CAMOSUN COLLEGE

COMPUTER SCIENCE DEPARTMENT

ICS 123 - GAMING AND GRAPHICS CONCEPTS

LAB 5 – AI AND PREFABS

DUE: DEMO BY END OF LAB WED FEB. 22
You must work individually for this lab.
Submission Instructions are on the last page of the lab.

Objective:

To have the students add AI to their enemy, re-spawn an enemy, and have the enemy shoot back!

Preparation:

Artificial Intelligence (AI):

In the last lab, you added your first Enemy! But the enemy wasn't very exciting; it didn't move (or shoot back). Having the enemy move and behave like a real human is what artificial intelligence is all about! AI in Computer Science is an enormous area of research. Whenever you play a game against a computer-generated opponent, AI is being used. In this Lab, you'll add some fundamental AI to your Enemy.

Spherecasting:

First, we want the Enemy to move, just like the Player (but instead, be controlled by your AI code). To avoid running into walls, your AI code will have to turn the Enemy in a new direction when a wall is detected in front of it. How will you do that??

In the last lab, you used *Raycasting* to shoot bullets from your Player and detect what and where it hit. We can use that same principle for the enemy but instead of it being a bullet, it will function more like a radar. The Enemy can fire a ray and determine if a wall is in front of it and how far away it is. Taking into account the width of the Enemy, the ray it emits has to be larger than the ray used for a bullet. Ray's are emitted from the centre of the Object but in some instances, this can be problematic. For example, the Enemy might be at an edge of a wall where its arm is against it but the ray emitted still doesn't detect a wall (the Enemy would get stuck!). Unity takes that into account and has a different method we can use called `SphereCast()`. *Spherecasting* is like *Raycasting* except it casts a sphere along the ray instead of a thin line. It is often called *thick raycasting*.

Tasks Part 2 will add Spherecasting to the Enemy.

Finite State Machines (FSM or SM):

Finite State Machines are a computational model based on the idea of state. For example, whether an Enemy is dead or not! A state diagram helps you design your state machine. The template for a state diagram is as follows:

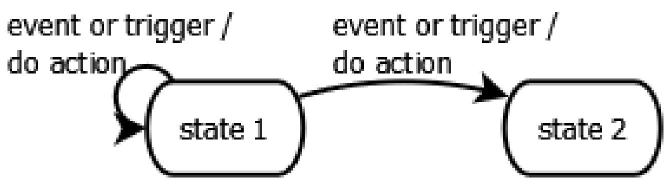


Figure 1: A simple state machine template

To code this is simple. In pseudo-code:

```
set state to state 1
Loop:
   if event or trigger for state 1, do state 1 action
   if event or trigger for state 2:
      change state to state 2
      do state 2 action
```

The first event/trigger on the left only happens if in state 1 and it doesn't cause a change in state. The second event/trigger does cause a change to state 2. This is a fairly simple example of a state machine. State machines can have many more states and state transitions. In any case, state machines are extremely useful tool when programming!

You will use a SM in Tasks Part 2 to keep track of whether an Enemy is dead or alive.

Unity Prefabs:

Unity prefabs are fully fleshed-out game objects. That is, a game object with all its components attached. It is very useful when you want to make several identical copies of a fleshed-out game object that you want to include in a Scene.

To create a Prefab is quite simple. You drag a fully fleshed-out game object (with all components attached) from the Hierarchy Tab and drop it into the Assets window of the Project Tab (you may realise that this is the reverse of attaching an asset to a game object). The Prefab will show up in blue in the Hierarchy Tab.

Prefabs can exist outside of any Scene but can only be edited within a Scene. This makes editing them a bit difficult.

You will use a Prefab to re-spawn the Enemy in Tasks Part 3.

