Raycasting Lab

The Art of LASERS

Introduction

This lab expects you to understand physics layers and colliders. With that said, let's ask the question: what is raycasting? In Unity, when someone talks about raycasting, they are referring to a form of collision detection that uses rays or one of the other potential shapes. Unlike colliders, raycasting is performed in a script by calling a Physics or Physics2D function. With the introduction out of the way, this lab will be covering:

- 2D vs 3D raycasting
- How to raycast
- The difference between casting and overlapping
- The difference between all and not all
- The various types of shapes that can be cast in 2D

2D vs 3D Raycasting

In order to perform raycasting in 2D, you must use the Physics2D class while performing raycasting in 3D requires you to use the Physics class. The main differences between 2D and 3D is that you are expected to be using the 2D version in a 2D game and the 3D version in a 3D game. Below are a portion of the signatures:

```
public static RaycastHit2D Raycast(Vector2 origin, Vector2 direction, ...)
public static bool Raycast(Vector3 origin, Vector3 direction, ...)
```

As we can see, the 2D version uses Vector2's while the 3D version uses Vector3's. Beyond the different types, using these two is nearly the same. You choose a starting position for your raycast, a direction for it to move in, a distance, and that is about it. One key difference of note is the retrieval of information from a collision. In 2D, the Raycast returns the information while in 3D it returns whether there was a collision. In order to retrieve the collision information from a Raycast in 3D, you must use do something like the following:

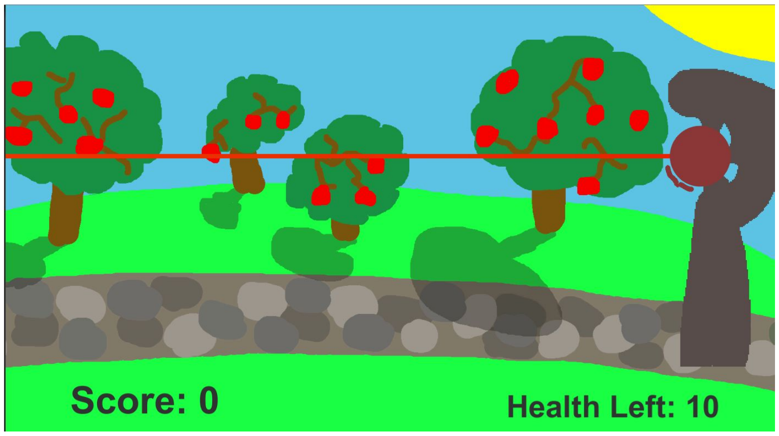
```
RaycastHit hit;
Physics.Raycast(start, dir, out hit);
```

With that, we will move on to actually using raycasting. For the rest of the lab we will only talk about raycasting in 2D but everything you learn is applicable to 3D.

How to Raycast

If raycasting still confuses you, it will hopefully make sense by the end of this section.

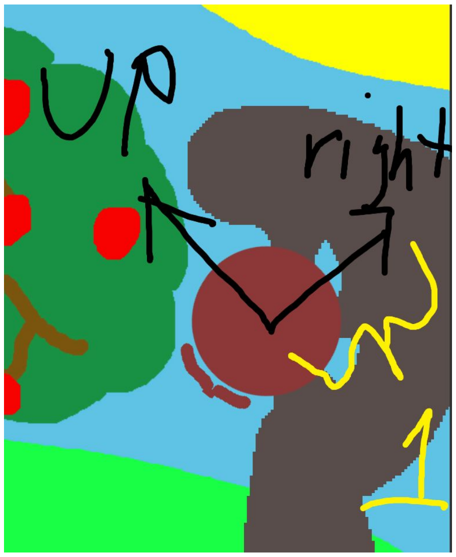
Go ahead and open up the lab's corresponding Unity project if you have not done so already. Once open, open up the Attack.cs script. In this script, under the "Attack Methods" region, you will find the methods "ShootLaser()" and "ShootMegaLaser()". These are the only methods that we will be editing throughout the lab. While you are more than welcome to look at the rest of the code, this document will not explain any of it.



