DH2323 Lab3 Track2

Collision Detection and Level of Details

1. Introduction

This lab will introduce you to collision detection which is the computational problem of detecting the intersection of two or more objects. This will form a basis for doing intersection tests between shells and the tank in order to calculate whether they hit it. You will apply particle effects for successful hits using the particle system code from Lab 2.

Finally, you will apply a discrete level-of-detail (LOD) scheme for the tank so that meshes with fewer primitives are used when the tank is further away from the viewer and specific details cannot be seen.

Note that in order to complete this lab you should have first completed Lab 2 of the animation track. Once you check that everything is built properly, you should integrate your codes from Lab 2 into this lab.

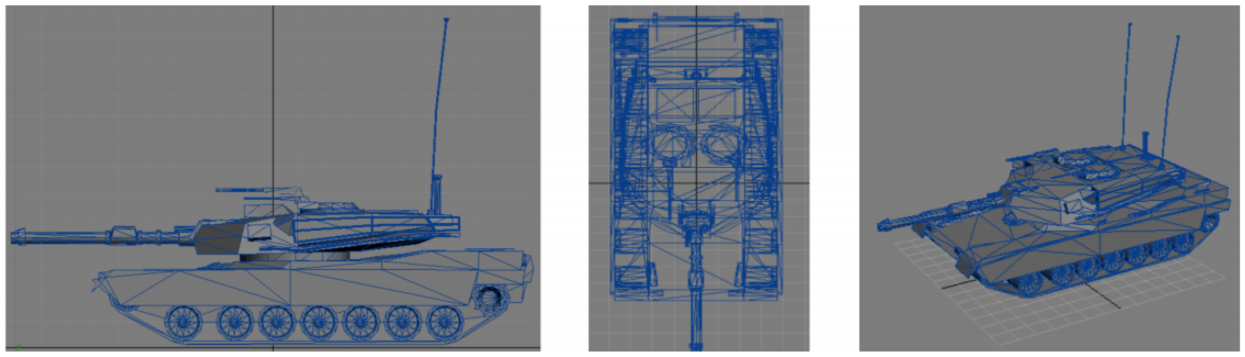


Figura 1: Wireframe view of the tank model from a number of different view angles. Tank model by knoxville@list.ru.

The lab consist of four related tasks.

- Get familiar with colliders and collision detection techniques in Unity.
- Use colliders to perform some simple collision checks. You can combine it with completed tasks in Lab2.
- Implement a basic discrete level-of-detail (LOD) scheme for the tank based on its distance to the camera, so that simplified tank geometry is displayed in case that the viewer cannot see any noticeable differences with respect to the more detailed version particles in real-time.

2. Collision Detection

This section will work mainly in the ***CollisionDetectionScene***. Collision detection in Unity will be introduced in this section, and you will combine ***ShellScene*** and ***TankScene*** from previous lab (Lab2) using collision detection to create a simple tank game.

2.1. Collision Detection

In games, you will often need to detect collisions between objects. This can be related to objects that you will collect or interact with. For this purpose, you can use ***Colliders***. However, the way you manage collisions, and the methods to be called in this case, will highly depend on the type of colliders used (for example, colliders or triggers) as well as the object from which you want to detect the collision. In some case, if you want to detect the presence of objects located far away from the player or the NPC, ray-casting may make more sense than collision or trigger detection.

In this section, ***CollisionDetectionScene*** will be used as the main scene. This scene is almost the same as TankScene in Lab2 with augmentations in terrain as shown in Figure 2. Please attach ***TankController.cs*** from Lab2 to the tank gameobject in this scene. When run the scene, the camera will always follow the tank.

