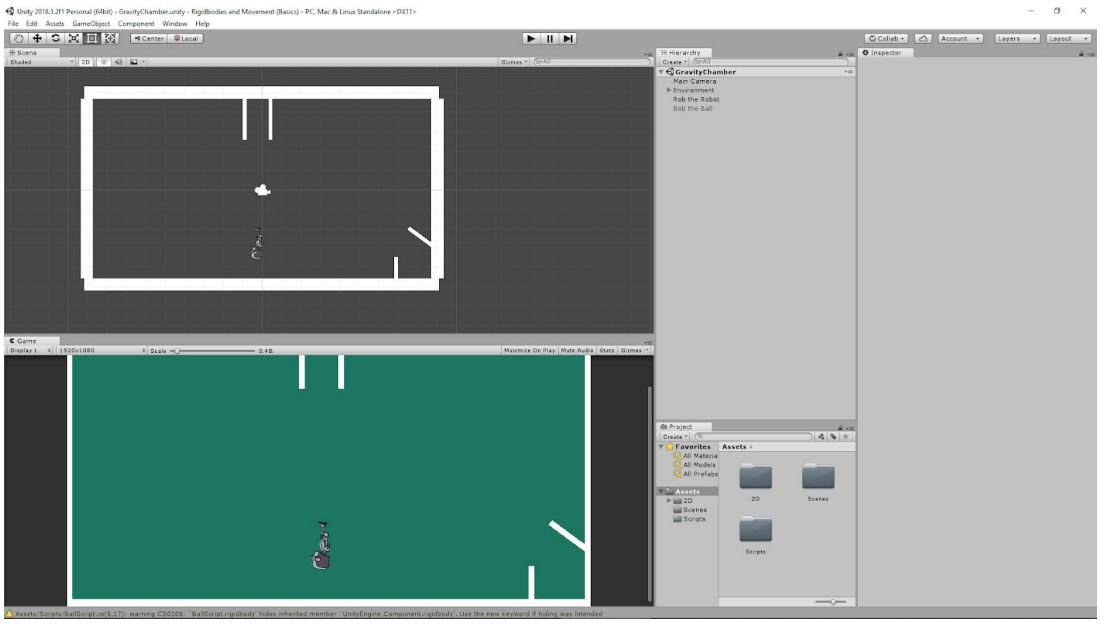
Rigidbody and Movement Lab (Basics)

Introduction:

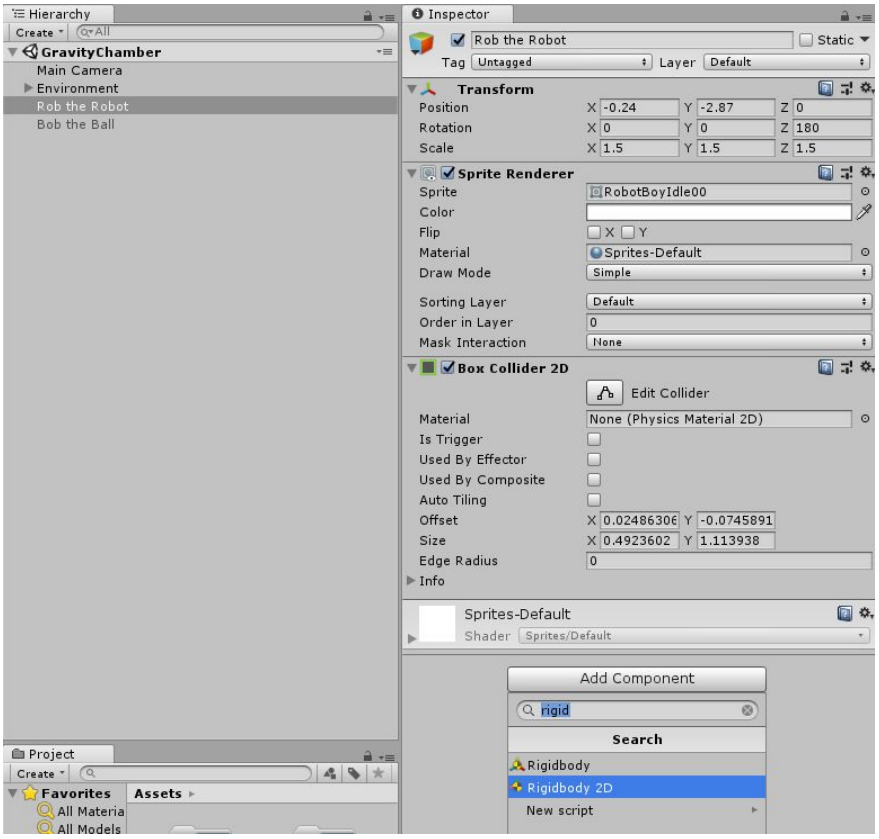
To apply any sort of physics to a GameObject in Unity, that GameObject is going to need a Rigidbody or a Rigidbody2D. In this lab we're going to be focusing on how we can use Rigidbodies to create physics based movement, and how we can tweak the physics of objects to obtain desired effects. Specific topics to be covered are gravity, mass and drag, basic control and acceleration/velocity, and finally rotation and body types.

