Rigidbody and Movement Lab (Advanced)

Introduction

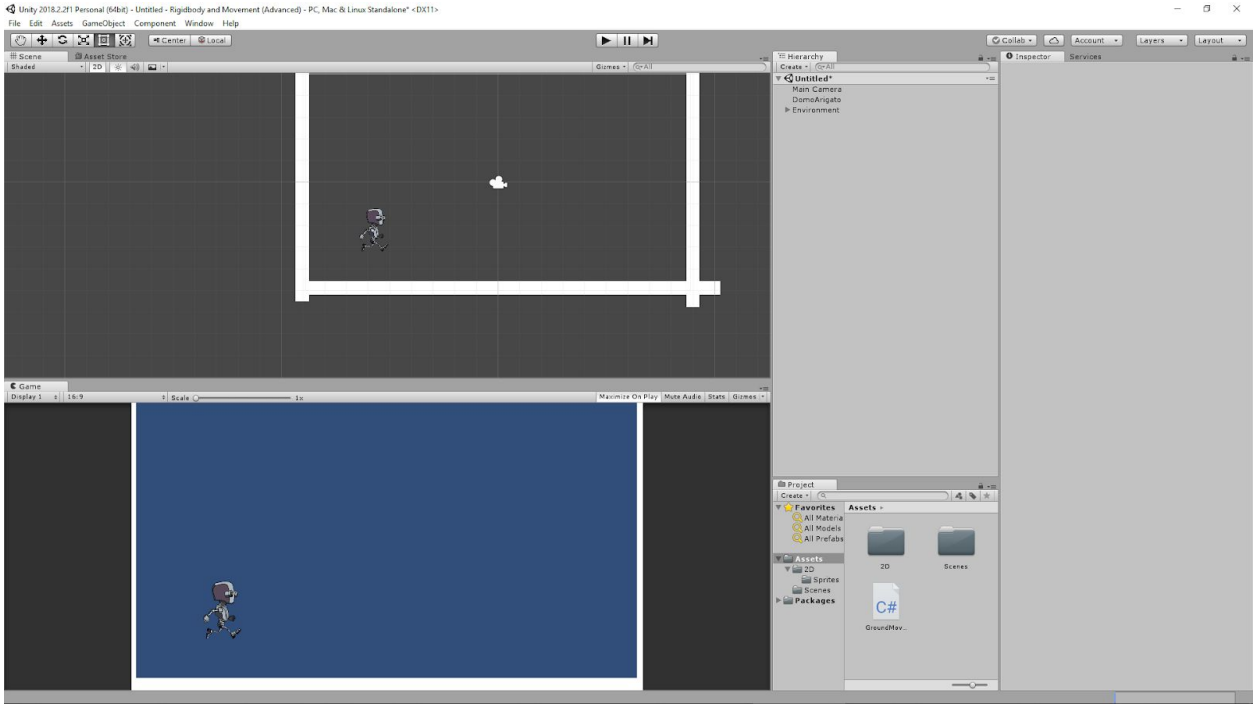
Welcome to the advanced portion of the movement lab! Here we'll be discussing smooth ground and aerial movement, some basic math and physics that will make your life a little easier, and give you a look into physics materials. Keep in mind this lab will assume you understand what we've already discussed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

Task One: 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 if 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:



The robot has a standard Rigidbody2D and a Box Collider 2D, while the box itself has Box Collider 2D as well. Make sure the robot's gravity scale is at 1, then hit play. You'll notice the robot doesn't really move once it hits the ground (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 Box Collider 2D 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 add a multiplicative scalar to 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 Update() function, change the last line from move1() to move2(). Hit play and move the robot left and right. Things should get pretty whacky, and you might even get the robot to clip out of the box. Uncomment the last line of move2() and try again.

```
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, 0, 3), robot.velocity.y);
}
```

In the last line of code, we're clamping the velocity Vector2, which after holding it down for even a brief moment will be (A very large number, 0). This particular clamp works by taking the very large number, and forcing it to be a number between -3 and 3. In general, Mathf.Clamp() takes in the first number as the number to be clamped, and makes sure the number is between a lower bound and an upper bound. If our velocity had been 2 and the clamp values were 0 and 3, nothing would have happened. -5 would be clamped to -3 and 5 would be clamped to 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. 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 Update() to move3!)

Task Two: 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 Update() to translate1(). 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.

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

Now scale up the movement by multiplying the vector by a constant (For the purpose of the lab choose a number between 10-15). 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 in place.

Task Three: 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:

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

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. Most video games a character's jump is similar to a spring that's been compressed suddenly being released. By manipulating the velocity directly, we can achieve the desired effect.

We won't dive into the OnCollisionEnter2D functionality 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 infinitely even 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?). We refer you again to the Colliders lab for a more in depth dive into Collider functionality.

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!

Task Three Point Five: Miscellaneous Sh...tuff

This section is mostly going to discuss various ideologies with regards to jumping and aerial mobility in some popular games, going over pros and cons as well as various mechanics that you may or may not want to implement later on in your games.

