CAMOSUN COLLEGE

COMPUTER SCIENCE DEPARTMENT

ICS 123 - GAMING AND GRAPHICS CONCEPTS

LAB 4 – COLLISIONS, ENEMIES, AND PROJECTILES

DUE: DEMO BY END OF LAB PERIOD MON FEB. 20
You must work individually for this lab.
Submission Instructions are on the last page of the lab.

Objective:

To have the students add collisions, enemies, and projectiles to their 3D Space that they have been building since Lab 2.

Preparation:

Supplemental Video Tutorials:

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

Background:

Collision Detection:

GetComponent( ):

In Lab 3, you added the capability for your Player to walk around your 3D Space but you probably noticed that the Player was a ghost (the Player could walk through walls!!). To prevent that from happening, you need to add collision detection. You already added a component to your Player object that can handle that, the *CharacterController* component. What you need to do is use this component in your movement code.

To do so, you will have to create a reference to the *CharacterController* object in your code. You didn't need to do that for the Transform component because it is a mandatory component and every game object has a reference to a Transform object as a property (called `transform`). Luckily, there is a method available that you can call which returns a reference for any attached component:

```
GetComponent<CharacterController>();
```

This syntax may look unusual to you. `GetComponent()` is called a ***generic method***. Broadly speaking, it means `GetComponent()` can work with several related yet unique objects. In Unity, it means that the same method can be called to fetch a reference to any component

currently attached to the Object. You simply set the type of the object (the component) in the angle brackets.

Move( ):

Once you have a reference to *CharacterController*, you can call its `Move()` method to move the Player. The `Move()` method expects a `Vector3` struct (see Lab 3 if you forget what a Vector is).

There's two issues you need to deal with. The first issue has to do with the magnitude if the Player moves diagonally. Recall in Lab 3 that you got the change in magnitude for X and Z by using the following statements:

```
float deltaX = Input.GetAxis("Horizontal") * speed;
float deltaZ = Input.GetAxis("Vertical") * speed;
```

`Input.GetAxis()` will return a value between 1 and -1. The number is dependent on the length of time a key is held down during the frame. For example, if the W key was held down for the entire frame, it would return a 1. Multiply that with a speed factor of 9 means the Player can move a maximum of 9. This would also apply to the S, A, and D keys. But what if both a Horizontal and Vertical key were pressed at the same time? For example, what if the W and A key were both held down the entire frame? The Player would move diagonally. But what would the magnitude be?

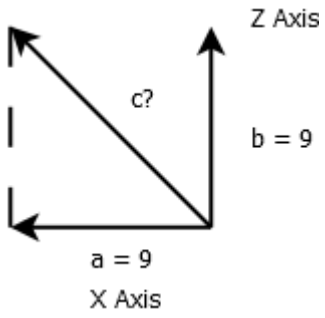

Figure 1: Moving Diagonally

You can find c by using Pythagorean Theorem:

$$c = \sqrt{a^2 + b^2}$$

Plugging in the numbers you get c = 12.73! This would be really weird in a game scenario as any diagonal component to a Player's movement would cause the Player to move at a faster rate (you can verify this in Lab 3). We can fix this problem by using `ClampMagnitude(vector, maxLength)` where we clamp the magnitude of any vector to the speed factor.

The other issue with the `Move()` function has to do with local vs. global coordinates (if you forget the difference, see Lab 3!). The Player moves according to its local coordinates

however, the `Move()` function requires global coordinates to detect collisions with objects in the Scene. Luckily, Unity provides a function called `TransformDirection()` that converts local coordinates into global ones.

Gravity:

Assuming that you don't want your Player to fly, you need to specify a gravity value (a downward force on the Y axis). One tricky thing here is that you want that force to always be pointing straight down regardless of the Player's tilt (when the Player uses the mouse to look up and down, the Player object tilts). To solve this problem, you will attach your *MouseLook* script to two different objects. In Lab 3, you already had it attached to the Player object with it set to horizontal rotation (Mouse X). That you can leave. Now, you will also attach your script to the camera object itself. This one will be set to vertical rotation (Mouse Y) so that when the Player looks up and down the Player object doesn't tilt, just the camera!

