Advanced Rigidbodies Lab

Table of Contents

Lab Overview

Welcome to the advanced portion of the movement lab! Here we'll be discussing smooth ground and aerial movement, some basic math and physics that will make your life a little easier, and give you a look into physics materials. Keep in mind this lab will assume you understand what we've already discussed in the previous rigidbody basics lab.

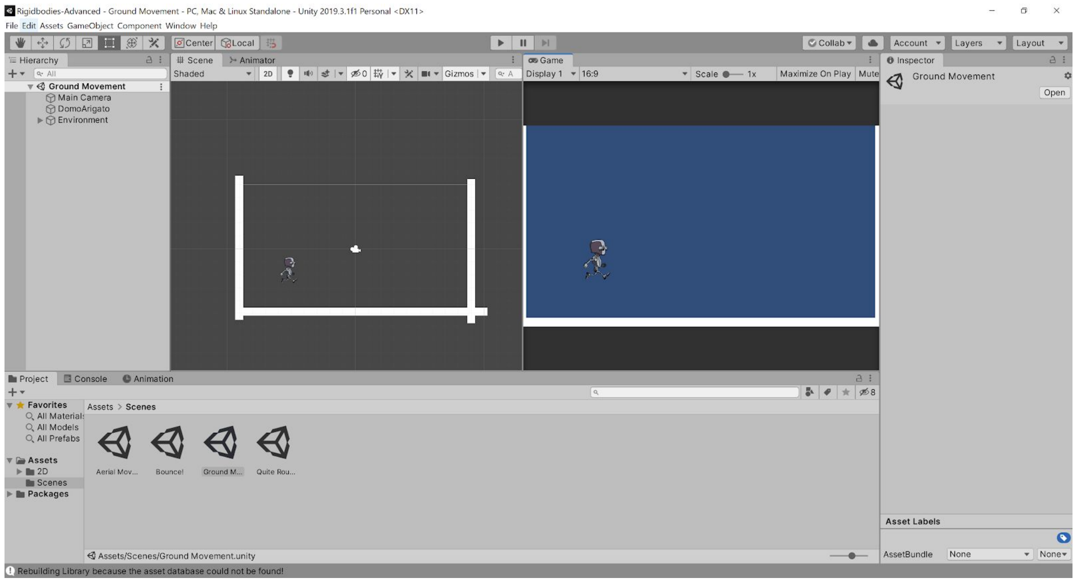
Lab Instructions

Part I: Smooth Physics

In this section we'll be going over some practices that make for smooth ground-based movement. Before we move on, keep in mind that these are general tips. They assume relatively Earth-like conditions, so they probably won't be as applicable, for example, in a zero-gravity environment.

In a 2D environment, there are a few ways to handle movement. The two main categories we will be exploring today are movement through the use of physics and movement through the direct manipulation of an object's transform.

If you haven't already, launch Unity and navigate to the Ground Movement scene. You should see a simple box and a robot, like so:


Rigidbodies-Advanced - Ground Movement - PC, Mac & Linux Standalone - Unity 2019.3.1f1 Personal <DX11>

Assets/Scenes/Ground Movement.unity

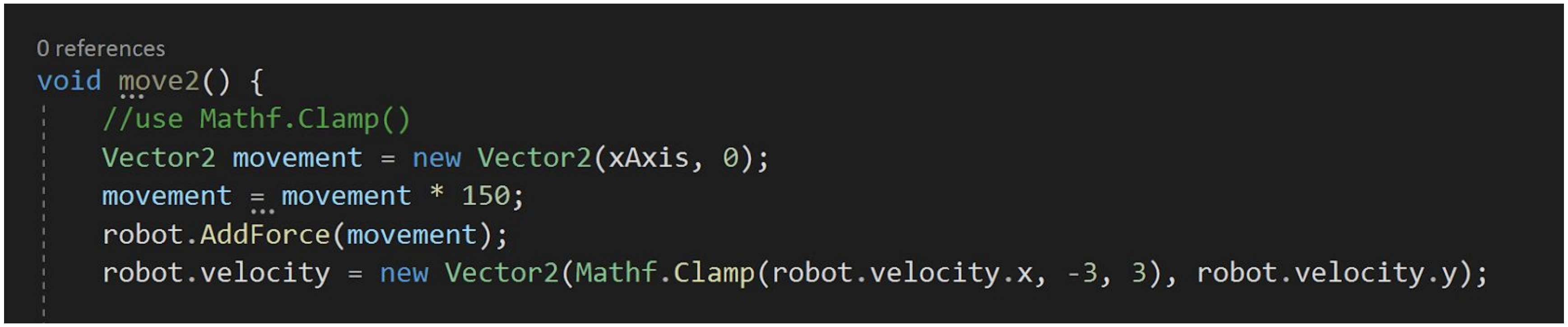
Rebuilding Library because the asset database could not be found!

The robot has a standard Ridigbody2D and BoxCollider2D. The box has a BoxCollider2D as well. Make sure the robot's gravity scale is at 1, then hit play. You'll notice that the robot doesn't really move once it hits the box (vertical movement is disabled for this section). Now change your gravity scale to 0 and try again. You should see that the robot can now move freely in the horizontal axis. Why is this?