Start running the game. Press the "Start" button. Now if you press Spacebar, you will see a narrow red beam fire to the left and if you press Left Shift, you will see a wide blue beam fire to the left. The red beam has a cooldown of 0.2 seconds while the blue beam has a cooldown of 3 seconds. Now if you hold W, S, the up arrow, or the down arrow, you will see the pointer moving. Unfortunately when you shoot, the laser is not firing in the correct angle!

The laser graphics actually have nothing to do with raycasting but they will help us with ensuring that we are casting in the correct direction. Remember how a raycast needs an origin and a direction? Let's go ahead and prepare those! We know we want our laser to start from approximately the center of the circle. Since the Attack script is on the circle, we can simply grab its "transform.position":

```
Vector2 start = transform.position;
```



Now we want the correct direction. Thankfully, Unity has made our lives easy by implementing a "transform.right" which will provide us the unit length "right" relative to the orientation of our game object. The picture to the right displays the vectors returned by "transform.up" and "transform.right". Again, they both have a magnitude of one. Currently we are using "Vector2.right" which will always return the vector (1, 0), regardless of the orientation of our game object.

We cannot use “transform.right” as it is because as the picture depicts, it will point in the incorrect direction and unfortunately, there is no “transform.left” so all we have to do is negate our vector (or multiply by -1) because that represents the opposite.

```
Vector2 dir = -transform.right;
```

And when we substitute “start” and “dir” appropriately, our laser will now fire in the correct direction! You can repeat this for the mega laser now, but I recommend waiting since there is a lot of code that you will be able to copy paste all at once.

One minor addition I recommend is:

```
start += dir;
```

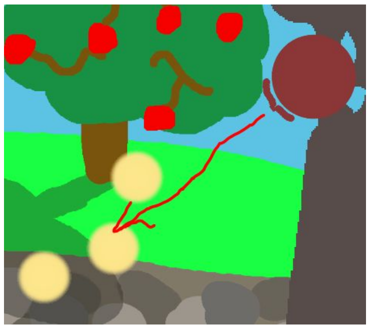
The difference is that now the laser originates at the peak of the arrow rather than the center of the circle.

Now that we have the visuals working, it is time to actually raycast! Go ahead and add this line:

```
RaycastHit2D hit = Physics2D.Raycast(start, dir, p_LaserRange);
```

That is it! Now we are raycasting. What does this actually mean? Essentially, “Physics2D.Raycast(...)” shoots an invisible laser out and sees what collides with it.

In this case, if we shoot the laser then it will collide with that yellow enemy. Once it collides with something, it will stop going forward and return the information about the collision. If we take a look at the image on the right, we can see that shooting a laser in that direction has two enemies in front. However as explained, the raycast will completely stop once that first enemy is hit. If there are no enemies in the raycast’s path anywhere between its origin and “p_LaserRange”, it will return that there was no collision.



With that said, we now need to figure out whether or not a collision occurred. That information, along with a lot of other useful information is all stored in the “RaycastHit2D” hit that we created. To check if there was a collision, we check for a collider! And if there was no collider then there was no collision so we can just draw our fully extended laser like before.

```
if (hit.collider == null)
      DrawLaser(m_LaserPrefab, start, dir, p_LaserRange);
```

However, if there was a collision, let’s first make sure the laser being drawn reflects the ray that we cast.

```
else
{
      DrawLaser(m_LaserPrefab, start, hit.point);
}
```

Quick note, place the rest of the code in the else clause until told otherwise.

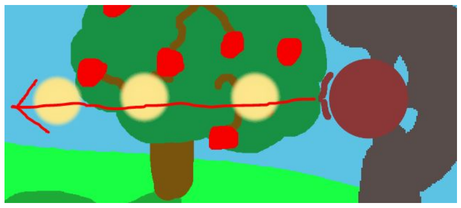
In the above statement, we see “hit.point”. This describes the exact point that the ray hit. We know that we hit something so we want to check if it was an enemy and if so, give us a point and make it explode. The following code does just that.

```
Enemy enemy = hit.collider.GetComponent<Enemy>();
if (enemy != null)
{
      enemy.Hit();
      AddScore();
}
```

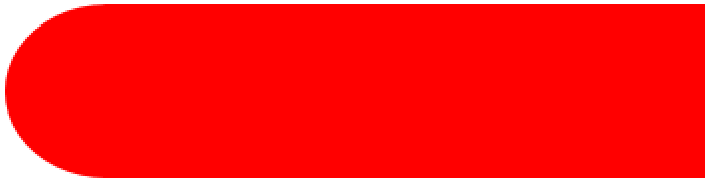
That’s it! If you now add an “EnemySpawner” prefab into the scene and press play, you will be able to fire the normal laser and kill enemies to gain points.

