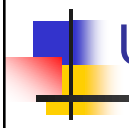


Unity Game Engine

Introduction to Unity – Adding projectile and enemies

Unity Manual: http://docs.unity3d.com/Manual/index.html
Unity Script References: http://docs.unity3d.com/ScriptReference/index.html

A basic FPS

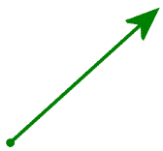
- Continuing to the previous work, let's add features for
  - Shooting into the scence
  - Detecting and responding to hits
  - Making characters/enemies that wander around
  - Spawning new objects(enemies) in the scene

Detail steps to take

1. Enable the player to shoot into the scene.
2. Create static targets that react to being hit.
3. Make the targets wander around.
4. Spawn the wandering targets automatically.
5. Enable the targets/enemies to shoot fireballs at the player.

Ray

- A *ray* starts out at a point (origin) and extends out in a specific direction infinitely.



What is Raycasting?

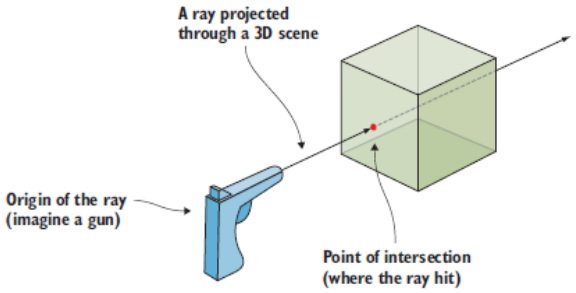
- Racasting is the use of ray–surface intersection tests to solve a variety of problems in computer graphics and computational geometry.
- Raycasting tells us what objects in the environment the ray runs into. In otherwords, Raycasting determines what intersects the ray when you cast a ray.

When do we use Raycasting?

- To handle shooting
- To determine what's 'in front of' (line of sight) an autonomous agent for AI purposes.
- …

Shooting via Raycasting

- When you fire a bullet from a gun: A ray is analogous to the path of the bullet, and raycasting is analogous to firing the bullet and seeing where it hits.

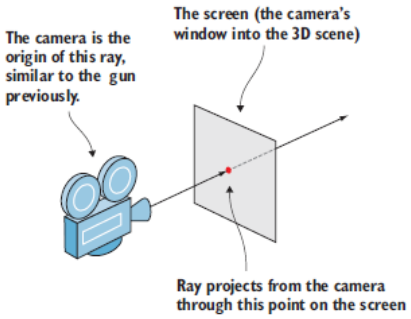


Raycasting for FPS

- For a FPS, the ray generally starts at the camera position (ray origin) and then extends out through the center of the camera view.
  - In other words, you're checking for objects straight in front of the camera.

Raycasting for FPS

- We'll implement shooting by projecting a ray that starts at the camera and extends forward through the center of the view.
  - This is a special case of an action referred to as *mouse picking*.

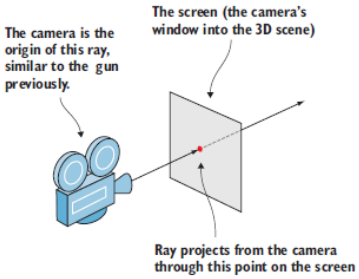


Detail steps to take

1. Enable the player to shoot into the scene.
2. Create static targets that react to being hit.
3. Make the targets wander around.
4. Spawn the wandering targets automatically.
5. Enable the targets/enemies to shoot fireballs at the player.

Raycasting for FPS in Unity

- Generate Ray & Calculate the hit point
  - Use **Camera.ScreenPointToRay(point)** to create a ray that starts at the camera and projects at an angle passing through the center of the screen for a FPS.
  - Pass the ray to the method **Physics.Raycast(ray, out hit)** to perform raycasting using that ray.



Functions for Raycasting

- **Camera.ScreenPointToRay(**Vector3 **position);**
  - Returns a ray object going from camera through a screen point. Resulting ray is in world space.
  - **position** is (x,y) pixel coordinates on the screen**. Position.z** is ignored). The bottom-left of the screen is (0,0); the right-top is (pixelWidth -1,pixelHeight -1).

- **Physics.Raycast(**Ray **ray**, **out** RaycastHit **hitInfo**);
  - Returns True if the **ray** intersects with a Collider, otherwise false.
  - If true is returned, **hitInfo** will contain more information about where the collider was hit.
  - **out** keyword causes arguments to be passed by reference.