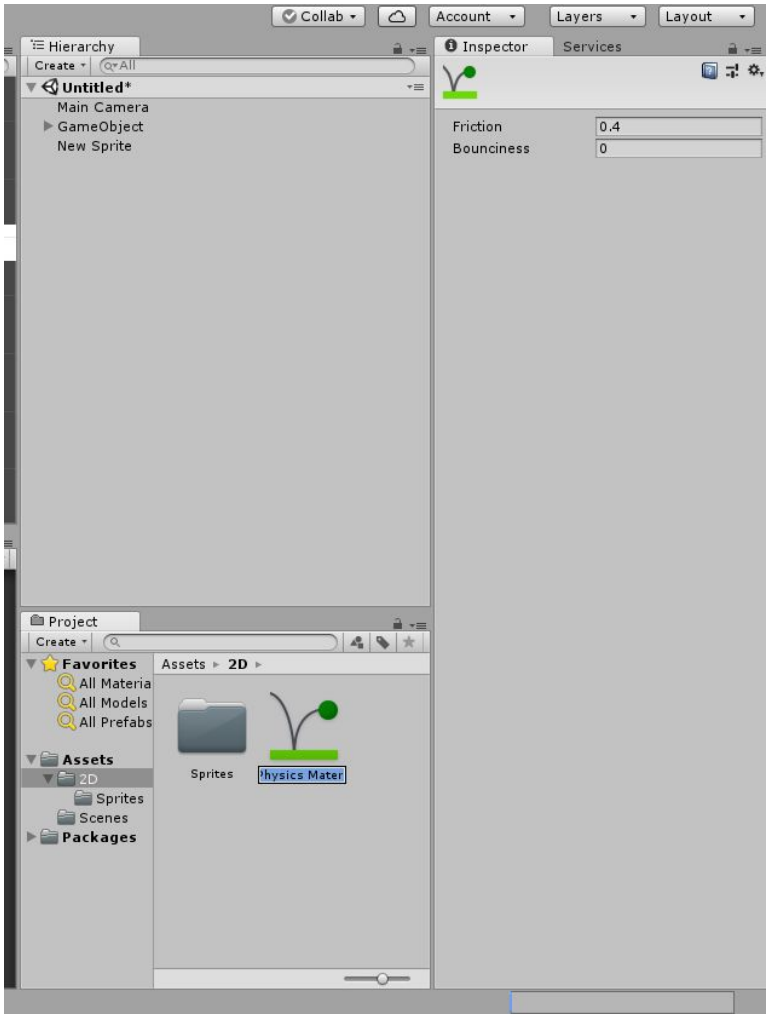
- Aerial Mobility: Mario is a perfect example of what it means to have high aerial mobility. Unless you have some special type of mechanic or reason why your character shouldn't have mobility while airborne, it's generally a good idea to give players some form of control while in the air. While it might not make the most sense from a physics standpoint, it generally feels pretty bad as a player to jump and feel completely useless until you land again. Remember, in most video games, movement is <u>POWER.</u>
- Ledge Forgiveness: This is the concept where games allow players a bit of leniency when jumping off the edge of a platform. Whether through slight hitbox manipulation or clever coding, this helps to empower the player by making them feel like they hit the perfect jump when in actuality they probably should have plummeted off the edge of the platform. This idea is also commonly known as "coyote time" (see any Bugs Bunny Cartoon for reference :D).
- Jump Height/Fall Speed: A staple in 2D games is to have proper jump heights and falling speed. These will heavily depend on what type of game you're trying to make, but in general, while again not making much physical sense, you want the player character to be able to jump a decent height, usually to at least their own head level. Falling speeds also need to be adjusted appropriately (whether through the manipulation of gravity or drag). Incorrectly balanced, some characters might feel too floaty or bulky (wouldn't want a giant burly character floating to the ground like a leaf while a piece of paper plummets to the ground almost instantly).

  Give your TA a quick rundown of what you learned for checkoff.

Task Four: 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, the bounciness and the friction. They are ways for you to implement some physics based interactions into your objects without going through copious amounts of math and coding. Physics materials can be attached to a Rigidbody, 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 -> Physics Material 2D. You should see a new Physics Material 2D show up in your assets like so:



The default values for friction and bounciness are 0.4 and 0 respectively. Hit play and watch as only the ball labeled Default bounces. It's the only one with a Physics material so far, with a bounciness of 0.9. Your task will be to attach different physics materials to each of the other balls, labeled 1-5 for you. 1 should bounce higher than Default, 2 should bounce higher than Default but lower than 1, 3 should bounce lower than Default, 4 should bounce higher than 3 but lower than Default, and 5...well just make 5 as whacky as you can.

Task Five: 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: the resistance that one surface or object encounters when moving over another. The more pressure there is between two objects, the higher the frictional force is going to be. Say you have a giant, rugged boulder moving

across asphalt. There's going to be more frictional force between the boulder and the asphalt then, for example, a small pebble getting dragged across the same asphalt.

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

Checkoff:

- Make sure you have move3() completed for task 1, tell your TA why one might use instantaneous velocity manipulation, and give a situation where it might not be advisable.
- For task two, 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 in place.
- For task three, you can try to implement a double jump, or you can try and disable horizontal controls while in the air.
- For task four, make sure you have all the correct bounce heights for the balls.
- For task five, have all three surfaces ready to go and let the ball roll down and explain why you chose the materials you did to simulate this behavior.