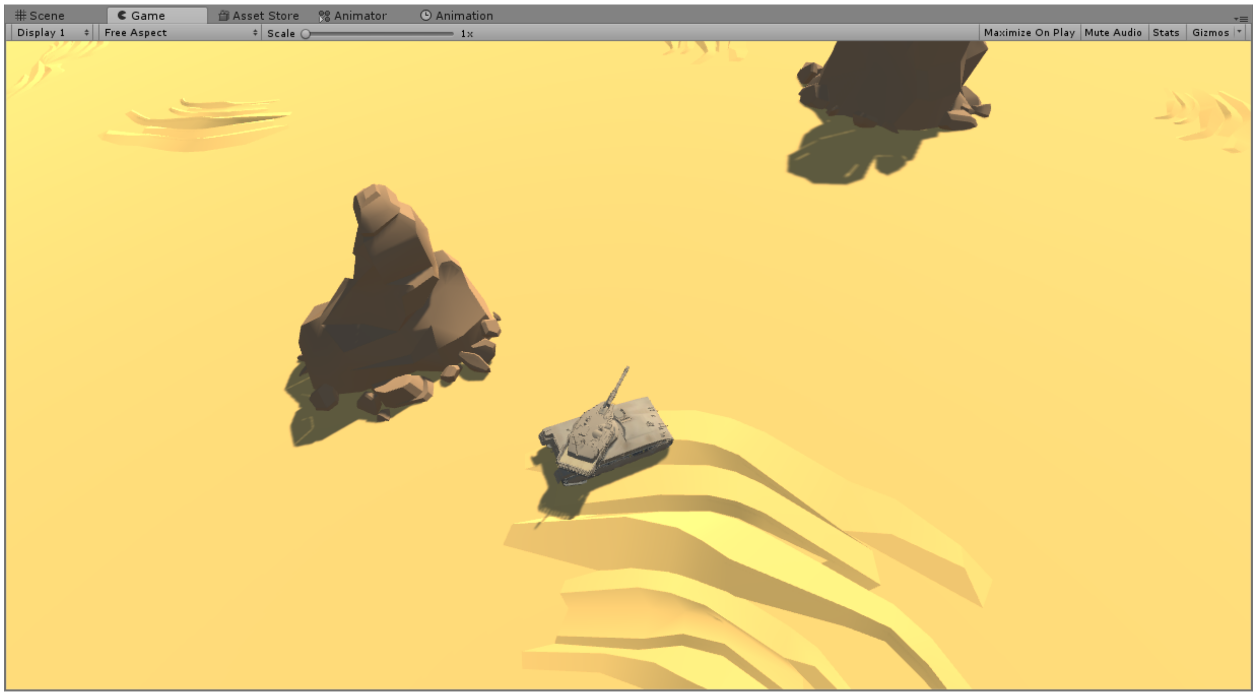


Figura 2: The terrain contains dunes and rocks.

2.2. Collider

Colliders define the shape of an object for the purposes of physical collisions. A collider, which is invisible, need not be the exact same shape as the object's mesh and in fact, a rough approximation is often more efficient and indistinguishable in gameplay. Basically, colliders have two types: primitive collider and mesh collider. In 3D, primivite colliders are the Box Collider, Sphere Collider and Capsule Collider. Any number of these can be added to a single object to create *compound colliders*. For example, as shown in Figure 3, the compound collider of the tank contains several sphere colliders with each sphere collider bounds each sub-object.

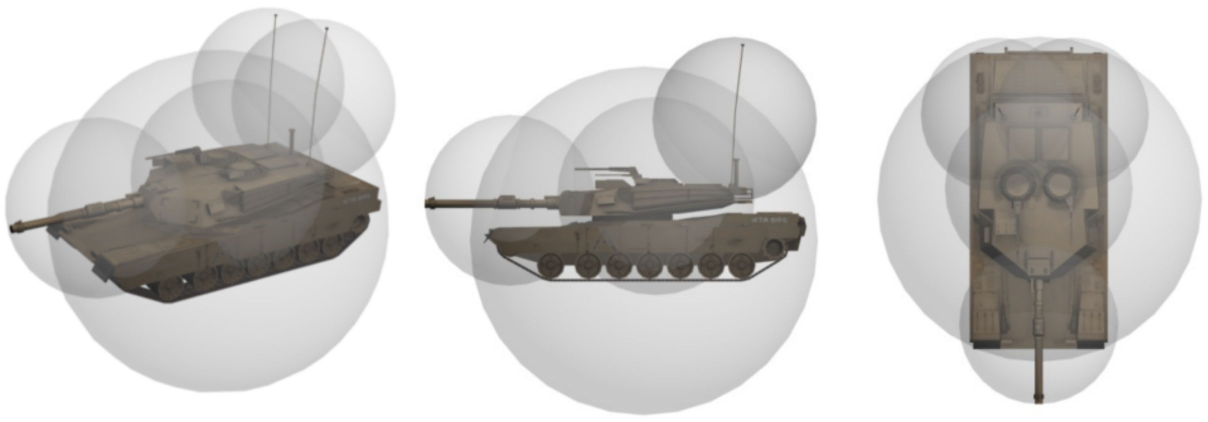
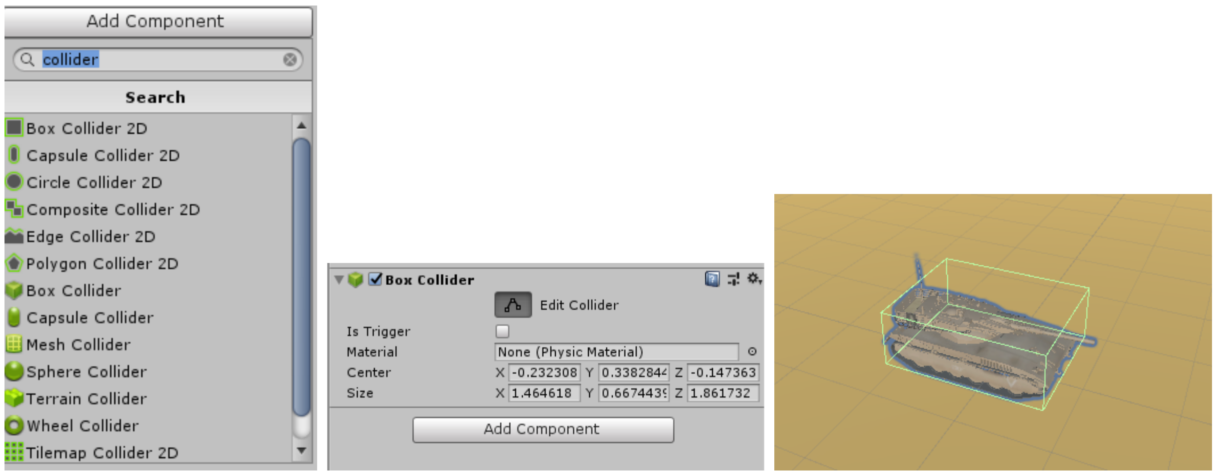


