CAMOSUN COLLEGE

COMPUTER SCIENCE DEPARTMENT

ICS 123 - GAMING AND GRAPHICS CONCEPTS

LAB 8 – INTERACTING WITH GAME OBJECTS

DUE: DEMO BY END OF LAB MON MARCH 27
You must work individually for this lab.
Submission Instructions are on the last page of the lab.

Objective:

To have the students react and collect items in their Scene.

Preparation:

Supplemental Video Tutorials:

https://unity3d.com/learn/tutorials/topics/physics

Background and Theory:

At this point, there really isn't that many new concepts to learn. Everything that is done in this lab is basically applying things you have already learned, in a new way.

You've already used the *Rigidbody* Component on the *LaserBeam* so that it can detect Collisions properly. In this lab, you'll use the *Rigidbody* Component to activate Unity's Physics System so that Game Objects can react in a physically realistic way. You will also learn about using 'invisible' Game Objects to help detect where the Player is in the Scene. Finally, you will learn about 'collecting' Game Objects in the Scene. Although you won't implement a full inventory system, and thus you won't really be able to collect items, you will be able to use an item to increase the Player's health.

Unity API References for this Lab:

https://docs.unity3d.com/ScriptReference/CharacterController.OnControllerColliderHit.html
https://docs.unity3d.com/ScriptReference/Collider.OnTriggerEnter.html
https://docs.unity3d.com/ScriptReference/Collider.OnTriggerExit.html

Tasks:

Tasks Part 1 – Import Lab 7's Assets into a New Project

1. You will start a new Project for Lab 8 but you will want to import the Scene and all the other Assets from Lab 7. First, open your Lab 7 project in Unity. Go up to the menu bar and select *Assets -> Export Package*. This should open the Exporting Package window:

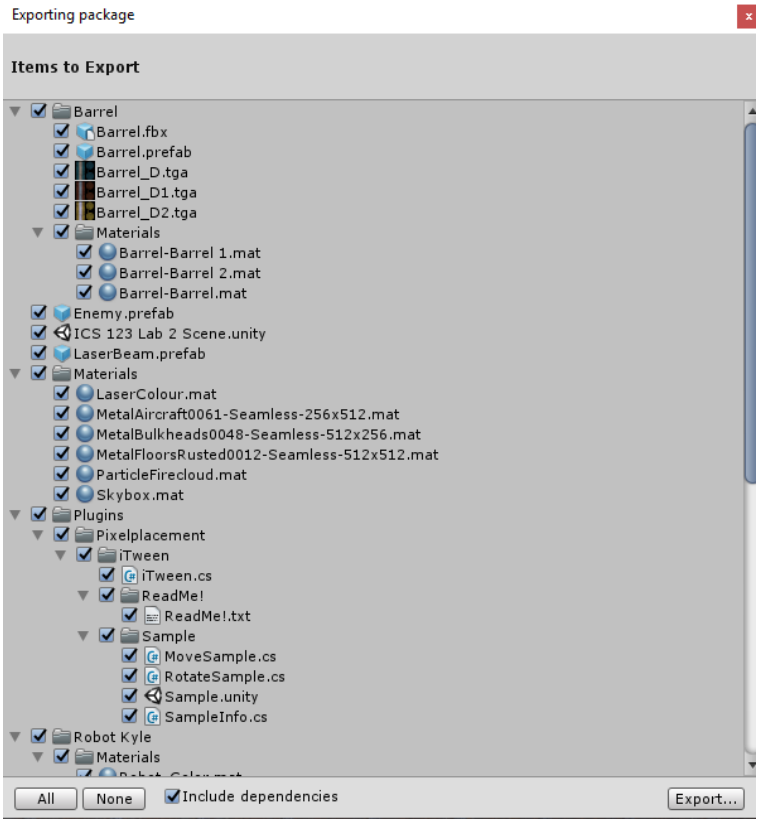

Figure 1: The Exporting package window

   Ensure that all your Assets are checked and that *Include dependencies* is checked. Hit the *Export…* button.

2. Save this package under your Lab 8 folder (create one if you haven't yet!). Save this new package as 'ICS123Lab7Assets'. A *.unitypackage* extension will automatically be added.

3. Now go up the menu bar and select *File -> New Project*. Name this new project 'ICS 123 Lab 8'. Hit the 'Create project' button. This will close your current Lab 7 Project and relaunch Unity with a new empty Lab 8 Project.

4. Once the new empty project is loaded in Unity, go up to *Assets* and select *Import Package -> Custom Package*. Browse for the package you created in Step 2. An Import Unity Package window will pop-up. Ensure everything is checked and hit the Import button.