Task One: Gravity Controls!

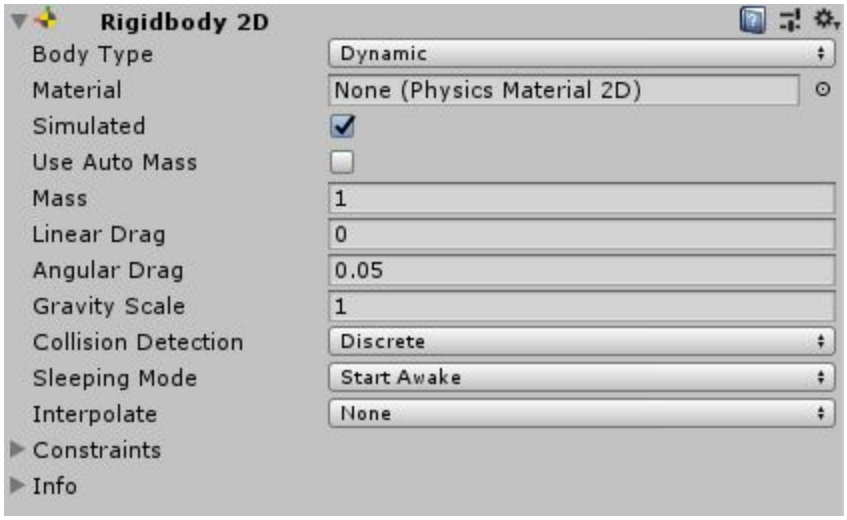
If the lab didn't start you off in the scene called "GravityChamber", swap to it now.



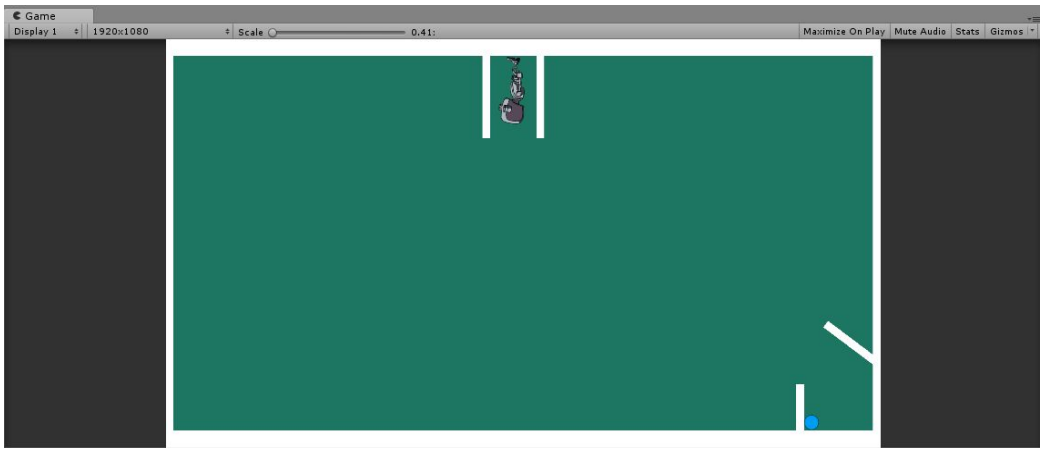
You should see something like this upon loading up the scene. Make sure the ball object is deactivated for this first part of the lab (you can deactivate/activate by clicking the box next to the object's name in the inspector above "Tag"). Navigate to Rob the Robot in the hierarchy, then in the inspector, click Add Component and add a RigidBody2D to it (note that you can also find it in the Physics2D section). See the picture below if you're not sure what to do.