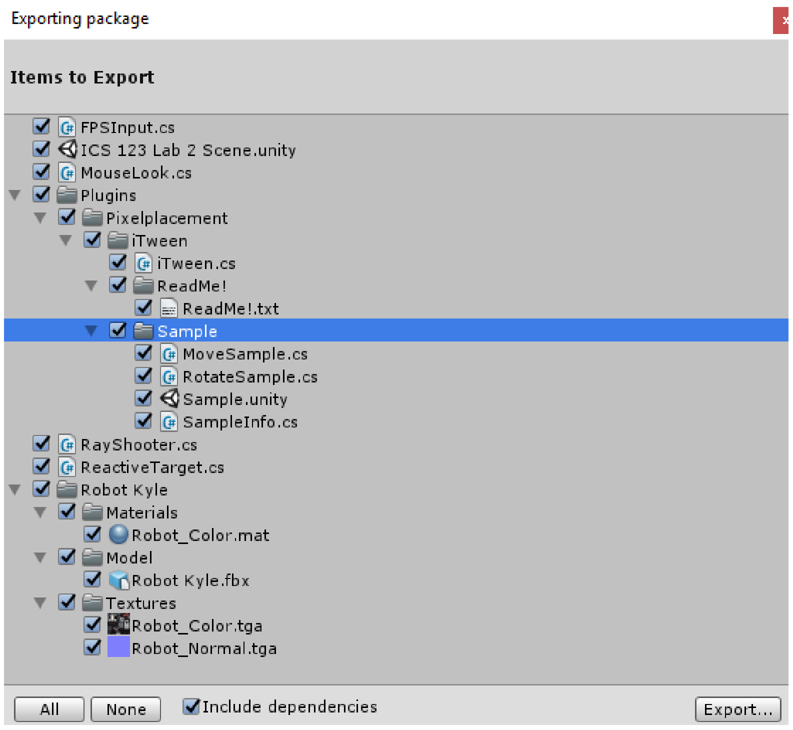
Unity API References for this Lab:

https://docs.unity3d.com/ScriptReference/Transform.html
https://docs.unity3d.com/ScriptReference/RaycastHit.html
https://docs.unity3d.com/ScriptReference/Physics.SphereCast.html
https://docs.unity3d.com/ScriptReference/Random.html
https://docs.unity3d.com/ScriptReference/Object.Instantiate.html
https://docs.unity3d.com/ScriptReference/Transform.TransformPoint.html
https://docs.unity3d.com/ScriptReference/Collider.OnTriggerEnter.html

Tasks:

Tasks Part 1 – Import Lab 4's Assets into a New Project

1. You will start a new Project for Lab 5 but you will want to import the Scene and all the other Assets you imported and created from Lab 4. First, open your Lab 4 project in Unity. Go up to the menu bar and select *Assets -> Export Package*. This should open the Exporting Package window:


Exporting package

Items to Export

```
☑ FPSInput.cs
☑ ICS 123 Lab 2 Scene.unity
☑ MouseLook.cs
▼ ☑ Plugins
  ▼ ☑ Pixelplacement
    ▼ ☑ iTween
        ☑ iTween.cs
      ▼ ☑ ReadMe!
          ☑ ReadMe!.txt
      ▼ ☑ Sample
          ☑ MoveSample.cs
          ☑ RotateSample.cs
          ☑ Sample.unity
          ☑ SampleInfo.cs
☑ RayShooter.cs
☑ ReactiveTarget.cs
▼ ☑ Robot Kyle
  ▼ ☑ Materials
      ☑ Robot_Color.mat
  ▼ ☑ Model
      ☑ Robot Kyle.fbx
  ▼ ☑ Textures
      ☑ Robot_Color.tga
      ☑ Robot_Normal.tga
```

Figure 2: 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 5 folder (create one if you haven't yet!). Save this new package as 'ICS123Lab4Assets'. 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 5'. Hit the 'Create project' button. This will close your current Lab 4 Project and relaunch Unity with a new empty Lab 5 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 – AI Part 1: Making the Enemy Move

A) Spherecasting:

1. Create a new C# Script and call it *WanderingAI*. Attach it to the Enemy by dragging it from the Assets window to the Enemy in the Hierarchy Tab (you've done this before!). Open the Script in **MonoDevelop**.

2. At the top of the Script, add two global variables:

```
public float enemySpeed = 3.0f;
public float obstacleRange = 5.0f;
```

   These are both pretty self explanatory. The variable `enemySpeed` controls how fast the Enemy will move and `obstacleRange` means react to obstacles by turning away within that distance.