5. In the Project Tab under Assets, you can double click on your Scene to load it up into the Scene Tab. You're done!

Note: You may want to disable the Enemies by commenting out the code in the `Update()` method of *SceneController* (select the code, right-click, select Toggle Line Comments) while Playtesting for this Lab.

Tasks Part 2 – Interacting with Physics-Enabled Objects

1. In order for a Game Object to utilize Unity's built-in Physics System, it must have both *Collider* and *RigidBody* Components attached. Add a new Prop from the Asset Store or use an existing Prop in your Scene to apply the Physics System to.

2. Add a *Collider* Component to your Prop if it doesn't have one. Then, add a *Rigidbody* Component. You shouldn't need to change anything in here except for the *Mass*. The *Mass* is the mass of the Object in kilograms. As far as Unity is concerned, it is used to determine just how much the Physics System affects the Object. For this example, the mass will be set to 10kg. Now make this Prop a Prefab (or if it already is a Prefab, apply your changes to it).

3. Make several instances of the Prefab and stack them up so that when the Player runs into them you can see the Physics System in action!

Figure 2: Stack of Crates that will have Physics Enabled

4. Open up your *FPSInput* Script in **MonoDevelop** (this is the Script you added to your Player way back in Lab 3 to control its movement). Add the following instance variable:

```
public float pushForce = 5.0f;
```

This variable is essentially how much muscle the Player has!

5. Now, add the following method:

```
void OnControllerColliderHit(ControllerColliderHit hit) {
    Rigidbody body = hit.collider.attachedRigidbody;
    // does it have a rigidbody and is Physics enabled?
    if (body != null && !body.isKinematic) {
        body.velocity = hit.moveDirection * pushForce;
    }
}
```

The method will get called automatically whenever the Player Game Object collides with something. The first statement gets a reference to the `Rigidbody` component of whatever object the Player collided with. The `if` statement checks to make sure this isn't `null` (i.e. that the Object has a `Rigidbody`) and that `isKinematic` is `false` (a Game Object can have a `Rigidbody` Component but disable the Physics System by checking the `Is Kinematic` box in the Inspector Tab). If this is the case, it applies a force to the Game Object and causes it to move.

6. Play the Scene and run into whatever stack of Props you created. You should see them physically react to your Player pushing them!

Tasks Part 3 – Interacting with a Door

Tasks Part 3a – Adding a Door to the Scene

1. First, you need a door! You have two options for a door, either import the custom package that you can download from D2L (*metaldoor.zip*) or find your own free door in the Unity Asset Store.

2. Next, position the door somewhere in your Scene (obviously where there's a gap between two walls). You'll probably need to change its X, Y, and Z scale. If it's going to be a sliding door, make it less thick than the wall so it can slide into it. You also may want to add some *Indirect Guidance* for the Player by adding a point light to highlight the door.

Figure 3: The Metal Door Placed in the Scene with a Point Light for Indirect Guidance

In the figure above, a position of 0, 0, -15; a rotation of 0, 90, 0; and a Scale of 100, 23, 32 was used.

3. Now, figure out the position of the door when it is open. In this example, I want my door to slide to the left. This puts it at positon 4.5f, 0, -15. Yours may not be the same. Note down the open position and then place the Door back in its closed position.

To make it a nice, fluid movement, we'll use a Tween! iTween, which you first used back in Lab 4, provides a method called `MoveTo` that will do this for us. Create a new C# Script and call it *DoorControl*. Make it a Component of the Door Game Object. Open the Script in **MonoDevelop**.

4. Add a private Boolean instance variable:

```
private bool doorIsOpen;
```

5. In the `Start()` method, set it to false:

```
doorIsOpen = false;
```

6. Add a new method called `Operate()`:

```
public void Operate() {
    if (doorIsOpen) {
        iTween.MoveTo (this.gameObject, new Vector3 (0, 0, -15f), 5);

    } else {
        iTween.MoveTo (this.gameObject, new Vector3 (4.5f, 0, -15f), 5);
    }
    doorIsOpen = !doorIsOpen;
}
```

Figure 4: The `Operate()` Method

The `Operate()` method is fairly straight forward. If `doorIsOpen` is `false`, which is the case when the Door is closed, the `if` statement will be false and the `else` clause will execute and move the door to the open position. Using the Tween, it will do this over a period of 5 seconds (the last parameter). The boolean `doorIsOpen` variable keeps track of which state the door is in. When `false`, the door is closed and when `true`, the door is open. So every time the `Operate()` method is called, the `!` operator in the last statement switches `doorIsOpen` from true to false or vice versa. The astute of you might recognize that this is the same sort of two-state FSM we implemented for the Enemy's dead or alive state. The one here is coded more succinctly at the cost of readability.