Tasks Part 3 explains how to add *Collision Detection* to your Player.

Raycasting:

Raycasting is when you introduce a ray into your Scene. What is a ray? A ray is an invisible line that starts at an origin point and goes in some direction. For example, if the Player were holding a gun and fired a bullet, the bullet's path in the Scene is the ray. Rays, by definition, have no end point; they are infinite. But in a 3D Scene like you created in Lab 2, the ray will eventually hit something (probably a wall at this point)! Raycasting is all about figuring out what and where that ray hit.

In this Lab, you are going to have your Player fire a gun! In a first-person shooter scenario, the bullet (ray) typically originates from the centre of the camera view.
`Camera.ScreenPointToRay()` takes a Vector at a given point and generates a ray.
`Physics.Raycast()` does the magic – it figures out what the ray hit in your Scene.

Tasks Part 3 adds *Raycasting* to your Scene.

Coroutines:

Recall that the `Update()` method is called once per frame. Any code (including other method calls) must finish before the next frame. But what if you want some code to happen over several frames? Unity Coroutines solve this problem.

Coroutines are able to pause their execution until the next frame. Here's how you create and use Coroutines:

- Put the code you want to execute over several frames in its own method. Set the return type of this method to `IEnumerator`. `IEnumerator` basically keeps track of the code to be run each frame.
- Call this method in the `Update()` method by wrapping it in a call to `StartRoutine()`.
- Finally, in your coroutine, use a `yield` statement where you want the method to pause and finish execution for that frame. On the next frame, execution will continue on the statement after the `yield`. Alternatively, you can call yield with a `WaitForSeconds` object to have execution pause over several frames for the

given amount of seconds.

If this seems a bit confusing, don't worry! You'll see an example of a coroutine in Task 3.

Heads Up Display (HUD):

Practically every game, whether 2D or 3D, requires some way to provide information about the status of the game to whomever is playing the game. This is typically accomplished with a HUD or Heads Up Display. A HUD is essentially a 2D GUI (Graphical User Interface) that superimposes over the view of the world. In Unity, there are two types of 2D GUI's: the *immediate mode* GUI and the *retained mode* GUI. The *immediate mode* GUI draws the HUD over the view of the world after it has been rendered (drawn on screen) every frame. The method `OnGUI()` does this. Like `Update()`, the `OnGUI()` method, if added in your Script, will run once every frame. *Retained mode* works differently, you need to define the visuals only once and they will automatically get re-drawn every frame. We'll look at *retained mode* later in the course.

In this lab, you'll use the *immediate mode* GUI. You won't be making a full HUD quite yet, but you will create a very (very!) simple 2D GUI that contains an aiming spot.

Unity's Asset Store:

Anything that you see in the Project Tab of the Unity Visual Editor is called an asset. This includes any 2D images, 3D models, scripts, scenes, and more! Unity provides a library where you can download pre-made assets; both free and paid. The Asset Store is located at https://www.**assetstore.unity3d**.com/ but it is much easier using it directly inside Unity.