3. Next, modify the `Update()` method in this Script to match the figure below:

```
void Update () {

    // Move Enemy and generate Ray
    transform.Translate (0, 0, enemySpeed * Time.deltaTime);
    Ray ray = new Ray (transform.position, transform.forward);

    // Spherecast and determine if Enemy needs to turn
    RaycastHit hit;
    if (Physics.SphereCast (ray, 0.75f, out hit)) {
        if (hit.distance < obstacleRange) {
            float turnAngle = Random.Range (-110, 110);
            transform.Rotate (0, turnAngle, 0);
        }
    }
}
```

Figure 3: WanderingAI `Update()` Method

The first statement moves the Enemy forward along the Z axis (into the Scene). Just like the Player, we use `Time.deltaTime` to make its movement code frame-rate independent. Next, we cast a ray with the origin at wherever the Enemy is now and cast it in a direction forward of the enemy.

The second block of code does the Spherecasting. First, we declare a new `RaycastHit` struct to store information about a hit (just like we did for the Player firing). Then a sphere is cast along the ray. The sphere's radius is 0.75f (<u>change this to half the width of your model if you used something besides Robot Kyle</u>). We use the `out` modifier to use *pass-by-reference* for the hit structure. The whole statement is in an `if` test to see if anything was hit by the sphere. If it did hit something, see how far away it is (stored in the `distance` variable of the `hit` structure). If it's within our 'react' distance, choose a random angle to turn between -110 and 110. Then rotate the Enemy along the Y axis by that angle. Since all of this is done in the `Update()` method, it will be done each frame.

4. Save your Script and close **MonoDevelop**. Now play your Scene. The Enemy should move and turn when necessary. It's artificial intelligence at work!!

5. You'll probably notice a couple of issues when running your Scene. First, the Enemy slides across the floor without moving its legs. This would require *animation* (and more than just a tween). We'll leave it for now and explore more complex animation later in the course. The second problem is after you shoot the Enemy, it floats up and dissappears!! Maybe it's going to a better place but a more technical reason is that your AI Script doesn't know it's dead and just keeps on moving in the direction the Enemy is facing (up in this case).

B) Tracking the Enemy's State:

6. We'll need to utilize a SM (see *Preparation* section of this Lab) to keep tract of whether the Enemy is alive or dead. Our state machine will look like:

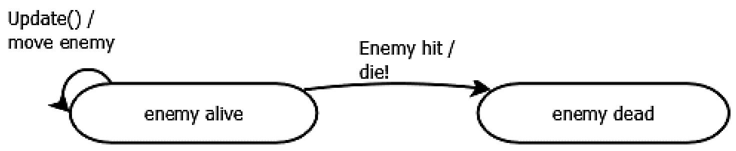


Figure 4: Enemy State Diagram

As long as the enemy is in the first state, *enemy alive*, on every call to `Update()`, move the enemy as normal. When the enemy is hit, make it die, and move into the second state, *enemy dead*. Since the *move enemy* action only happens in the *enemy alive* state, the enemy can't move in the *enemy dead* state. This should fix the problem from Step 5.

7. Implement the State Diagram from the Figure above. Start by declaring an enumeration. An enumeration allows us to define a new custom variable type. Since it will be used in multiple classes and isn't a variable declaration (just a type declaration) it is safe to declare it outside a class definition. Put the following statement below any `using` statements but before the class declaration in *WanderingAI*:

```
public enum EnemyStates { alive, dead };
```

8. Now declare a variable to store the state using the above type. Make it private and give it class-scope:

```
private EnemyStates state;
```

Since the type of this variable is declared as `EnemyStates`, it can only have the value `alive` or `dead`.

9. In the `Start()` method, set `state` to `EnemyStates.alive`. When the Scene starts, the Enemy will of course be alive.

10. In the `Update()` method, you want to nest everything already there in an `if` statement that checks if the Enemy is alive. If it's dead, you won't do anything (no moving or spherecasting needed). Do that now!