Tasks Part 3b - Triggering the Door to Open and Close

1. Now that you have the door placed in the Scene and a method to open and close it, you need a way to trigger this to happen! There's a couple of different ways you could do this, depending on what your needs are. But the way shown here is probably one of the simplest. Go up to the Menu Bar and select *GameObject -> 3D Object -> Cube*. Re-name it to *DoorTriggerZone*.

2. The cube is going to be the area which will trigger the door to operate. Position and scale the cube so that it surrounds the door and the area around it. In this example, a position

of 0, 2, -15 and a scale of 5, 4, 5 works well.

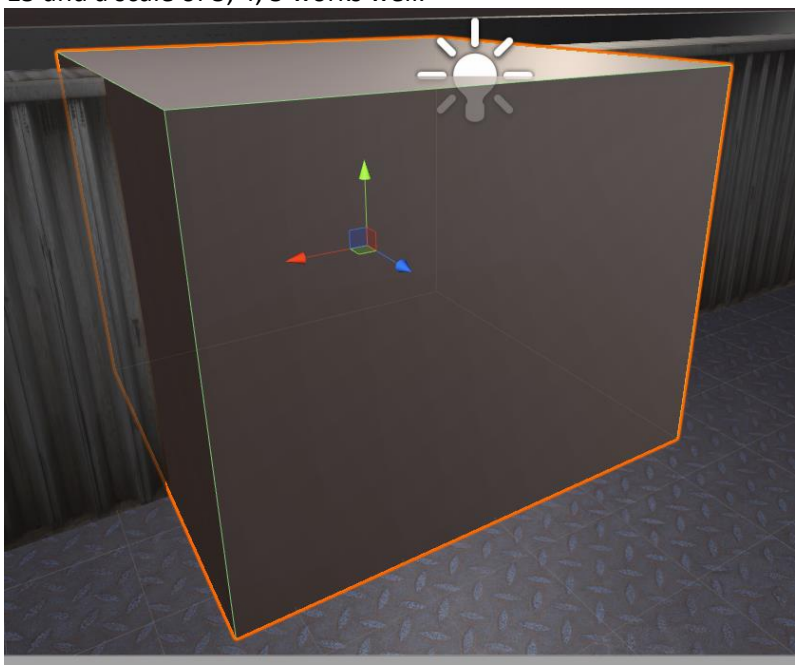

Figure 5: The Area which will 'Trigger' the Door

3. For obvious reasons, you don't want this cube to be rendered in the Scene (but you still want it to be there). To make it invisible, simply *delete the Mesh Renderer Component*.

4. Now if you play your Scene and head for the door, you won't see the Cube but you can shoot at it and walk into it like a solid object. This is no good! What we need are two things: first, that we can't shoot at the cube and we can walk through it (so it truly is invisible in the Scene), and second, we want it to generate a trigger when we collide with it. To make this happen, there are two things you need to do:
    a. We'll take advantage of Unity's Layering System. With the Cube selected and in the upper right of the Inspector Tab, below the Static label, you'll see a Layer label with a dropdown. Change this to *Ignore Raycast*. This means that any rays cast on this object will simply go through it.
    b. Check the *Is Trigger* box in the *Box Collider* Component. This allows objects to pass through it but it still causes a trigger when a collider touches it.

    If you play the Scene again, you will be able to walk through the cube.

5. Create a new C# Script and call it *DeviceTrigger*. Attach it to the *DoorTriggerZone*. Then, double-click on it to open it in MonoDevelop.

6. First, add a `SerializeField` private instance variable to hold a reference to the Door Game Object:

```
[SerializeField] private GameObject device;
```

7. Next, add the `OnTriggerEnter` and `OnTriggerExit` methods that will be called automatically when something with a collider enters and exits the trigger area:

```
void OnTriggerEnter (Collider other) {
    DoorControl door = device.GetComponent<DoorControl> ();
    if (door != null) {
        door.Operate ();
    }
}

void OnTriggerExit (Collider other) {
    DoorControl door = device.GetComponent<DoorControl> ();
    if (door != null) {
        door.Operate ();
    }
}
```

The code is fairly straightforward. First, we use `GetComponent` to get a reference to the *DoorControl* script. Then we call `Operate()`. The code for `OnTriggerEnter` and `OnTriggerExit` are identical. This isn't ideal – you have replication of code. But the *DeviceTrigger* script could be used as a single script to trigger several devices. The `device` variable could be changed to an array, and the different methods could do different things to multiple devices at the same time!