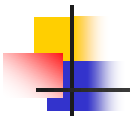
RayShooter.cs Script

- Create a new C# script, RayShooter.cs.
- Attach it to the camera (not the player object).


```
using UnityEngine;
using System.Collections;
public class RayShooter : MonoBehaviour {
  private Camera _camera;
  void Start() {
    _camera = GetComponent<Camera>();
  }
  void Update() {
    if (Input.GetMouseButtonDown(0)) { //mouse left button down
       // create a ray that pass through the screen center
      Vector3 point = new Vector3(_camera.pixelWidth/2,
                                  _camera.pixelHeight/2, 0);
       Ray ray = _camera.ScreenPointToRay(point);

      // Raycast
      RaycastHit hit;
      if (Physics.Raycast(ray, out hit)) {
        Debug.Log("Hit " + hit.point);
      }
    }
  }
}
```

The script shoots a ray when the mouse has been clicked and displays the hit point to the console.



RayShooter.cs Explained

- The camera component is retrieved in `Start()`.
- The code checks if the mouse left buttons was pressed.
- `Input.GetMouseButtonDown(int button)`
  - Returns true during the frame the user pressed the given mouse button.
  - `button`: 0 for left button, 1 for right button, 2 for middle button.
- `_camera.ScreenPointToRay(point)`
  - z value of the point will be ignored.
  - (x,y) of the point is a screen point. The bottom-left of the screen is (0,0); the right-top is (pixelWidth -1,pixelHeight -1).

RayShooter.cs Explained (2)

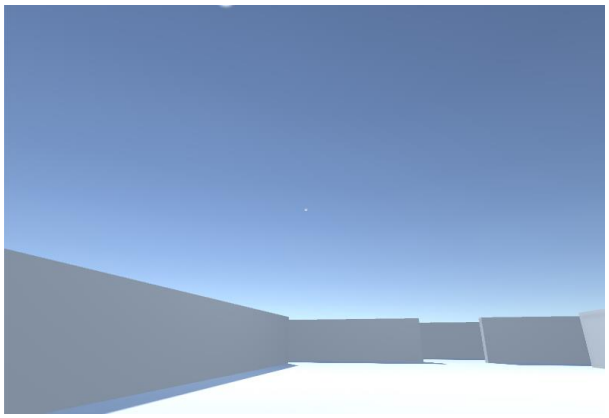
- `RaycastHit` data structure includes a bundle of information about the intersection of the ray, including the intersection/hit point and the hit object.
  - https://docs.unity3d.com/ScriptReference/RaycastHit.html
- `Physics.Raycast()` method checks for intersections with the given ray, fills in data about the intersection, and returns true if the ray hit anything.
- The code emits a console message to indicate when an intersection occurred with the 3D coordinates of the point where the ray hit.

Adding Visual Indicators – Aiming Spot

- an aiming spot on the center of the screen

```
//RayShooter Script
...
  void Start() {
    _camera = GetComponent<Camera>();
    Cursor.lockState = CursorLockMode.Locked;
    Cursor.visible = false;
  }
  void OnGUI() {
    int size = 12;
    float posX = _camera.pixelWidth/2 - size/4;
    float posY = _camera.pixelHeight/2 - size/2;
    GUI.Label(new Rect(posX, posY, size, size), "*");
  }
...
```


Adding Visual Indicators – Aiming Spot



Add Aiming Spot

- Cursor settings in `Start()`
  - hides the mouse cursor at the center of the screen.
- `OnGUI()`
  - Every `MonoBehaviour` automatically responds to an `OnGUI()` method.
  - `OnGUI()` runs every frame right ***after*** the 3D scene is rendered, resulting in everything drawn during `OnGUI()` appearing on top of the 3D scene.
- `GUI.Label()` displays an asterisk (*) in the center of the screen.

Detail steps to take

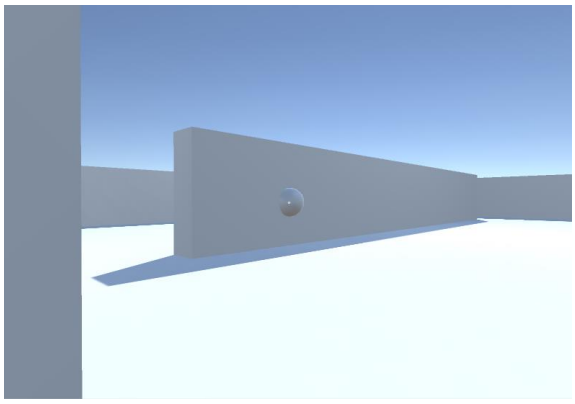
1. Enable the player to shoot into the scene.
2. Create static targets that react to being hit.
3. Make the targets wander around.
4. Spawn the wandering targets automatically.
5. Enable the targets/enemies to shoot fireballs at the player.

Types of Targets

- Environment (static target)
  - When the ray hits, show a bullet hole.
- Enemy (moving target)
  - When the ray hits, make it die.

Adding Visual Indicators – Bullet Hole

- Bullet hole - a mark where the ray hit (let's put a sphere)



Adding Visual Indicators – Bullet Hole

- StartCoroutine() sets a coroutine in motion.

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


Unity Coroutines

- When you call a function, it runs to completion before returning.
- This effectively means that a function call can't be used to contain a procedural animation or a sequence of events over time.
  - See Fade() function example.