11. Finally, we need a method that we can call to change the state. Add the following method to your *WanderingAI* Script:

```
public void ChangeState(EnemyStates state) {
    this.state = state;
}
```

When called, this will set the member variable `state` to whatever the parameter `state` is.

So when does the method in the above step get called? When the enemy gets hit of course! Save and close the *WanderingAI* Script. Open up the *ReactiveTarget* Script. We'll need to make some changes to `ReactToHit()`. We first need to get a reference to the *WanderingAI* Component. Here, we'll use our trusty `GetComponent()` generic method:

```
WanderingAI enemyAI = GetComponent<WanderingAI> ();
```

12. Next we'll ensure that `enemyAI` isn't `null`, i.e. that there is indeed a *WanderingAI* component. There's absolutely no reason why there shouldn't be but a null check is good practice to prevent a program crash (any operation on a null reference would cause a crash). If it isn't null, we'll change state to dead.

```
if (enemyAI != null) {
   enemyAI.ChangeState (EnemyStates.dead);
}
```

No comments needed here. The code is very self explanatory! Your final `ReactToHit()` method should look like the figure below.

```
public void ReactToHit() {

    WanderingAI enemyAI = GetComponent<WanderingAI> ();
    if (enemyAI != null) {
        enemyAI.ChangeState (EnemyStates.dead);
    }

        StartCoroutine (Die ());
}
```

Figure 5: The final `ReactToHit()` method

13. Save your Script and close **MonoDevelop**. Run your Scene again. This time, when the Enemy is shot, it shouldn't float away!

Tasks Part 3 – Re-Spawning the Enemy Using a Prefab

1. Once the enemy is shot and dies, there's nothing left for the Player to do! Let's fix that by causing the enemy to re-spawn. We'll first need to create a Prefab of the Enemy. Our enemy is actually already a prefab but not a *full-fleshed prefab*. It is a *model prefab*. What we want to do is make it a full-fleshed one with both the Collider and the Script Components we created attached to it. To do so, drag the Enemy object from the Hierarchy Tab into the Assets window of the Project Tab. There, you just created a Prefab!!

2. Instead of having the Enemy placed in the Scene before you run it, you are now going to spawn in programmatically. Delete the Enemy in the Scene by right-clicking on it in the Hierarchy Tab and selecting Delete. **Do not** delete the one in the Assets window, that's the Prefab!!

3. Here's where things get a little weird. You want to spawn your Prefab in the Scene but there's no object to attach your spawning code to. Luckily, Unity allows you to create an empty game object, an object that doesn't render anything in the Scene. Go up to the menu and Choose *GameObject->Create Empty* and rename it to *Controller*. Its Transform position isn't really important but you might like to set its position to 0, 0, 0.

4. Create a new C# Script and call it *SceneController*. Open it in **MonoDevelop** and modify it to look like the figure below.

```
   public class SceneController : MonoBehaviour {
1      [SerializeField] private GameObject enemyPrefab;
2      private GameObject enemy;

       // Update is called once per frame
       void Update () {
3          if (enemy == null) {
4              enemy = Instantiate (enemyPrefab) as GameObject;
5              enemy.transform.position = new Vector3 (0, 0, 5);
6              float angle = Random.Range (0, 360);
7              enemy.transform.Rotate (0, angle, 0);
           }
       }
   }
```

Figure 6: *SceneController* `Update()` Method

Explanation of Code:

Line 1 declares a private variable that will hold a reference to the prefab we created. The `[SerializeField]` attribute in this statement serializes the object. Serialization converts an in-memory object into a serial stream of bytes. Any variable that is serialized by the `SerializeField` attribute gets displayed in Unity's visual editor. You have already seen serialized variables whenever you have declared a public class-scope variable (for example `enemySpeed` and `obstacleRange` in *WanderingAI*). Public variables are serialized automatically by Unity which is why you didn't need the `SerializeField` attribute. In this case, the variable is declared private but by adding the `SerializeField` attribute, it will show up in Unity's Inspector Tab. We need to access this variable in the Inspector to make the link with the Prefab but we don't want other classes changing it. That's why it was declared as private.

Line 2 is a reference to the Enemy game object that will be rendered in the Scene.