Minor Differences and a Mega Laser

There are three options available to you when choosing how to raycast. You have the normal “Raycast” that we just learned about, but you also have “RaycastAll” and “RaycastNonAlloc”. When using “RaycastAll” or “RaycastNonAlloc”, the ray will travel to the end of its range regardless of whether it hit something or not. What ends up happening is that it captures all of the colliders in front of it instead of solely the first. “RaycastAll” is a slightly less efficient version of “RaycastNonAlloc” so the signature is different but otherwise, they can be treated identically. The image on the right shows an example of a “RaycastAll”.



In order to proceed to the Mega Laser, we need to learn about some of the other types of “rays” that we can cast. When casting a ray, you are casting a single line however, it is possible to use other shapes for different circumstances! Two of the other shapes available to us are circle and box. When casting a box or a circle, you can treat it the same as a ray. Provide it with a starting point, a direction, and a range then it moves from the provided starting point to the ending point described through its direction and range. A circle cast would form this type of trail and the first thing in its path will be hit; in the function signature, you must define its radius.



Now, what if we want to model an explosion? We can do that using an overlap instead of a cast. If we take an “OverlapCircleAll” for example, this will place a circle with its center and radius defined by us. Anything with a collider within our circle will be returned. The partial signatures for “OverlapCircle” and “OverlapCircleAll” are:

```
public static Collider2D OverlapCircle(Vector2 point, float radius, int
layerMask = ...)
public static Collider2D[] OverlapCircleAll(Vector2 point, float radius,
int layerMask = ...)
```

We are finally ready to implement the Mega Laser! Start by copy-pasting everything from your “ShootLaser” method into your “ShootMegaLaser” method. Now they should be identical. The one modification to make is wherever you see “m_LaserPrefab”, change it to “m_MegaLaserPrefab” so that it is displaying the correct graphics.

The first thing we want to do is create an explosion so that the mega laser can destroy multiple enemies with a single shot. We will take advantage of “OverlapCircleAll” here. Add the following line after “AddScore()” so that it looks like:

```
Collider2D[] colliders = Physics2D.OverlapCircleAll(hit.point, 2f);
```

Essentially, we are going to check for anything within the circle of radius two, with its origin at the hit point of the laser. We need an array of “Collider2D” because of the possibility that there is more than one enemy blob in the specified region.

Next, we will simply iterate over each of the items that were in our region, check if they are an enemy and if they are, hitting them and giving ourselves a point.

```
foreach (var collider in colliders)
{
      enemy = collider.GetComponent<Enemy>();
      if (enemy != null)
      {
            enemy.Hit();
            AddScore();
      }
}
```

If go into the game and try it out now, shooting the mega laser into a group should kill the group (assuming you hit an enemy inside the group)! To make the explosion slightly more clear, let’s add an accompanying effect. After the first enemy was hit and before the first point is given, add a call to “PlayMegaLaserExplosion(...)” and use “hit.point” as the point the explosion should originate from.