8. Play the Scene. You should see the Door automatically open and close when your Player approaches the door.

Tasks Part 4 – 'Collectible' Items

1. A common feature of most games is the ability for the Player to pick up items. These items typically provide some sort of advantage to the Player: extra health, more ammunition, better weapons, money, etc. For this lab, we'll use the example of picking up extra health.

2. Go up to the menu and select *GameObject -> 3D Object -> Sphere*. Rename it to *HealthItem*. Place the Sphere somewhere in your Scene so that it is at about waist level for the Player. Scale it down and shape it like a disk (I used a scale of 0.1, 0.5, 0.5).

3. Create a new Material. Go up to the menu and select *Assets -> Create -> Material*. Call it *HealthItemColour*. Apply it to the *HealthItem* you created above (just drag the Material onto the Sphere in the Scene). Change the colour to what you want (the *Albedo*). You might also want to add a bit of a glow to it to make it easier to spot. Change the *Emission* value to around 0.5. You can also change the Emission colour.

Figure 6: The Health 'Vitamin' Placed in the Scene

4. Create a new C# Script and name it *CollectibleItem*. Open it in **MonoDevelop**. To draw the Player's attention to it, let's have it rotate. In the `Update()` method, add the following statement:

```
transform.Rotate (0, 3, 0);
```

   Attach the Script to the *HealthItem* Game Object. If you play the Scene, you should see the 'Vitamin' rotate 3 degrees / frame.

5. Just like you did with the *DoorTriggerArea* object from the last task, change the Layer of the *HealthItem* to *Ignore Raycast* and check the *Is Trigger* box in the Sphere Collider.

6. One other thing that you might want to change is to turn off shadowing with the *HealthItem*. Shadowing is computationally intensive for graphics hardware. Since this is a 'collectible' item and not a 'true' Game Object in the Scene, we can save some resources by not using shadowing here. In the *Mesh Render Component* of *HealthItem*, change Cast Shadows to Off and uncheck Receive Shadows.

7. Add the following method to the *CollectibleItem* Script:

```
void OnTriggerEnter(Collider other) {
    PlayerCharacter player =
        other.GetComponent<PlayerCharacter> ();
    if (player != null) {
       // apply first-aid and remove the item
       player.FirstAid ();
       Destroy (this.gameObject);
    }
}
```

Figure 7: The `OnTriggerEnter` method in CollectibleItem

   Unlike the door, we only need the `OnTriggerEnter` method here. This method will check to see if the Player is the one who collided with it. If so, call the `FirstAid()` method and remove the *HealthItem* from the Scene.

8. Now you have to think a bit about Game Balance. What will the first-aid do? How many health points will it add back? In this example, a `maxHealth` variable is added in the *PlayerCharacter* Script and made public so it can be adjusted in the Editor for Playtesting:

```
public int maxHealth = 5;
```

   In this case, the `maxHealth` is set to five.

9. Also in the *PlayerCharacter* Script, add the `FirstAid()` method:

```
public void FirstAid() {
    if (health < maxHealth) {
        health += 2;
        if (health > maxHealth) {
           health = maxHealth;
        }
        Messenger<int>.Broadcast (GameEvent.HEALTH_CHANGED, health);
    }
}
```

Figure 8: The `FirstAid()` method in PlayerCharacter

If the Player's health is below `maxHealth`, add two to the `health` (you can choose a different value). If this puts it above `maxHealth`, make health equal to `maxHealth`. Then, broadcast an event that the health has changed (this will be the same event as what's in the `Hit()` method in *PlayerCharacter*).

10. The *Listener* for this Event is in *UIController*. You may need to make changes there to ensure that if the Health increases, the *HealthBar* will properly change color.

11. Make a Prefab out of your *HealthItem* (I really hope you know how to do this by now!). Place a few around the Scene. Play your Scene and you should see your Player regain Health when it walks into one of the Health 'Vitamins' in the Scene. The health bar should properly reflect the change in Health!!

Submission:

Demo your Scene to the Lab Instructor:

- **Show that your Player can walk into a stack of props and they react using Unity's Physics System.**
- **Show that your Player can walk up to a door and it automatically opens and closes.**
- **Show that the Player can walk into a Health Item and that the Player's Health increases, the Health Bar properly changes color, and the Health Item is removed from the Scene.**

- A successful demo scores you 10 marks.

Note and Project Idea:

We didn't implement a full inventory system in this lab. As soon as the Player walks into a Health Item, it is applied right away so it isn't truly collectible. *You could add an Inventory system that would allow the Player to collect Health and other Items*. This would require an addition to the HUD and some supporting code to keep track of the items the Player has collected. You'd have to use the Event System for the HUD and supporting code to communicate.