Unity Coroutines (2)

- Consider the Fade() function called in Update() to fade the scene.

```
void Update() {
..
    Fade()
..
}
void Fade() {
    for (float f = 1f; f >= 0; f -= 0.1f) {
        Color c = renderer.material.color;
        c.a = f;
        renderer.material.color = c;
    }
}
```

- It won't see the effect you want. The object will disappear instantly. In order for the fading to be visible, the alpha must be reduced over a sequence of frames to show the intermediate values being rendered. However, the function will execute in its entirety within a single frame update. The intermediate values will never be seen.

Unity Coroutines (3)

- Solution: you can design Update function 1) to fade on frame-by-frame basis or 2) use a coroutine.
- Coroutines are a great way of writing routines that need to happen over time.

Unity Coroutines (4)

- A coroutine is declared like this:

```
IEnumerator Fade() {
    for (float f = 1f; f >= 0; f -= 0.1f) {
        Color c = renderer.material.color;
        c.a = f;
        renderer.material.color = c;
        yield return null;
    }
}
```

Return type should be **IEnumerator** and the **yield keyword in the return statement**. Any function that includes the yield statement is understood to be a coroutine and the IEnumerator return type need not be explicitly declared:

Unity Coroutines (5)

- To set a coroutine running (invoke), you need to use the `StartCoroutine` function.
- By default, a coroutine is resumed on the frame after it yields
- It is also possible to introduce a time delay using `WaitForSeconds`.

```
void Update() {
    if (Input.GetKeyDown("f")) {
        StartCoroutine("Fade");
    }
}
IEnumerator Fade() {
    for (float f = 1f; f >= 0; f -= 0.1f) {
        Color c = renderer.material.color;
        c.a = f;
        renderer.material.color = c;
        yield return new WaitForSeconds(.1f);
    }
}
```

SphereIndicator Coroutine

```
...
  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)) {
        StartCoroutine(SphereIndicator(hit.point));
      }
  }
  private IEnumerator SphereIndicator(Vector3 pos) {
    GameObject sphere =
      GameObject.CreatePrimitive(PrimitiveType.Sphere);
    sphere.transform.position = pos;
    yield return new WaitForSeconds(1);
    Destroy(sphere);
  }
...
```


SphereIndicator Coroutine

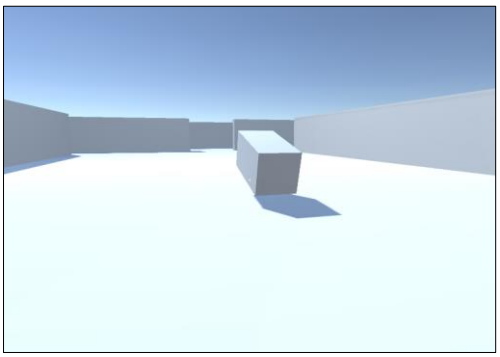
- This method creates a sphere at a point in the scene and then removes that sphere a second later.
- The **yield** keyword causes the coroutine to temporarily pause, handing back the program flow and picking up again from that point in the next frame.
- WaitForSeconds(1) causes the coroutine to pause for one second.

Unity Coroutines (6)

- Coroutines seemingly run in the background of a program, through a repeated cycle of running partway and then returning to the rest of the program.
  - Coroutines are *not* threads. It will be executing on the main thread of the game.
  - A coroutine is like a function that has the ability to pause execution and return control to Unity but then to continue where it left off on the following frame.

Enemy (moving target)

- Enemy (moving target)
  - When the ray hits, make it die.
  - The target object will fall over and disappear when you shoot it.



Create a Target Enemy

- Create an enemy
  - Create a new cube object (Navigation: GameObject > 3D Object > Cube)
  - Scale as you want. Elongate it vertically.
  - Name the object Enemy.
- Create a new script called **ReactiveTarget.cs**.
- Attach the script to the Enemy.

ReactiveTarget.cs

```
using UnityEngine;
using System.Collections;
public class ReactiveTarget : MonoBehaviour {
   public void ReactToHit() {
     StartCoroutine(Die());
   }
   private IEnumerator Die() {
     this.transform.Rotate(-75, 0, 0);
     yield return new WaitForSeconds(1.5f);
     Destroy(this.gameObject);
   }
}
```

The target object will fall over and disappear when you shoot it.

Modify Rayshooter.cs Script

```
…
        RaycastHit hit;
        if (Physics.Raycast(ray, out hit)) {
         GameObject hitObject = hit.transform.gameObject;
         ReactiveTarget target =
             hitObject.GetComponent<ReactiveTarget>();
         if (target != null) {
             //Debug.Log("Target hit");
             target.ReactToHit();
         } else {
           StartCoroutine(SphereIndicator(hit.point));
         }
…
```


Resource

- RayShoot.cs
- ReactiveTarget.cs