Your code should look like:

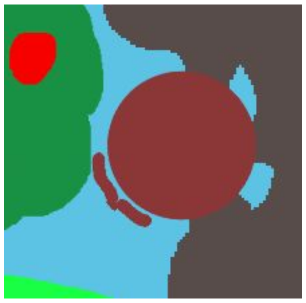
```
enemy.Hit();
PlayMegaLaserExplosion(hit.point);
AddScore();
Collider2D[] colliders = Physics2D.OverlapCircleAll(hit.point, 2f);
```

Sweet! There is now an explosion! Unfortunately, you may notice that the mega laser is large and sometimes does not explode when clearly hitting an enemy. That is because we are still using a ray which is a single line. To fix this, we will switch to casting a box instead. Go ahead and modify our initial raycast from

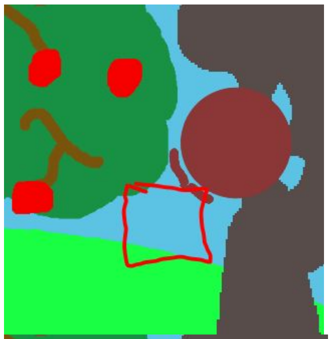
```
RaycastHit2D hit = Physics2D.Raycast(start, dir, p_LaserRange);
```

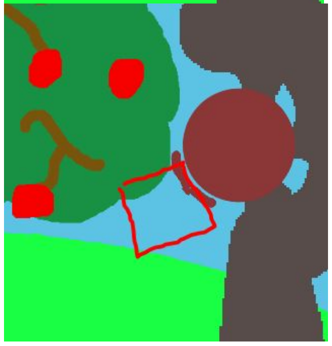
to

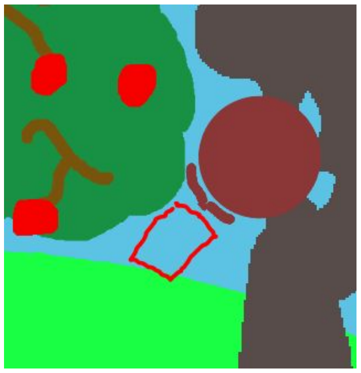
```
Vector2 size = new Vector2(1, 0.5f);
RaycastHit2D hit = Physics2D.BoxCast(start, size,
transform.localEulerAngles.z, dir, p_LaserRange);
```



As before, the origin is at start however instead of just using a direction and range, we also have a size and an angle. The size is to define the size of our box. If we choose a very small size, it may be preferable to just switch back to using a ray instead. As for the angle, we just ensured that it is facing the direction of our laser. In the instance to the right, we want the box angled appropriately. That means we do not want a box with the angles shown in the left two images. What we do want is a box with the angle show in the image below.







With that, we are just about done. Unfortunately, if you play the game now and aim down, you will experience some weird behaviour. The behaviour is being caused by the box cast colliding with the collider on the right of the screen (used to know when an enemy cannot be killed anymore).

Luckily for you, the enemies are on a different physics layer that we can take advantage off. All we have to do is tell our box cast exactly which layers it is allowed to hit. We will be using something called a layer mask to achieve this. As you know, if you go into Tags and Layers under Project Settings, you will see all of the available layers. Now image that we had a binary representation of 0. If you want the box to collide with a particular layer, you must set its corresponding bit to a 1. For example, the Default layer is the first layer so we can assume it corresponds to the first bit. Furthermore, for the sake of this example, assume we have a total of five layers (equating to five bits). If we want to only affect the Default layer, we use the binary number 00001. If we want to affect every layer except for the Default layer, we use the binary number 11110.

For our project, we want the box to collide with the Enemy and AntiExplosionEnemy layers. The problem is that we do not know which bit to turn on so we have to go in a roundabout way to create our binary number. First, we take the number 1, then we shift it by the layers "number". To retrieve a layer's "number", we do "LayerMask.NameToLayer(...)". After performing this on our two layers, we perform a bitwise or operation. The box cast should now look like:

```
RaycastHit2D hit = Physics2D.BoxCast(start, size,
transform.localEulerAngles.z, dir, p_LaserRange, 1 <<
LayerMask.NameToLayer("Enemy") | 1 <<
LayerMask.NameToLayer("AntiExplosionEnemy"));
```

Check Off

- Make the explosion of the mega laser only affect enemies on the Enemy layer
- Run the game and show the following (feel free to add more spawners in):
    - Aiming down and shooting the mega laser shoots all the way down
    - Hitting an enemy with the mega laser will kill enemies in the surrounding region
    - The normal baby laser can kill enemies too
- Explain what raycasting means.
- Provide an example of raycasting (unrelated to this lab).