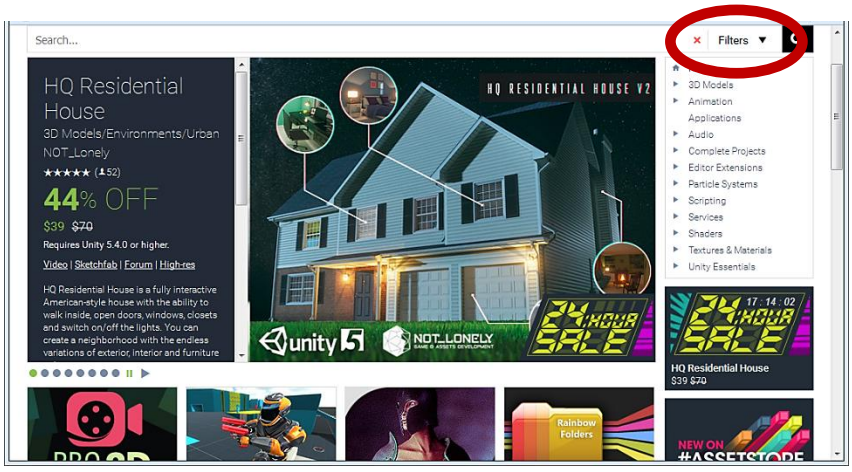

Figure 2: Unity's Asset Store

To open the Asset Store in Unity's Visual Editor, simply click on the Asset Store Tab in the centre of the Editor. If for some reason you don't see it, hit CTRL+9 to open it. Circled in Figure 2 above is the Filters control. Hitting this button opens up filtering options. For example, if you want to see only free Assets, you can hit the *Free Only* button:

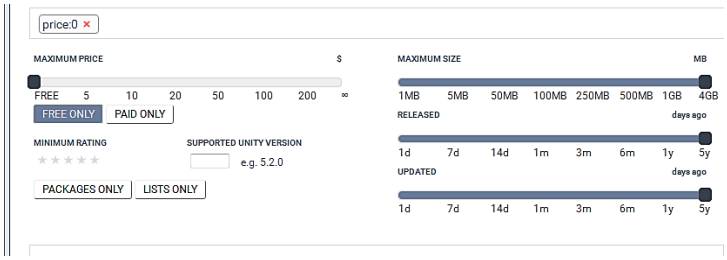

Figure 3: Asset Store Filter Controls with Free Only button activated

Unity API References for this Lab:

https://docs.unity3d.com/ScriptReference/GameObject.GetComponent.html
https://docs.unity3d.com/ScriptReference/CharacterController.Move.html
https://docs.unity3d.com/ScriptReference/Vector3.html
https://docs.unity3d.com/ScriptReference/Input.GetAxis.html
https://docs.unity3d.com/ScriptReference/Transform.TransformDirection.html
https://docs.unity3d.com/ScriptReference/Input.GetMouseButtonDown.html
https://docs.unity3d.com/ScriptReference/Camera.ScreenPointToRay.html
https://docs.unity3d.com/ScriptReference/Physics.Raycast.html
https://docs.unity3d.com/ScriptReference/RaycastHit.html
https://docs.unity3d.com/ScriptReference/Coroutine.html

Tasks:

Tasks Part 1 – Import Lab 3's Assets into a New Project:

1. You will start a new Project for Lab 4 but you will want to import the Scene you created from Lab 3. First, open your Lab 3 project in Unity. Go up to the menu bar and select *Assets -> Export Package*. This should open the Exporting Package window:

Figure 4: 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 4 folder (create one if you haven't yet!). Save this new package as 'ICS123Lab3Assets'. 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 4'. Hit the 'Create project' button. This will close your current Lab 3 Project and relaunch Unity with a new empty Lab 4 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!

Tasks Part 2 – Adding Collision Detection to the Player's Movement:

1. Double click on your *FPSInput* C# Script in the Assets window of the Project Tab to open it in **MonoDevelop**. The first thing you want to add to your script is to declare and initialize a reference to the *CharacterController* component. Add the following just below your declaration of the speed factor:

```
private CharacterController _charController;
```

Then, in your `Start()` method get the reference:

```
_charController = GetComponent<CharacterController> ();
```

For an explanation of this code, see *Collision Detection* of the *Preparation* section of this Lab.

2. Change your `Update()` method to look like the Figure on the following page.

```
void Update () {
    float deltaX = Input.GetAxis("Horizontal") * speed;
    float deltaZ = Input.GetAxis("Vertical") * speed;

    Vector3 movement = new Vector3 (deltaX, 0, deltaZ);

    // Clamp magnitude for diagonal movement
    movement = Vector3.ClampMagnitude (movement, speed);

    // Movement code Frame Rate Independent
    movement *= Time.deltaTime;

    // Convert local to global coordinates
    movement = transform.TransformDirection (movement);

    _charController.Move (movement);
}
```

Figure 5: Collision Detection Code

The big change here is you are going from moving the *Transform* object attached to your Player that you did in Lab 3 to moving the *CharacterController* object instead (for collision detection). For an explanation of this code, see *Collison Detection* of the *Preparation* section of this Lab.