After adding on the Rigidbody2D, you should see something like this show up. These are the default values that a Rigidbody2D start with. We'll go over each of these in depth a little later, but for now let's focus on the Gravity Scale. Feel free to hit play and mess around with the gravity settings to get a feel for how it works. Keep in mind gravity can also go negative or be noninteger values!



Now activate the ball object. You will see that it's already got a Rigidbody2D loaded up onto it. Upon hitting play, the ball will be launched to the right at a certain speed. Your job in task one is to adjust the gravity scales on both the ball and the robot so that the ball will end up in the bottom right corner of the box, and the robot ends up in the goal zone on the ceiling.

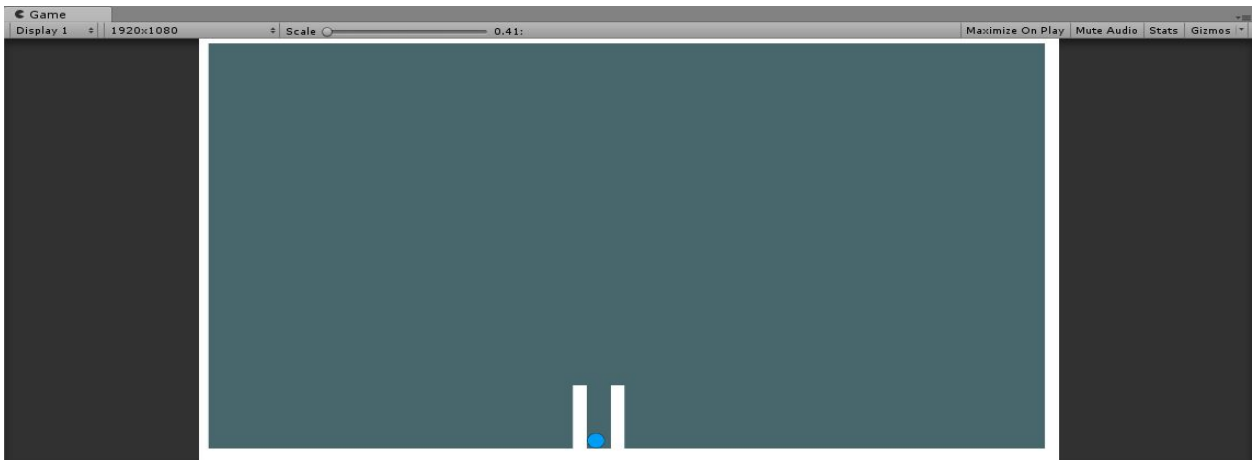


If you've managed to get something like this, great! Save the scene and move onto task two. (Please make sure you save the scene, we'd hate to see you move onto the next scene and realize at check off that you don't have any of the settings saved)