Figura 3: Illustration of the compound colliders for the tank. Each sub-object in the tank is enclosed by its own sphere collider.

Collider can be simply added to a gameobject by click Add Component and search for Collider in the Inspector. As shown in Figure 4, collider shape can be edited by enable ***Edit Collider*** (middle), in order to better bound the model. With the collider attached to the rock, it can prevent tank go inside rock mesh as shown in Figure 5. Note that tank has a box collider attached already.

Referring to the Unity documentation about colliders[1], colliders interact with each other differently depending on how their Rigidbody components are configured. The three important configurations are the Static Collider (i.e., no Rigidbody is attached at all), the Rigidbody Collider and the Kinematic Rigidbody Collider. As we can see from the Unity documentation, the capsule collider attached to rocks are Static Colliders, and the box collider attached to the tank is a Rigidbody Collider.

[1]https://docs.unity3d.com/Manual/CollidersOverview.html



Is Trigger ☐
Material None (Physic Material)
Center X -0.232308 Y 0.338284 Z -0.147363
Size X 1.464618 Y 0.667443 Z 1.861732

Figura 4: Add and edit a collider.

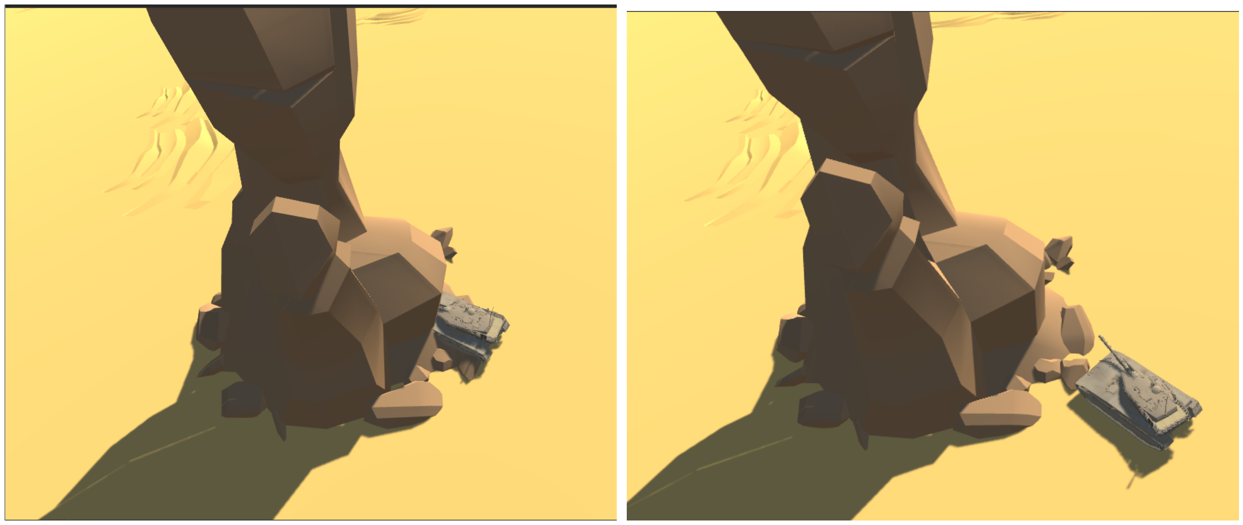


Figura 5: The rock with no collider attached (left), and with a capsule collider attached (right).