3. Save your script and close **MonoDevelop**. Click on the Player object in the Hierarchy Tab. In the Mouse Look Component of the Inspector Tab, change the Axes dropdown to *Mouse X*. Now play your Scene and move around. Does your Player still act like a ghost?

4. Stop the Scene and click on your Player object in the Hierarchy Tab. In the Mouse Look Component of the Inspector Tab, change the Axes dropdown to *Mouse X and Y*. Now play your Scene again and this time look up and down and move around. Your Player won't be able to go through the floor due to collision detection BUT you will notice that your Player can fly! For this lab, we don't want that behaviour so we'll add some gravity to the situation!

5. Stop your Scene and double click on FPSInput to open it in **MonoDevelop**. Add the following declaration at the top of the class:

```
public float gravity = -9.8f;
```

`gravity` is a negative value because we want to 'push' the Player towards the ground (-ve Y axis). The value 9.8 was chosen because that matches the force of gravity on Earth (9.8m/s). Now in your Update method, add the following statement after you clamp magnitudes and before you apply deltaTime to apply your gravity:

```
movement.y = gravity;
```

6. Save your script and close **MonoDevelop**. Click on the Player object in the Hierarchy Tab. In the Mouse Look Component of the Inspector Tab, change the Axes dropdown to *Mouse X*. Expand your Player object in the Hierarchy Tab by clicking on the small arrow on the left so that the Main Camera object appears in the Hierarchy Tab.

7. Drag your MouseLook Script from the Assets window of your Project Tab and drop it on the Main Camera in the Hierarchy Tab. Change the Axes drop down in the Mouse Look Component of the Hierarchy Tab for **the Main Camera** to Mouse Y (see *Collison Detection* of the *Preparation* section of this Lab to see why you need to do this). Thus, you now have two instances of the same Script. One script is attached to the Player object and is in charge of X axis mouse input and horizontal looking around. The other script is attached to the Camera object (which, in turn, is attached to the Player object) and is in charge of Y axis mouse input and vertical looking around.

8. Now Play your Scene. You should be able to both look and move all around now without leaving the floor and going through walls!!!

Tasks Part 3 – Adding a Projectile to Your Scene (Firing a Gun):

1. Create a new C# Script and name it *RayShooter*. Attach this Script to the Camera not the Player. Double click on it to open it in **MonoDevelop**.

2. You need to programmatically access the Camera object so you need to retrieve the reference to the Camera just like you did for *CharacterController* in Tasks Part 2, step 1. Therefore, declare a variable for the Camera:

```
private Camera _camera;
```

   And retrieve the reference in the `Start()` method using `GetComponent()`:

```
_camera = GetComponent<Camera> ();
```

3. Now, modify your `Update()` method to look like the Figure below:

```
void Update () {
    if (Input.GetMouseButtonDown (0)) {
        Vector3 point = new Vector3 (_camera.pixelWidth / 2,
            _camera.pixelHeight / 2, 0);
        Ray ray = _camera.ScreenPointToRay (point);
        RaycastHit hit;
        if (Physics.Raycast (ray, out hit)) {
            // visually indicate where there was a hit
        }
    }
}
```

Figure 6: Raycasting Code (Incomplete)

The code is fairly self-explanatory. First, it checks to see if the left mouse button (button 0) has been pressed. This will be the fire button. If it has, it generates a Vector with a point in the centre of the camera view. The Z axis is 0 because we only care about the X and Y coordinates as the Ray will get projected in the Scene along the Z axis of the camera. The next statement does just that, it generates the ray. `RaycastHit` is a struct that stores information about a Raycast. `Physics.Raycast()` uses it to store that information. The `out` modifier means that you want `Physics.Raycast()` to modify the `hit` struct directly, not a copy of it (i.e. pass-by-reference instead of pass-by-value).