Task Two: A Massive Drag

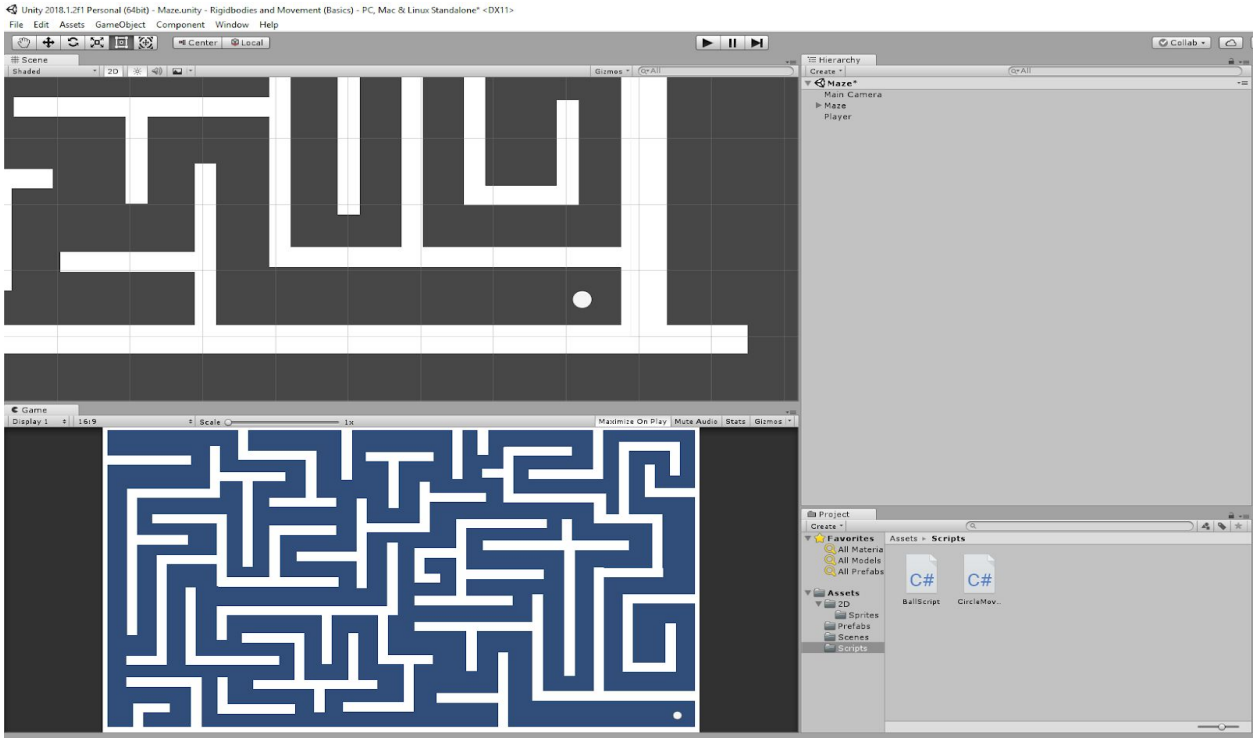
Swap over to the scene called "Drag Room". Here we'll be exploring linear drag and mass. **Drag** is a physics term, but in a game, drag is essentially a force which actively slows down an object over time. The **mass** of an object determines how forces will affect it. A more massive object will need more force to move. Take the ball in the scene, for example. The script moving the ball essentially adds a singular, horizontal push to the ball with constant force. Increasing or decreasing mass will affect the initial speed of the ball. Feel free to play around with the values (you'll find them in the Rigidbody2D right above gravity).

Your goal in this section is to get the ball into the goal area marked by the two columns by only editing linear drag and mass. Here's your goal. It's possible to reach the goal by only manipulating one of the values, but try to play around with the settings to get a feel for it



Task Three: Being Controlling

This section might be a bit dense, but is probably one of the most important in this lab. Go ahead and switch to the scene titled “Maze”. You should see something like this:



Later on, your goal will be to run the maze, getting the white ball to the red square. Now go ahead and open up the “CircleMovement” script. We’ll be focusing on the three movement functions, but familiarize yourself with the Update method as well. Let’s start with the first move function, shown below.

```
// Update is called once per frame
void Update () {
    xAxis = Input.GetAxisRaw("Horizontal");
    yAxis = Input.GetAxisRaw("Vertical");

    moveFunction1();

}

void moveFunction1() {
    Vector2 movementVector = new Vector2(xAxis, yAxis);
    playerRigidbody.AddForce(movementVector);
}
```

This function defines a 2D vector using our horizontal and vertical input axes (more on these later). The AddForce() function applies this vector as a force to the Rigidbody2D, moving it. If you want to visualize it, imagine a sphere suspended in a vacuum. AddForce() would be like gently tapping the ball. The ball is still subject to other outside forces, but until then, it will continue traveling in the same direction with the same speed. Think Newton’s 1st Law!

Try to run the maze with moveFunction1() as the function in Update() (no need to finish the maze just yet in this part). Note what was easy and what was difficult with this type of movement and control. Now let's look at moveFunction2() and moveFunction3(). **(Make sure to swap the move function in update!)**

```
void moveFunction2() {
    Vector2 movementVector = new Vector2(xAxis, yAxis);
    playerRigidbody.MovePosition(playerRigidbody.position + movementVector);
}

void moveFunction3() {
    Vector2 movementVector = new Vector2(xAxis, yAxis);
    playerRigidbody.velocity = movementVector;
}
```

moveFunction2() creates the same vector as the first function. However, there is one distinct difference, and that's the fact that this moves the actual rigidbody to the new coordinates specified by the vector. For example, if your gameObject is at (3, -3), pressing left and up simultaneously, the gameObject will move to (3-1, -3 + 1), or (2, -2). Go ahead and try to navigate the maze with this function.

If you're having a hard time with this one(I've had the ball clip out of the map, skip walls, etc), it's because the gameObject translation occurs in a single frame, and the update function loops extremely quickly. To fix this, we can scale the movement with time to have a smoother, more controlled translation. Add this line to moveFunction2():

movementVector = movementVector * Time.deltaTime;

To put it simply, this line scales your movement by the change in time, allowing for much smoother movement. This is a technique with wide ranging applications to deal with the fact that the update function is called many times per single second.

```
void moveFunction2() {
    Vector2 movementVector = new Vector2(xAxis, yAxis);
    movementVector = movementVector * Time.deltaTime;
    playerRigidbody.MovePosition(playerRigidbody.position + movementVector);
}
```

The last type of movement and control we'll go over is velocity control. Essentially, we'll be defining a Vector2 in much the same way as we've been doing, then set the velocity of our gameObject to be that Vector2 (friendly physics reminder that a vector has both a direction and a magnitude/speed). If you tried it right now, you might find that moveFunction3() feels almost identical to moveFunction2(). Note that while it might seem like these functions accomplish the same thing, one function moves the actual rigidbody, while the other instantaneously sets the velocity. To help scale your speed, you can add a multiplier to your vector to help speed things up (also keep in mind this type of vector scaling can be done with all your movement):

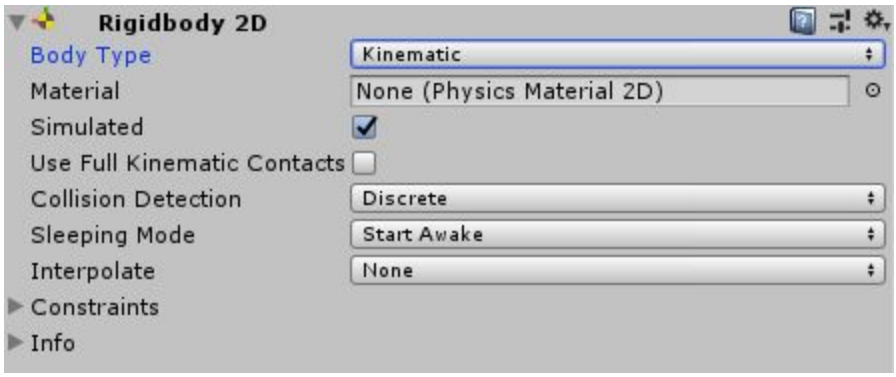
```
void moveFunction3() {
    Vector2 movementVector = new Vector2(xAxis, yAxis);
    movementVector = movementVector * 4;
    playerRigidbody.velocity = movementVector;
}
```

Now try to run the maze with whichever function you feel the most comfortable with, and be prepared to explain the differences.