Line 3 checks to see whether `enemy` is referencing an object. When you first play your Scene this will be null because you deleted the existing Enemy in Step 2 and there isn't one in the Scene yet! Another time this variable will be null is after the Enemy dies. In either of these cases, the code in the `if` block will be executed otherwise nothing happens (there's already an Enemy in the Scene).

Line 4 is where the magic happens. We use Unity's `Instantiate()` method to clone our Prefab and create an object from it. `Instantiate()` returns it as type Object, i.e. it does an *upcast*. But we need to *downcast* it again to a Game Object (its original type). That's what the `as GameObject` does. This is called casting. There's no problem doing this because `GameObject` is a child of `Object`.

Lines 5, 6, and 7 are fairly self explanatory. Line 5 places the Enemy in the Scene, line 6 chooses a random angle, and line 7 rotates the Enemy to that angle. Thus, every time a new Enemy is placed in the Scene, it will be facing some random direction! (Note: it's possible that the Enemy will end up spawning in a wall, depending on how you designed your Scene. If that's the case, change the Z position).

5. Save your Script and close **MonoDevelop**. In the Unity Visual Editor, attach this Script to the empty *Controller* object you created in Step 3 by dragging it onto *Controller* in the Hierarchy Tab.

6. Next, you need to link your Enemy Prefab to the `enemyPrefab` variable you created in *SceneController*. Drag the Enemy Prefab from the Assets window over the box beside the *Enemy Prefab label* where it says *None (Game Object)* in the Scene Controller Component of the Controller Game Object in the Inspector Tab. The box should turn blue. Once you let go of the left mouse button, you should see the text in the box change to Enemy.

7. Now play the Scene. After you shoot the Enemy dead, a new one should spawn!!

Tasks Part 4 - AI Part 2: The Enemy Strikes Back!

1. Our first-person shooter seems a bit skewed in the Player's favour. The Enemy can't shoot back! Let's change that. Recall that we had the Player shooting by using *raycasting*. This time, we'll use prefabs to accomplish something similar. The big difference with using a Prefab is that now the projectile is a Game Object. That means it can be rendered in the Scene (instead of being invisible like the ray) and instead of *raycasting*, we'll use collision detection. Our robot Enemy will fire lasers! The first thing we need to do is create a model that will become the Prefab. We'll start with a sphere. Go up to the menu and select *GameObject->3D Object->Sphere*. Rename this to *LaserBeam*.

2. You want to turn the sphere into something that looks like a laser beam. To do so, use the *Scale Tool* or enter values directly in the Inspector Tab. For example, something like 0.2,

0.15, 1 but you can use your own values (careful with the Z value though, otherwise the beam will stick into your model and not work). The position and rotation are irrelevant as those will be determined when it gets placed in the Scene programmatically.

3. We want to change the colour of this laser. Perhaps an orange or deep red colour. To do so, you need to create a material. A 3D object's material controls what its surface looks like. Go up to the menu and choose *Assets->Create->Material*. Name the material *LaserColour*.

4. Select *LaserColour* in the Asset window of the Project Tab and look for *Albedo* under Main Maps in the Inspector Tab. Albedo is the surface colour. Click on the coloured square to open up the color picker. Use the color picker to choose the color for the laser beam.

5. In the Inspector Tab, beside where it says Emission, change the value in the box from 0 to 0.5 (or some other value, you pick!). Emission means that the material itself is a light source. It adds brightness to it.

6. Attach the *LaserColour* material to the *LaserBeam* by dragging it from the Assets window on to the *LaserBeam* in the Hierarchy Tab. You should see the surface of the *LaserBeam* in the Scene take on the colour of the material you just created.

7. Our Prefab is almost ready. Next, we need to attach a Script to it. Create a new C# Script and name it *LaserShooter*. Attach it to the *LaserBeam* Game Object. We can edit it later but we need to attach it now so that we can make the Prefab.

8. Now, make the Prefab, just like you did in the last task. Once you have the Prefab made, you can delete the *LaserBeam* Object in the Scene.