4. The code from Figure 6 isn't quite complete. We need some sort of visual indicator where the ray (bullet) hit. In this exercise, we will use little spheres that appear for just a second after an object in the Scene is hit. In order to do this, we'll need a **coroutine** (for an explanation on coroutines, see the *Preparation* section of this lab). A sphere will appear where the ray hit, several frames will go by that make up one second, and then we'll remove the sphere. Add the following statement to the code you did in Figure 6 just below the *'visually indicate where there was a hit'* comment:

```
StartCoroutine(SphereIndicator(hit.point));
```

`SphereIndicator` is the name of our method that will create and destroy the spheres. We have to wrap the function call inside `StartCoroutine()` since it will be a coroutine. We pass our `SphereIndicator()` method the *point* property of `hit` which is the 3D global coordinates of where the ray hit something (as a `Vector3`). That's where we want to create a sphere.

5. Next, you need to write the `SphereIndicator()` method:

```
private IEnumerator SphereIndicator(Vector3 hitPosition) {
    GameObject sphere =
        GameObject.CreatePrimitive (PrimitiveType.Sphere);
    sphere.transform.localScale = new Vector3 (0.5f, 0.5f, 0.5f);
    sphere.transform.position = hitPosition;

    yield return new WaitForSeconds (1);

    Destroy (sphere);
}
```

Figure 7: The `SphereIndicator()` Method

The code is pretty self-explanatory. The first statement programmatically creates a sphere object. Up to this point, you have been creating Game Objects using the Unity Visual Editor Menu bar (*GameObjects->3D Objects*). This statement is how you do the exact same thing but in code! The next statement scales the sphere to half size from default (which is too large). The next statement puts the sphere in the Scene where the ray hit.

The `yield` statement is where the magic of coroutines happen. That tells Unity to pause execution of the coroutine and go on to the next frame. In this case, with the

`WaitForSeconds(1)` argument, Unity won't continue with this coroutine until the number of frames that make up one second have executed. Afterwards, it will continue to the next statement which de-allocates the reference to the sphere object and removes it from the Scene.

6. Add the method shown in the figure above to your *RayShooter* script below the `Update()` method. Save your script and close **MonoDevelop**. Run your Scene. You may notice it's a little hard to aim. Let's fix that!

7. You are going to superimpose a 2D GUI (your HUD) over your 3D Scene where you will draw the aiming point (for an explanation of HUDs see the *Preparation* section of this lab). First, declare and initialize a public global variable:

```
public int aimSize = 16;
```

   By making this variable global, you can modify it right in the Inspector Tab of the Unity Visual Editor whenever you select the Camera component. This variable defines how large the aiming spot will be.

8. Next, add the following two statements to your `Start()` method in *RayShooter*:

```
// hide the mouse cursor
Cursor.lockState = CursorLockMode.Locked;
Cursor.visible = false;
```

   This locks the mouse to the centre of the screen and turns it off. The mouse cursor inputs affect the X and Y position of the view but it doesn't mean that the cursor will always be in the centre of it. Thus, the first statement above locks the mouse cursor to the centre of the view. Since you are adding an aiming spot and you don't want the mouse cursor superimposing on it, the second statement turns it off. **When running the Scene, you can hit ESC to unlock the mouse cursor.**

9. Finally, add the `OnGUI()` method:

```
void OnGUI() {
   GUIStyle style = new GUIStyle();
   style.fontSize = aimSize;

   // find the centre of the camera view and adjust for asterisk
   float posX = _camera.pixelWidth / 2 - aimSize / 4;
   float posY = _camera.pixelHeight / 2 - aimSize / 2;

   GUI.Label (new Rect (posX, posY, aimSize, aimSize), "*", style);
}
```

Figure 8: The `OnGUI()` Method

The first two statements set the size of the Font. This will set the size of the aiming spot since we are using a simple textual asterisk (*) to mark it. The next two statements set the X and Y origin for drawing the Label (a Rectangle) that will contain the asterisk (the aiming spot). Rectangles are drawn as follows in Unity:

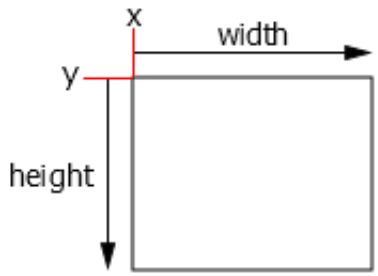

Figure 9: A Unity Rectangle

The origin marks the top left corner. We divide the camera pixel height and width in two to find the centre of the camera view. But we need to subtract from this value slightly to make room for the asterisk to be drawn and to ensure it is as close to the centre of the camera view as possible. That's what the `aimSize/4` for the X axis and `aimSize/2` for the Y axis are doing. The reason it's divided by 4 for the X axis is because the font is typically twice as tall as its width.

The last statement creates the label and draws it on the screen.

10. Save your Script and close **MonoDevelop**. Run your Scene and start firing!! Is it a lot easier to aim now??

Tasks Part 4 – Adding a Target (an Enemy):

1. It's great you can shoot but it starts getting old shooting at just the walls! You need a target and ideally, a target that does something when hit. Since this is going to be a target you can see when you run the Scene, let's use something a bit fancy. We'll use a pre-made model that is free to use. In the Unity Visual Editor, click on the ***Asset Store*** Tab. If you don't see it, hit *CTRL+9* to open it.

2. Activate the Filters control and hit the *Free Only* button. Look for a 3D model you can use as a target. For this example, we'll use Space Robot Kyle (but you can pick your own)!! Click on your chosen 3D model.

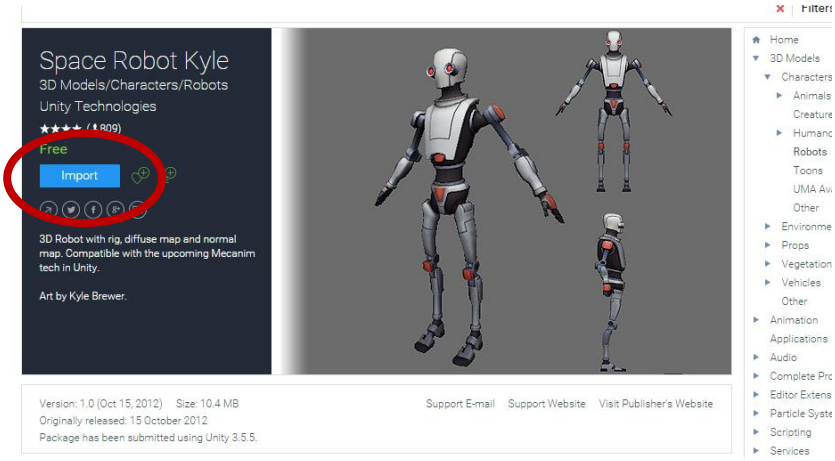

Figure 10: Robot Model from the Asset Store

3. Hit the *Download* and then the *Import* button (circled in the figure above) for the Asset you chose. Accept the Terms of Service (if needed) and afterwards, this should start the download (if needed). Once finished, the Import Unity Package Window will open. Make sure everything is checked (you can hit the All button) and then press the *Import* button. This will add the model to your Project. You should see a new folder in your Assets window.

4. To place the model in your Scene, click on the Robot Kyle folder in the Assets window. Then click on the Model folder. Drag Robot Kyle from the Assets window into the Scene Tab to place it in your Scene. Rename it to Enemy in the Hierarchy Tab.

5. Have it both facing the Player and at some distance from the Player. A good position would be 0, 0, 5 with a rotation of Y = 180.