Task Four: Body 'Em

In this section of the lab we'll go over different body types. Go ahead and save your previous scene and change over to "BodiesIntro". A Rigidbody2D's body type determines what type of forces you can or can't apply to it. There are three types of bodies: **Dynamic, Kinematic,** and **Static**.

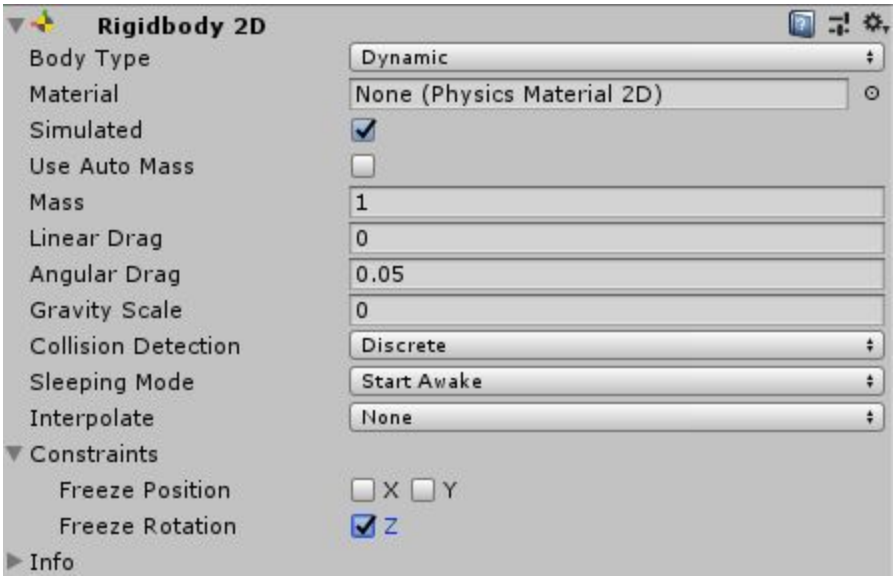
A dynamic object, which we've been using so far, can be affected by physics and scripts, while kinematic objects will only be affected by scripts and not physics. If you play around with the circle using different movement functions, you'll see that moveFunction1() doesn't work if the circle is kinematic, since it applies a physics force. However, the other two should work fine because they directly modify the object. Static objects cannot be moved. You'll notice that most of the options go away when you change to kinematic/static:



Dynamic objects are the best for things that are player controlled. Kinematic objects can be used for moving obstacles, platforms, or whatever you need to be moved that the player can interact with but not move outside of script interactions. Static objects should be used when you don't want anything moving (walls, the floor, etc).

From here on out, keep your circle as a dynamic object with gravity set at 0, and you may use whichever movement script you prefer. Using the circular player object, run into the robot, making note of if and how the robot responds when it has different body types.

You may notice that the robot tends to start rotating when you collide with it at certain angles while it is dynamic. To fix this, we explore the area of the RigidBody2D called