The answer is actually a bit more involved, but the short explanation is that while the BoxCollider2D and Rigidbody2D don't have a physics material on them, there *is* a default frictional value of 0.4 and a bounciness level of 0 on the robot and box. We'll dive into physics materials a little later, but for now all you need to know is that friction exists, and that it will decelerate your ground based objects. You can open up the GroundMovement script and multiplicatively scale the movement vector in *move1()* in order to see how strong this deceleration is.

Movement like that in *move1()* is physics based movement. If the size of the box were infinite, holding down the right arrow with *move1()* would mean that the robot would constantly accelerate and its velocity would continuously increase. This probably isn't a desired result, and unchecked acceleration at higher rates can cause serious, game-breaking bugs. We can prevent this type of behavior in a variety of ways, but we'll be showcasing *Mathf.Clamp* here.

In the GroundMovement script under the FixedUpdate() function, change the last line from *move1()* to *move2()*. Hit play and move the robot left and right. Things should get pretty wacky, and you might even get the robot to clip out of the box. Uncomment the last line of *move2()* and try again.



```
0 references
void move2() {
    //use Mathf.Clamp()
    Vector2 movement = new Vector2(xAxis, 0);
    movement = movement * 150;
    robot.AddForce(movement);
    robot.velocity = new Vector2(Mathf.Clamp(robot.velocity.x, -3, 3), robot.velocity.y);
```

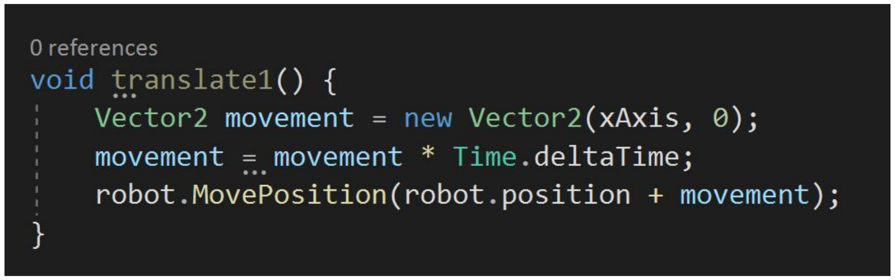
In the last line of code, we're clamping the velocity Vector2. *Mathf.Clamp* makes sure that the robot's x velocity is bounded by -3 and 3. There are many other types of clamps but all in all they're very useful for preventing numbers from getting out of control.

Your task will be to fill out *move3()*. You will be making a similar move function to that of *move2()*, with a few differences. You will use *Input.GetKeyDown* or its variants and directly set the robot's velocity upon those button presses. Make sure the robot doesn't clip through walls with high speed, and make sure the robot comes to a stop when no keys are pressed. Upon checkoff, tell your TA why one might use instantaneous velocity manipulation, and give a situation where it might not be advisable. Remember to change the last line of FixedUpdate to *move3()*!

Part II: Transformers

Another way of moving an object is to manipulate transforms or by translating an object. Direct manipulation offers the benefits of giving the player very fine control. However, the movement will often feel more rigid and won't be as smooth, with controllable objects starting up and coming to a stop almost instantaneously. Furthermore, transform manipulation clashes with physics, so you'll either have to be careful when moving a physics based object using transform translation or just disable physics on that object altogether.

Change the *move3()* function of the robot in FixedUpdate to *translate1()* by replacing the last line of the FixedUpdate function. Hit play and try it out. Does movement feel super choppy? If so, it's probably because the robot is literally teleporting to its new position many times a second across the screen. A way to minimize the effects of the choppiness is to multiply the movement vector by Time.deltaTime.

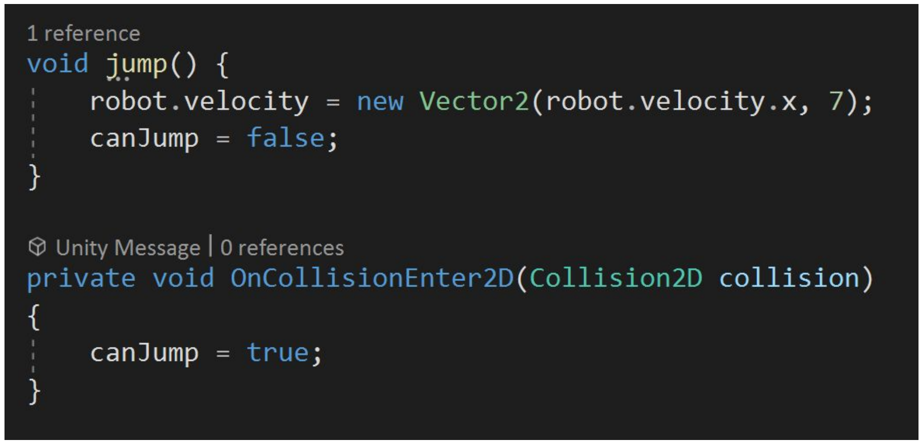

```
0 references
void translate1() {
    Vector2 movement = new Vector2(xAxis, 0);
    movement = movement * Time.deltaTime;
    robot.MovePosition(robot.position + movement);
}
```

Now scale up the movement by multiplying by a constant. (For the purpose of the lab, choose a number between 10 and 15.) Why does the same choppiness come back? Upon checkoff, explain to your TA why multiplying the vector by a constant to speed up the movement might not be such a good idea when Time.deltaTime is used.

