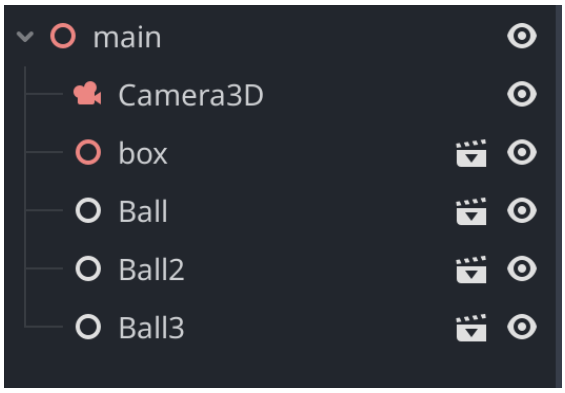
Game name: Box Bounce

Implemented functionality:
There is a 6 walled box in which 3 spherical balls bounce around. There are 2 lights (1 directional and 1 point) that light up the inside of the box and a camera that displays this. The 3 balls move around in random directions each time and at random speeds each time. The 3 balls bounce off the walls and off each other with perfect elasticity meaning no energy loss. When multiple balls collide at once, we assume that they simply bounce off each other all at the same time still with the perfect elasticity mentioned before.

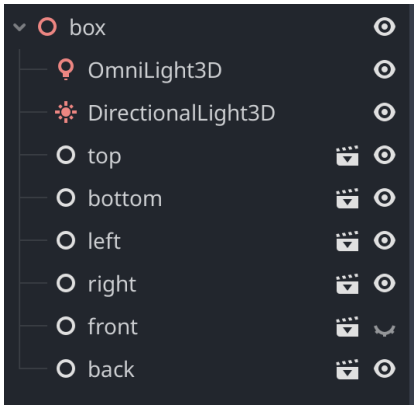
How to start:
Download the GitLab repository and unzip the files. Clone the godot cpp submodule by running the following in the terminal in the project folder: "git clone -b 4.1 https://github.com/godotengine/godot-cpp". Compile the code with the command "scons" in the directory with the SConstruct file. Open the godot project manager software by running "godot" command in the terminal. Click scan in the right hand side navigation bar and choose path to the Project folder. Open the project and click the play button to run the game.

Link to video footage: https://youtu.be/zDFK3Bj3tO0

Code layout:

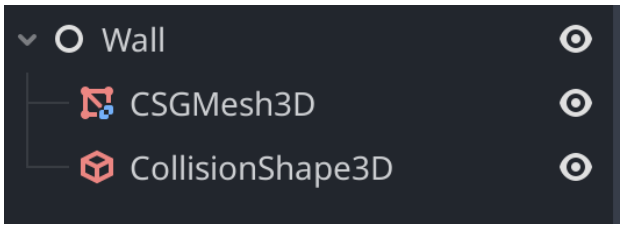


At the highest level, the game is made of a main scene that is composed of a Camera3D node and different prefabs. The main scene is just a generic Node3D but it has a Camera3D node used to capture the gameplay and a box prefab that has the walls and various ball prefabs that have the ball structure and behavior.



At the next level, we have the box scene. This is the main box structure and has 2 lights and 6 walls. It has a pink OmniLight3D node that acts as a point light that light up the room from the bottom outwards as well as a blue DirectionalLight3D node that acts as a spotlight that lights up the base of the room from the top. We also have 6 walls that are configured to be rotated and positioned such that they form a cube box. Each of these also has a normal vector that will be useful in calculating the reflection vectors for when the balls make contact

and need to bounce off the walls. Notice also that the front wall is made invisible so that the camera may be moved outwards and still see through the wall.