9. Now comes the coding. Create a new C# Script and call it *PlayerCharacter*. Attach the Script to the Player Game Object. Our Player, unlike the Enemy, won't die on a single shot. Instead, the Player can take 5 hits. We'll keep track of this in a variable called `health` inside this Script.

```
public class PlayerCharacter : MonoBehaviour {
    private int health;

    // Use this for initialization
    void Start () {
        health = 5;
    }

    public void Hit() {
        health -= 1;
        Debug.Log ("Health: " + health);
        if (health == 0) {
            Debug.Break();
        }
    }
}
```

Figure 7: The PlayerCharacter Script

For now, we'll print debug messages to the console to indicate the Player's current health.

Later, we'll add this to the Game's HUD. Once health reaches 0, we'll pause the Scene, for now. Obviously, in a full-fledged game, you'd want to add some option to re-start or quit the game.

10. Next, we'll need to make some changes to the *WanderingAI* Script. We only want the Enemy to shoot when it is facing the Player so we need to add code to detect when this happens. Change the `if` block in *WanderingAI* to look like the following figure:

```
if (Physics.SphereCast (ray, 0.75f, out hit)) {
    GameObject hitObject = hit.transform.gameObject;
    if (hitObject.GetComponent<PlayerCharacter> ()) {
        // Spherecast hit Player, fire laser!

    } else if (hit.distance < obstacleRange) {
       // Must've hit wall or other object, turn
       float turnAngle = Random.Range (-110, 110);
       transform.Rotate (0, turnAngle, 0);
    }
}
```

Figure 8: Updated WanderingAI Script (Incomplete)

If the new added code looks familiar, it's because it is very similar to code in the *RayShooter* Script for the Player to determine what the Player hit.

The Enemy issues a *Spherecast* and we store a reference to the object it hit in `hitObject`. We then use the handy `GetComponent()` method to see if it was the Player. If so, the Enemy won't turn but fire its laser! If it was anything else that it hit, it will turn as before.

11. We need variables to hold references to the *LaserBeam* Prefab and a *LaserBeam* Game Object that will be rendered in the Scene. Like in *SceneController*, we'll need to make use of the `SerializeField` attribute. Below where you declared `state`, add the following two declarations:

```
[SerializeField] private GameObject laserbeamPrefab;
private GameObject laserbeam;
```

12. Next, we need to declare two variables that will control the rate-of-fire of the Enemy. Add the following two variable declarations right below the ones from the last step:

```
public float fireRate = 2.0f;
private float nextFire = 0.0f;
```

`nextFire` will act like a timer. `fireRate` will, of course, control just how quickly the Enemy can keep firing.

13. The next thing to do is fill out the code in the `if` block when the *Spherecast* has hit the

Player. This is going to be very similar to the code in *SceneController*. In that Script, we instantiated the Enemy into the Scene. This time, we'll be instantiating our *LaserBeam* Prefab! Add the following code below the *'Spherecast hit Player, fire laser!'* comment from figure 8 (**DON'T FORGET TO INCLUDE THE CLOSING CURLY BRACE ON LINE 6**):

```
1   if (laserbeam == null && Time.time > nextFire) {
2       nextFire = Time.time + fireRate;
3       laserbeam = Instantiate (laserbeamPrefab) as GameObject;
4       laserbeam.transform.position =
        transform.TransformPoint (0, 1.5f, 1.5f );
5       laserbeam.transform.rotation = transform.rotation;
6   }
```

Figure 9: The code for firing the laser beam

Explanation of Code:

Line 1 checks two things. First, that there isn't already a laser beam in the Scene from this specific Enemy. Then, it checks if enough time has passed since the last laser beam was fired. For example, if `fireRate` is set to `2.0f`, then the Enemy can't fire again for about two seconds.

Line 2 `Time.time` is a static variable that stores the time in seconds since the start of the game (to a total of seven significant digits). This statement will get the current timestamp, add it to `fireRate`, and then store it in `nextFire`.

Line 3 instantiates the Prefab. We aren't using `new` here because it is actually cloning an already existing object (the Prefab). `Instantiate` does an upcast to `Object`; we have to downcast back to `GameObject`. That is what the '`as GameObject`' is doing.