Part III: Aerial Ace

Aerial mobility is an integral portion of many video games. 2D jumps and aerial movement can be tricky though. This section will only discuss physics based aerial movement and jumps, as there's not really much to be said about transform translation that is much different from the ground based version. The only real difference is the addition of one axis of movement. Swap over to the Aerial Movement scene to get started.

Let's begin with jumping. While you might be tempted to use *.AddForce()* like we did for horizontal movement, keep in mind the nature of a jump. Using *.AddForce()* would not only make it so your jump is competing with gravity in a struggle to move your robot upwards, the acceleration would be gradual and not immediate. Take a look at the *jump()* function in the AerialMovement script attached to the robot:



```
1 reference
void jump() {
    robot.velocity = new Vector2(robot.velocity.x, 7);
    canJump = false;
}

Unity Message | 0 references
private void OnCollisionEnter2D(Collision2D collision)
{
    canJump = true;
}
```

Instead of applying forces using physics, we directly modify the robot's y axis velocity. If you haven't already, uncomment the code, fill out your own *move1()* function, and take the robot out for a spin.

The rationale for direct velocity manipulation is that jumps occur instantaneously. In most video games, a character's jump is similar to a compressed spring suddenly being released. By manipulating the velocity directly, we can achieve the desired effect.

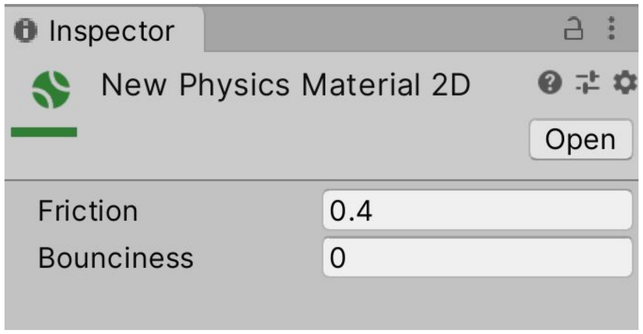
We won't dive into the *OnCollisionEnter2D* functionary in this lab; refer to the Colliders lab for a full breakdown. All you really need to know is that this prevents you from jumping when not touching the ground. Note that this isn't really the most ideal way to implement jumping, since a myriad of edge cases exist (can you think of a few?).

You'll have two options for the task in this section. You can try to implement a double jump, or you can try and disable horizontal controls while in the air. In both cases, *canJump* is going to be your best friend.

Part IV: Let's Bounce

Alright, let's get into a very useful part of Unity that goes sorely underused: Physics Materials. Physics Materials consist of two parts: bounciness and friction. They are ways for you to implement some physics based interactions into your objects without having to use code. Physics Materials can be attached to a Rigibody, in which case the Physics Material will be applied to all colliders. Or, you can attach different Physics Materials to different colliders on the same object for more refined interactions and results.

Change over to the Bounce! Scene to begin. In the top bar next to File and Edit, click on Assets>Create>PhysicsMaterial2D. You should see a new PhysicsMaterial2D show up in your assets like so:



Friction 0.4
Bounciness 0

The default values for friction and bounciness are 0.4 and 0 respectively. Hit play and watch as only the ball named Default bounces. It's the only one with a Physics Material so far, which has a bounciness of 0.9. Your task will be to attach different Physics Materials to each of the other balls, labelled 1 through 5 for you. 1 should bounce higher than Default. 2 should bouncer higher than Default but lower than 1. 3 should bounce lower than Default. 4 should bounce higher than 3 but lower than Default. 5...well, just make 5 as wacky as you can.

Part V: It's A Bit Rough

Let's take a look at the frictional components now. Open up the Quite Rough scene and hit play. You'll see the ball with a default of 0.4 friction rolling down a ramp of 0 friction. For those of you who need a review, here's the google definition of friction: it's the resistance that one surface or object encounters when moving over another.

Your task is to disable the sample ramp and finish the three distinct ramps. You'll be simulating ice, wood, and rough asphalt. During checkoff, show your TA the ball rolling down these three distinct surfaces.

Lab Summary

Now you know a bit more about how to implement more complicated physics in Unity, and how to use Physics Materials! Hopefully this knowledge will be useful for creating your own platformer, or any game that's influenced by gravity and physics.

Checkoff

- Make sure you have *move3()* completed and explain why one might use instantaneous velocity manipulation, and give a situation where it might not be advisable to do so
- Explain why multiplying the vector by a constant to speed up the movement might not be such a good idea when Time.deltaTime is used
- Show your double jump or disabled controls in the air
- Make sure you have the correct bounce heights for the balls
- Have all three surfaces ready to go and let the ball roll down; explain why you chose the materials you did to simulate this behavior