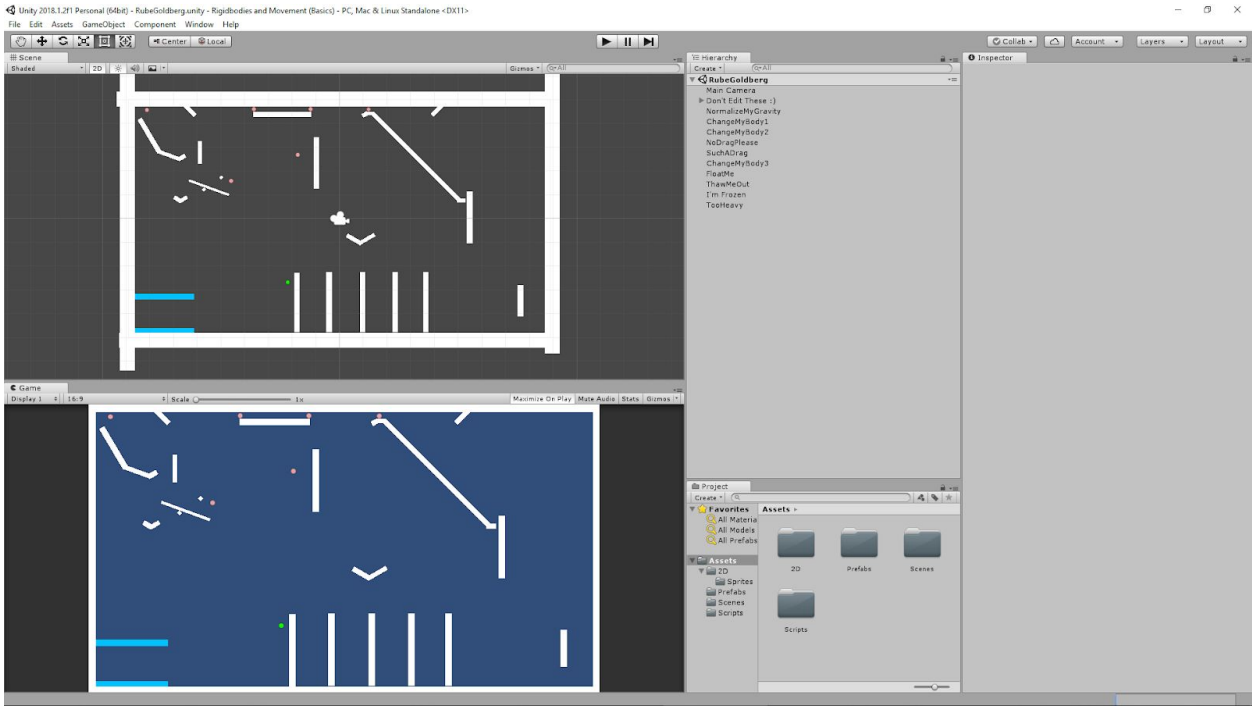


constraints.

By freezing the rotation of the robot around the Z axis, the robot will no longer rotate no matter the collision. These constraints will prove useful if you want to prevent characters from spinning in undesired ways.

Task Five: The Final Test

Welcome to the final portion of the lab which will require you to actually do stuff in Unity. Your task is to design your own Rube Goldberg machine, like the one detailed below. Hit play and watch one such machine in action. In your Rube Goldberg machine, you're required to have at least 5 static bodies, 5 dynamic bodies, negative/positive/null gravity, and to utilize drag, mass, and rotation freezing. Good luck!