Line 4 places the laser beam into the Scene. The numbers used here for Y and Z are based on the Robot Kyle model so that the laser emanates from its chest. If you used a different model, follow the below steps:

1. Place your Enemy Prefab in the Scene at position 0, 0, 0 (you may need to move the Player out of the way).
2. Place your Laser Beam Prefab in the Scene. Position it where you want it to emanate from the model. Ensure that no part of the beam is actually touching the model. Note the X, Y, and Z values and use those numbers instead of the ones shown in the figure above. You can delete both the Prefabs from the Scene.
3. If you Scaled your model, for example, changed the X, Y, and Z scale to 10, you need to divide your X, Y, and Z position values by 10 (the same value that you used for Scale). This is because `TransformPoint`, which converts a local point to a global one, is affected by the Scale.

Line 5 sets the rotation of the beam to match that of the Enemy so that the beam is always emanating from the front of the model.

14. Now click on the Enemy Prefab in the Assets window. You'll see a new box with the *Laserbeam* Prefab label in the Wandering AI Component. You need to attach your *LaserBeam* Prefab to this Component just like you did with the *Enemy Prefab* and the

SceneController Component of the *Controller Game Object*. Simply drag *LaserBeam* from the Assets window and drop it on the *Laserbeam Prefab* box in the Wandering AI Component of the *Enemy* Prefab.

15. Now we need to flesh out the *LaserShooter* Script that you already attached to the Prefab. Open it **MonoDevelop** and modify it to look like the figure below.

```
public class LaserShooter : MonoBehaviour {
    public float speed = 6;

    // Update is called once per frame
    void Update () {
        transform.Translate (0, 0, speed * Time.deltaTime);

    }

    void OnTriggerEnter(Collider other) {
        PlayerCharacter player = other.GetComponent<PlayerCharacter> ();
        if (player != null) {
            player.Hit ();
        }
        Destroy (this.gameObject);
    }
}
```

Figure 10: LaserShooter Script

In `Update()`, we move the Laser forward along the Z axis toward the Player according to the speed factor. `Time.deltaTime` is there to make the code *frame-rate independent*.

`OnTriggerEnter()` is called when a certain event occurs (called a *trigger event*). Specifically, it will be called automatically when the Collider component of the *LaserBeam* (Sphere Collider) collides with another object's Collider. If it's the *PlayerCharacter*, then we call the `Hit()` method to damage the Player. Whether the LaserBeam hit the player or not, the LaserBeam is removed from the Scene.

16. Next, add a *RigidBody* Component to the *LaserBeam*. *RigidBody* ensures that Collisions are properly detected. It is important to de-select *Use Gravity*. Otherwise, it will get pulled to the ground.

17. Finally, with LaserBeam still selected, check *Is Trigger* in the Sphere Collider. That ensure the `OnTriggerEnter()` method is called when a collision occurs.

18. Now click on the Console Tab to activate it so you can monitor the Player's health. Then, play the Scene!

19. One Enemy just isn't enough. It's not challenging enough! Cause multiple Enemies to spawn at the start of the Scene (around 3 to 5) and re-spawn when shot.

    Hints:

    - You only need to make changes to the *SceneController* script.

- Declare a private `int` variable to hold the number of Enemies.
- Use an array to store the Enemy Prefabs. Declare an array of GameObjects. Make its visibility private. This array will store references to each enemy prefab in the Scene.
- Allocate (instantiate) the array in the `Start()` method. Use the number of enemies variable you declared to set the size of the array.
- Use a loop structure in the `Update()` method to iterate through the array. If the current location in the array is null, an Enemy has to be spawned! Instantiate, position, and rotate an Enemy Prefab. Then, store it in the array.

Submission:

Demo to the Lab Instructor:

- **Show that your Enemy moves and avoids obstacles by turning.**
- **Show that when the Enemy dies, it doesn't float away.**
- **Show that when the Enemy dies, a new one spawns.**
- **Show that the Enemy can fire 'lasers' back and that in the Console, the Player's Health gets reduced (from 5).**
- **Show that multiple Enemies can be spawned.**

- A successful demo scores you 10 marks.