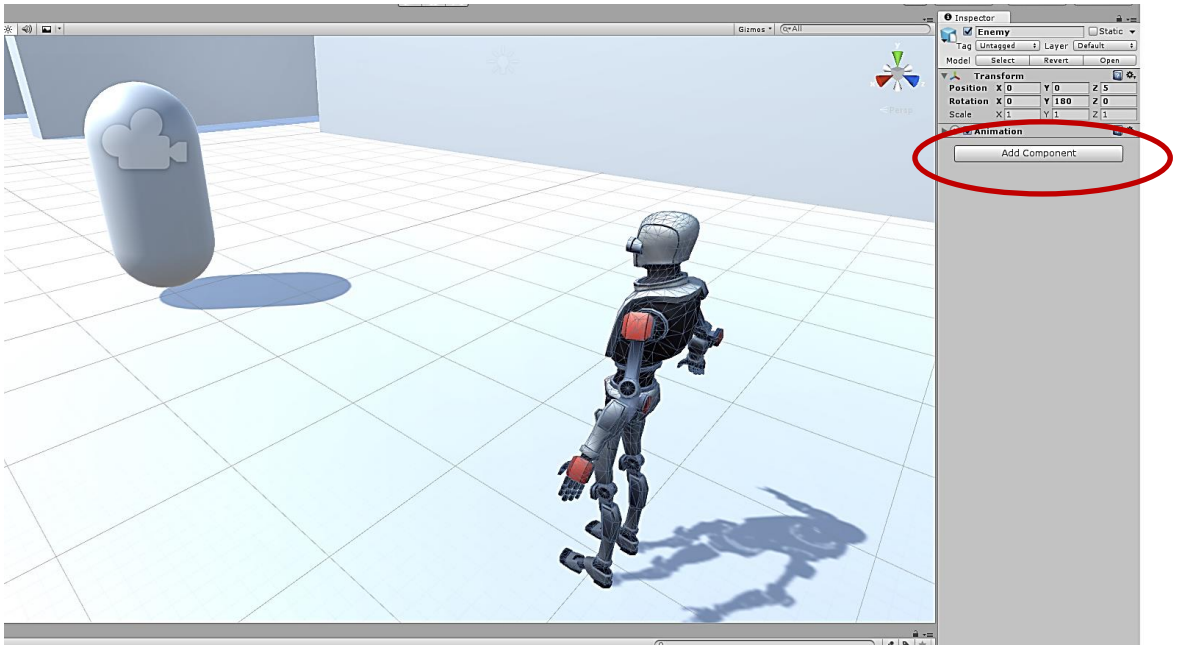

Figure 11: Kyle placed in the Scene

6. We need to add a Collider Component to our Enemy so that your Player won't walk through it. Click on *Add Component* (circled in the Figure above) then select Physics->Mesh Collider. The *Mesh Collider* is a more advanced collider that overlays the mesh of the object (recall that meshes are the polygons that make up a 3D object). To tie together the Mesh Collider with the Mesh of the Enemy object, hit the small circle beside the rectangle with the None (Mesh) text (circled in the figure below).

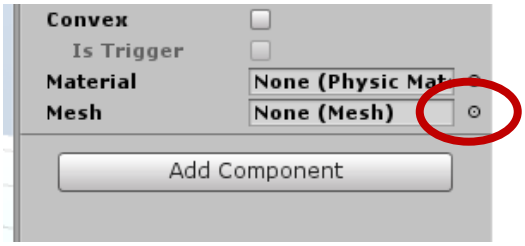

Figure 12: Mesh Collider Mesh Chooser

7. A Select Mesh window will pop up. Choose Robot2, the mesh for Robot Kyle. Close this window. You should see a green outline over the Enemy in the Scene Tab. That's the collider (don't worry, it doesn't get rendered when you play the Scene).

8. For now, this Enemy is going to be really easy. It won't move and requires a single shot to kill. We do, however, want it to react when shot. We'll have it fall over and then disappear. This will all be done in a new Script called *ReactiveTarget*. Create this script now and attach it to your Enemy by dragging it from the Assets window in the Project Tab onto the Enemy listed in the Hierarchy Tab.

