravity: GRAVITY);
```

This initializes the world objects and sets its gravity (GRAVITY is just a constant representing -9.81).

You need to ensure that the world::update_world(float dt) method is called, so that the world actually updates automatically.

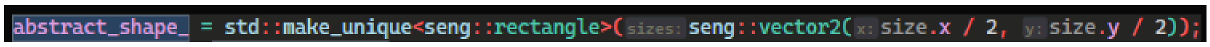
To delete a body from the world, use this method:



```
void remove_body(body* body)
{
    if (body)
    {
        bodies_.erase(body->id);
    }
}
```

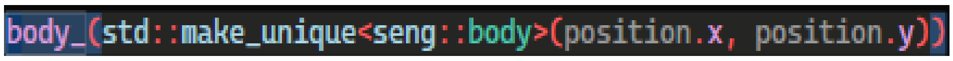
Body creation

We created a world, now how can we add objects into it, so they get updated within the physics simulation? To add an object it is necessary to store its body and a shape. The body shape is basically just how we want the object to be interpreted. There are 4 basic types of shapes (so far 2 of them are fully supported) – Circle, Rectangle, Polygon, Ellipse. These are all represented as classes in the library, all inheriting the body_shape class. To create a body_shape object (for example a rectangle), just create the desired interpretation for your object's shape inside the world:



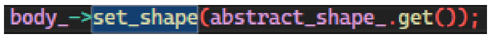
```
abstract_shape_ = std::make_unique<seng::rectangle>(sizes: seng::vector2(x: size.x / 2, y: size.y / 2));
```

Then we have to define the body, which basically only stores the physical data about the object (velocity, position, angle and shape), this can be done like this:



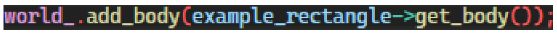
```
body_(std::make_unique<seng::body>(position.x, position.y))
```

This calls the default constructor for the body, but we can also manually set its values according to our needs using the setters in the body class. After this, we need to use one of the setters (body::set_shape(body_shape*)) like this:



```
body_->set_shape(abstract_shape_.get());
```

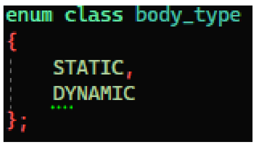
And finally, you have to just add the body to the world:



```
world_.add_body(example_rectangle->get_body());
```

Now the body is inside the world and will be registered during collision detection and response phases.

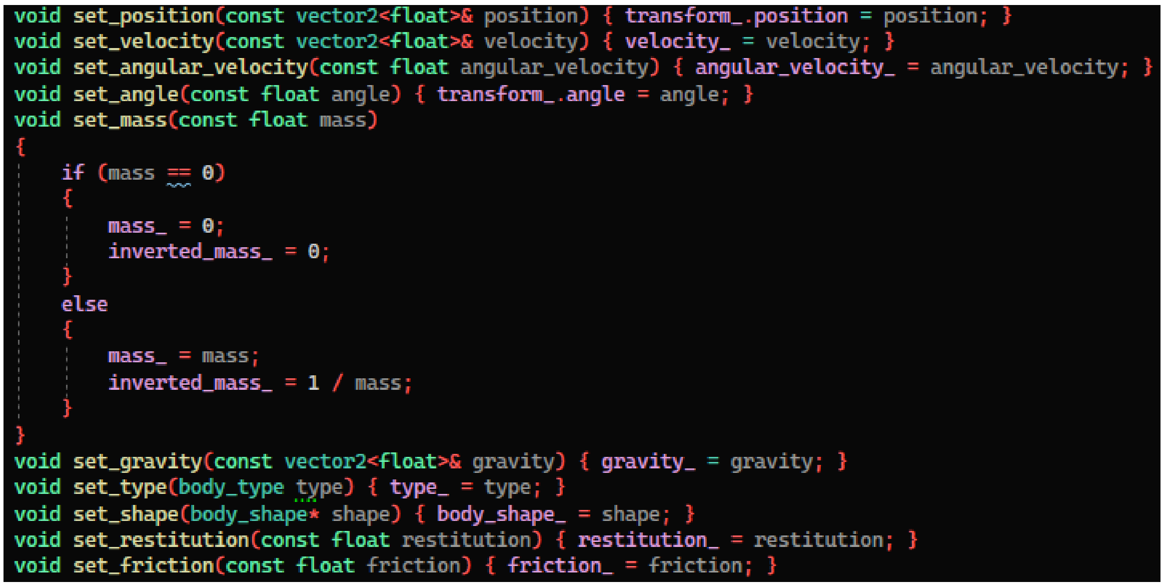
Body also has different physical types. The only ones defined so far in this version are:



```
enum class body_type
{
    STATIC,
    DYNAMIC
};
```

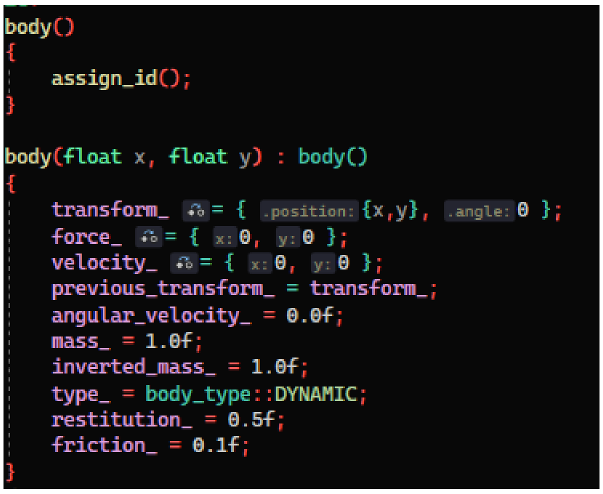
Static means that the body does not move and only other objects can interact with it. Dynamic moves and collides with other objects.

The fields that can be changed manually:



```
void set_position(const vector2<float>& position) { transform_.position = position; }
void set_velocity(const vector2<float>& velocity) { velocity_ = velocity; }
void set_angular_velocity(const float angular_velocity) { angular_velocity_ = angular_velocity; }
void set_angle(const float angle) { transform_.angle = angle; }
void set_mass(const float mass)
{
    if (mass == 0)
    {
        mass_ = 0;
        inverted_mass_ = 0;
    }
    else
    {
        mass_ = mass;
        inverted_mass_ = 1 / mass;
    }
}
void set_gravity(const vector2<float>& gravity) { gravity_ = gravity; }
void set_type(body_type type) { type_ = type; }
void set_shape(body_shape* shape) { body_shape_ = shape; }
void set_restitution(const float restitution) { restitution_ = restitution; }
void set_friction(const float friction) { friction_ = friction; }
```

Finally, this is the default way of constructing body:



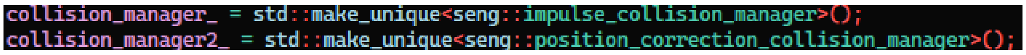
```
body()
{
    assign_id();
}

body(float x, float y) : body()
{
    transform_ = { .position: {x,y}, .angle: 0 };
    force_ = { x: 0, y: 0 };
    velocity_ = { x: 0, y: 0 };
    previous_transform_ = transform_;
    angular_velocity_ = 0.0f;
    mass_ = 1.0f;
    inverted_mass_ = 1.0f;
    type_ = body_type::DYNAMIC;
    restitution_ = 0.5f;
    friction_ = 0.1f;
}
```

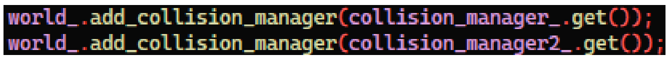
Collision Managers

To make the world respond to these collisions, we have to give it some algorithms, which will resolve them. This can be achieved by either using the two pre-set ones (the impulse CM and position correction CM) or defining a function and using the custom Collision Manager:

1. Use the pre-set ones:

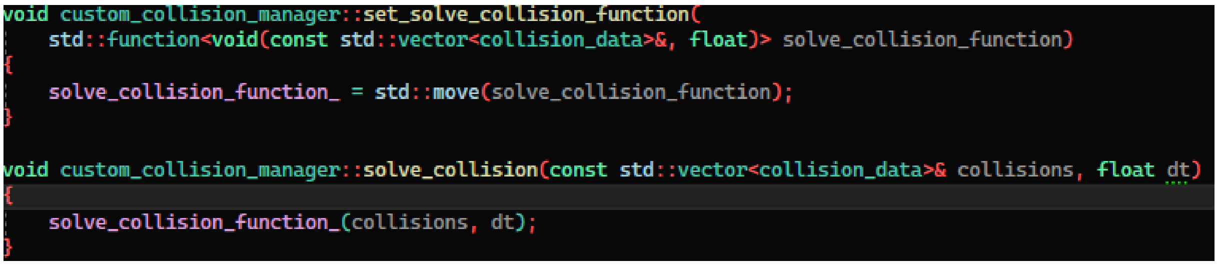


```
collision_manager_ = std::make_unique<seng::impulse_collision_manager>();
collision_manager2_ = std::make_unique<seng::position_correction_collision_manager>();
```



```
world_.add_collision_manager(collision_manager_.get());
world_.add_collision_manager(collision_manager2_.get());
```

2. Create instances of seng::custom_collision_manager and then give them proper resolving functions (check code documentation for an idea of how to do that), use the setter method for passing the function to the class and then the code will automatically call the latter method:



```
void custom_collision_manager::set_solve_collision_function(
    std::function<void(const std::vector<collision_data>&, float)> solve_collision_function)
{
    solve_collision_function_ = std::move(solve_collision_function);
}

void custom_collision_manager::solve_collision(const std::vector<collision_data>& collisions, float dt)
{
    solve_collision_function_(collisions, dt);
}
```

2DMath

The library comes with its own helping math module, which will aid in working with either defining the collision manager or generally working with 2D objects and values.

It contains:

- Definition for transform (which is just a structure encapsulating position and angle).
- Definition for 2d vector (seng::vector<T>), which is mostly made to be use with floating point values since it also contains specific numeric methods, only possible on the floating point type. It basically contains the typical 2d vector functions like dot&cross product, vector arithmetic, min, max and floor function
- Definition for 2x2 matrix<T>. Less strict, but still recommended to be used with either integer or floating point types.