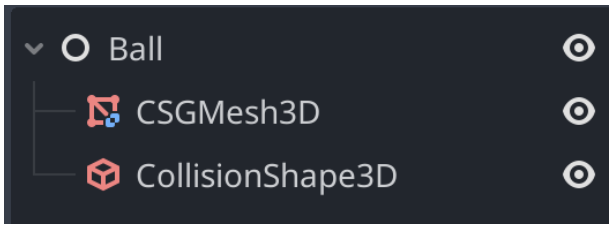

```
void Wall::_bind_methods() {
    ClassDB::bind_method(D_METHOD("get_normal"), &Wall::get_normal);
    ClassDB::bind_method(D_METHOD("set_normal", "p_normal"), &Wall::set_normal);
    ClassDB::add_property("Wall", PropertyInfo(Variant::VECTOR3, "normal"), "set_normal", "get_normal");
}
```

```
Wall::Wall() {
    normal = Vector3(0, 0, 0);
}
```

```
void Wall::set_normal(const Vector3 p_normal) {
    normal = p_normal;
}

Vector3 Wall::get_normal() const {
    return normal;
}
```

The wall nodes are custom nodes that extend the Area3D godot class. As mentioned they have an additional property that stores the normal vector of each wall for calculating the reflection vector of balls that contact and bounce off the various walls. They each have a CSGMesh3D node to create the texture of the walls. They also have a CollisionShape3D node used to track the contact of the balls with these walls. As shown, the main code for these custom wall nodes is adding the normal property as well as the getter and setter methods for this property (this includes making the methods, initializing the property, binding the methods, and adding the property to the node).




```
Ball::Ball() {
    velocity = Vector3(0, 0, 0);
    connect("area_entered", Callable(this, "_when_area_entered"));
}
```

```
void Ball::_bind_methods() {
    ClassDB::bind_method(D_METHOD("set_velocity", "p_velocity"), &Ball::set_velocity);
    ClassDB::bind_method(D_METHOD("_when_area_entered", "area"), &Ball::_when_area_entered);
}
```

```
void Ball::_when_area_entered(const Area3D* area) {
    const Wall* wall = Object::cast_to<Wall>(area);
    if (wall) {
        Vector3 N = wall->get_normal();
        Vector3 R = velocity - 2 * N * (N.x * velocity.x + N.y * velocity.y + N.z * velocity.z);
        set_velocity(R);
    } else {
        const Ball* ball = Object::cast_to<Ball>(area);
        if (!ball) {
            return;
        }
        Vector3 N = get_position() - ball->get_position();
        N = N / N.length();
        Vector3 R = velocity - 2 * N * (N.x * velocity.x + N.y * velocity.y + N.z * velocity.z);
        set_velocity(R);
    }
}
```

```
void Ball::_ready() {
    RandomNumberGenerator rand;
    float x = rand.randf_range(-1, 1);
    float y = rand.randf_range(-1, 1);
    float z = rand.randf_range(-1, 1);
    set_velocity(Vector3(x, y, z));
}
```

```
void Ball::set_velocity(const Vector3 p_velocity) {
    velocity = p_velocity;
}
```

```
void Ball::_process(double delta) {
    if(Engine::get_singleton()->is_editor_hint()) {
        return;
    }
    Vector3 old_position = get_position();
    Vector3 new_positon = old_position + velocity;
    set_position(new_positon);
}
```

Lastly, there is the custom ball node created by extending the godot Area3D node. It has a CSGMesh3D node to create the texture of the balls. It also has a CollisionShape3D node used to track the contact of the balls with the walls as well as balls with balls, which allows us to see when the ball makes contact with the walls/other balls. The code of the ball node is a bit more complex than that of the wall node. First, we initialize a velocity attribute that will be used to determine the direction and speed of the ball's movement and we connect godot's area_entered signal to our custom callback method. We create a set_velocity method to actually set the value of the attribute as well as the method that controls the logic of collisions. We bind both of these methods. The area entered collision logic method checks whether the colliding object is a wall or a ball to get the normal vector by either getting it as an attribute of the wall or calculating it with vector subtraction and normalization. We do reflection vector calculation and set the velocity of the ball to this new reflected vector. We also have a ready method that starts the balls off in a random direction and at a random speed by setting the velocity to a random value. Lastly, we have the process method where we update the position by the velocity each time the method is run, which allows the ball to move as the game runs.