9. We first need to make some changes in our *RayShooter* script. Double click on it to open it in **MonoDevelop**. Modify the `if (Physics.Raycast (ray, out hit))` block to match that below:

```
if (Physics.Raycast (ray, out hit)) {
    GameObject hitObject = hit.transform.gameObject;
    ReactiveTarget target =
        hitObject.GetComponent<ReactiveTarget> ();
    // is this object our Enemy?
    if (target != null) {
        target.ReactToHit ();
    } else {
        // visually indicate where there was a hit
        StartCoroutine (SphereIndicator (hit.point));
    }
}
```

Figure 13: Modifed *RayShooter* if block

The first statement (where `hitObject` is assigned) gets the reference of which Game Object was hit. The next statement uses `GetComponent()` to get the `ReactiveTarget` Script component attached to this Game Object and assigns it to the variable `target`. The if statement checks to see if `target` is null. If not, then it means a Game Object with a ReactiveTarget component was hit. That can only be the Enemy! We

didn't attach the Script to the walls, floor, etc. So if our Enemy was hit, call the method `ReactToHit()`. If not, place a sphere as before.

10. The next step is to implement the `ReactToHit()` method. Since this is being called on `target`, which is an instance of our *ReactiveTarget.cs* Script, we need to implement it there. Double click on it in the Assets window to open it in **MonoDevelop**.

11. You can delete both the `Start()` and `Update()` methods (or just leave them blank) in this script. We want our enemy to fall when shot and then disappear after several seconds. Since this will need to happen over several frames, we'll need to use a ***coroutine***. Thus, the `ReactToHit()` method will call a new method Die that will be wrapped in a call of `StartCoroutine()` (see *Preparation* section as to why we need to do this):

```
public void ReactToHit() {
    StartCoroutine (Die ());
}

private IEnumerator Die() {
    // Enemy falls over and disappears after two seconds
}
```

Figure 14: `ReactToHit()` and the incomplete `Die()` methods in *ReactiveTarget*

12. Ideally, we want the Enemy to fall over several frames instead of instantly in a single frame (which would look funny). Typically, this is done by *Animation*. There are several different ways to animate and for this lab, we'll use one of the simplest. It's called a *tween*. Tweens are systems used to animate objects over time. We'll use a free one that we can add to our project from Unity's Asset Store. Activate the Asset Store Tab (or if it closed, hit CTRL+9) and search for a free Tween called ***iTween***. Download and Import it just like you did for Robot Kyle. If you did it correctly, you should see a new *Plugins* folder in your Assets window.

13. Once iTween has been imported into your project, you can use it right away. The documentation for it is located at http://itween.pixelplacement.com/documentation.php. We'll use the `RotateAdd()` function to Rotate our Robot along the X axis by -75 degrees (so it will fall backwards) over 1 second. Add the following statement to your `Die()` method:

```
iTween.RotateAdd (this.gameObject, new Vector3 (-75, 0, 0), 1);
```

This is pretty self explanatory. It takes the Game Object this script is attached to (first argument) and rotates it by 75 degrees (second argument) over one second (third argument).

14. Next, we'll add a yield statement so that the Enemy will disappear after 2 seconds. We'll destroy it like we did for the sphere. The complete `Die()` method will look like the figure on the next page.

```
private IEnumerator Die() {
    // Enemy falls over and disappears after two seconds
    iTween.RotateAdd (this.gameObject, new Vector3 (-75, 0, 0), 1);

    yield return new WaitForSeconds (2);

    Destroy (this.gameObject);
}
```

Figure 15: The complete `Die()` Method

15. Save your Script and close **MonoDevelop**. Run your Scene and shoot the Enemy!

Submission:

Demo Your Lab to the Instructor:

- **Show that the player collides with the walls**
- **Show that the player can look around but stays on the ground (gravity)**
- **Show that the player can fire a weapon (spheres appear for 1 second to indicate a hit)**
- **Show that there is a simple HUD (an asterisk)**
- **Show that there is an Enemy 3D Model to fire at**
- **Show that the Enemy dies by slowly falling over 1 second**

- A successful demo scores you 10 marks.