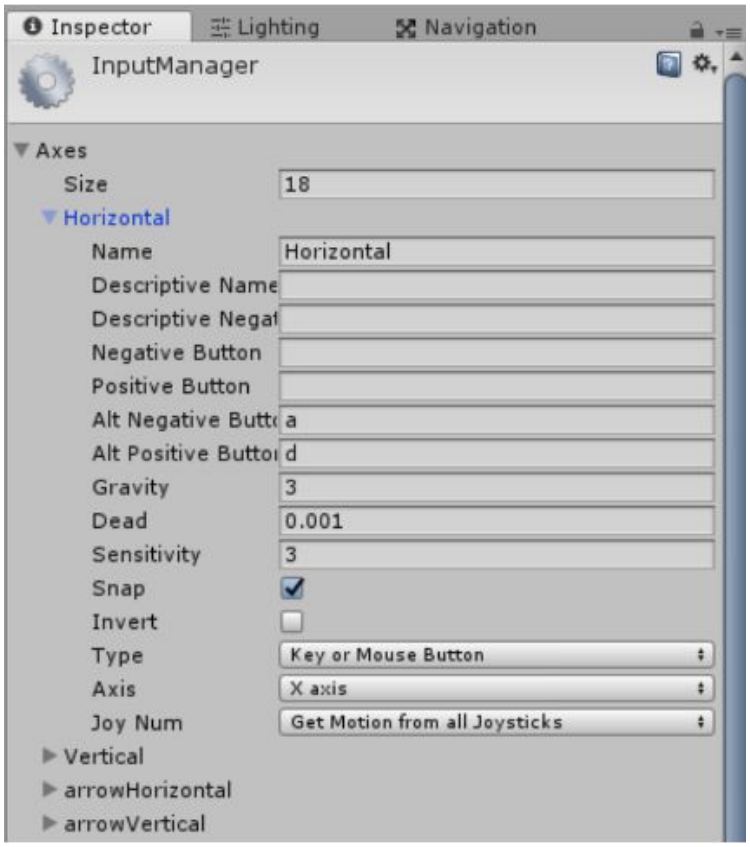
Task Six: Read Some VERY Important Stuff

All the things listed above are only part of what makes movement what it is. Physics can only take you so far. Here we'll be discussing inputs and how they will affect your scripts.

One very important aspect of scripting movement is getting the input right. In the previous portions, we've been using something called GetAxisRaw, which returns values of -1, 0, or 1. For example, on the horizontal axis, pressing the left key would give -1, right would yield 1, and not pressing anything would give a 0. GetAxis is another way to get inputs. It functions similarly to GetAxisRaw, however the values returned are conditional on how hard a button is being pressed or how far a joystick is moved. For all intents and purposes, GetAxis and GetAxisRaw are almost identical on keyboard, but joysticks and controllers will be affected by GetAxisRaw.

Another way to get inputs is to use GetButton and its variations (GetButtonDown, GetButtonUp, etc) as well as GetKey and its variations as well. These only return true/false instead of a number. You can also use GetMouse as well to receive mouse inputs. There are a large variety of Get_____ that you can use for your games, and it's highly recommended you take a look at the unity documentation for a better understanding of the power you have.

Another way you can adjust inputs is the actual sensitivity of the inputs themselves. Going to Edit ->, Project Settings, -> Inputs, you can see the Input Manager:



This is where everything about inputs can be changed. The name is what can be called from code. Negative and positive buttons are what are pressed to create the desired effect. For example, a typical Horizontal would be to have the left arrow (it would be called “left”) be Negative Button to go left (since negative values go towards the left) and the right arrow (it could
be called “right”) be Positive Button to go right (since positive values go toward the right). The names of the buttons can be found online, but for a keyboard, they just tend to be whatever the key is called.

Gravity is how fast an object will recenter and is only used when the keyboard and mouse are used. Dead is how much error Unity will tolerate before it begins an input. This is more applicable to controllers and joysticks, which might not always be at the center at 0, so this makes sure that the slightest offset doesn’t cause movement. Sensitivity is also related to how fast keyboard input is picked up by Unity. The higher the number, the faster Unity will process it. Snap determines if input will be set to 0 when opposite axes are being pressed. Invert will reverse the negative and positive axes. Type is whether or not the input is from mouse and keyboard or from a controller. Axis is what axis Unity will be receiving input for. Joy Num is used for controllers, especially is there are multiple controllers. Gravity, sensitivity and dead (on controllers) can all have an effect on how the controls feel, so be sure to keep that in mind.

If you want to get some advanced tips on fine tuned movement, make sure to check out the advanced version of this lab!

Check Off:

- Make sure you can do land the robot and ball in Task 1
- Make sure the ball lands in the goal zone in Task 2
- Make sure you can complete the maze and explain the different types of basic control
- Complete your own Rube Goldberg machine.
- Recap sections six and seven to the facilitator checking you off.