2.3. Trigger

The scripting system can detect when collisions occur and initiate actions using the functions such as ***OnCollisionEnter()***[2]. OnCollisionEnter() uses ***Collision*** class which contains information about contact points, impact velocity etc. You might use this function to further calculate collision response. However, you can also use the physics engine simply to detect when one collider enters the space of another **without** creating a collision. A collider configured as a ***Trigger*** (using the Is Trigger property) does not behave as a solid object and will simply allow other colliders to pass through. When a collider enters its space, a trigger will call the ***OnTriggerEnter()*** function on the trigger object's scripts .

If you checked ***Explosion.cs*** in Lab2, you will find OnTriggerEnter() has already been used to trigger particle system as shown in Listing 1. Note that '***Is Trigger***' needs to be

[2]https://docs.unity3d.com/ScriptReference/Collider.OnCollisionEnter.html

checked in order to use trigger.

```
public void OnTriggerEnter(Collider other)
{
    if (explosionParticlesPrefab)
    {
        GameObject explosion = (GameObject)Instantiate(
    explosionParticlesPrefab, transform.position, explosionParticlesPrefab.
    transform.rotation);
        Destroy(explosion, explosion.GetComponent<ParticleSystem>().main.
    startLifetimeMultiplier);
    }
}
```

Listing 1: Explosion.cs

2.4. Raycast

In games, especially shooting games, it is not efficient to use collision detection for fast moving object like bullet. However, we can use ray cast to simulate shooting bullets. We touched Raycast in Lab2 already (guide turret to look at mouse cursor), and we can use similar raycast method to simulate firing from the secondary gun.

2.5. Tasks

2.5.1. Radar Detection

In the scene, you will find a Radar gameobject with ***RadarAlarm.cs*** attached. In this script, you will find ***DoRender()*** function which draws a ring arond the radar to indicate the detection range. Also, a capsule collider is already attached to the Radar.

Use ***OnTriggerEnter*** and ***OnTriggerExit*** to make the radar detect the tank, i.e. if the tank enters the detecting range (presented by the capsule collider), the ring should turn red (the material can be found in folder Resources/Materials/Alarm) as shown in Figure 6.

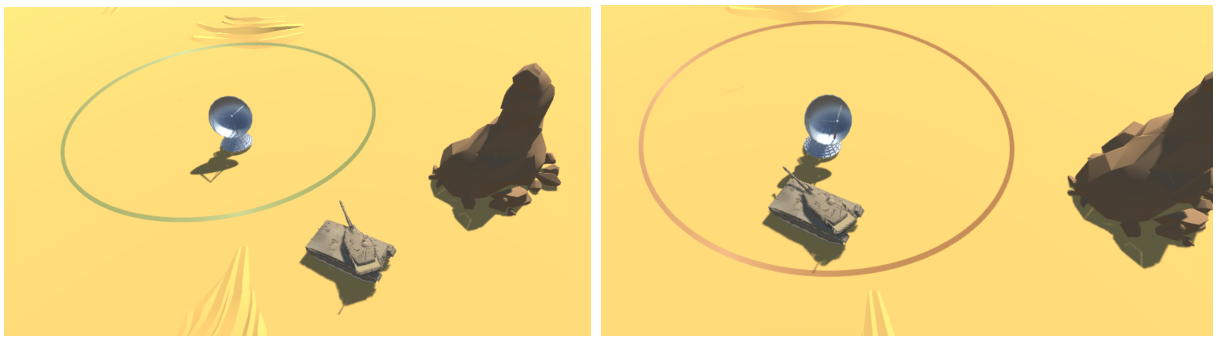


Figura 6: If the tank enters the radar's detecting range, the ring will turn red. Otherwise, the ring stays green.

Note: Only code is not enough, you should justify why you do that.

```
public void OnTriggerEnter(Collider other)
{
    // Your code here.
}

public void OnTriggerExit(Collider other)
{
    // Your code here.
}
```

Listing 2: RadarAlarm.cs

Hints and tips:
Current capsule collider attached to the radar can detect all colliders which have collisions with it. You need to select the tank collider from all detected colliders.

You might also think of using ***Physics.OverlapSphere***[3] without adding any collider onto radar (This is optional).

2.5.2. Firing of the Secondary Gun

In Section 2.4, we talked that fast moving bullet or shooting can be simulated using raycast in computer games. In this task, you will need to implement the secondary gun shooting as shown in Figure 10. When the right mouse key is pressed, the secondary gun will shoot.

Go to the tank hierarchy in the scene, in its children objects, you will find Secondary Gun (Tank/Turret/Secondary). ***SecondaryGunShoot.cs*** is attached to the gun barrel end. In the ***Shoot()*** function, you need to use raycast to determine where the ray ends, i.e. if there a a rock in the front, the ray should be stopped by the rock, otherwise, it will reach to the maximal raycast range as shown in Listing 3.

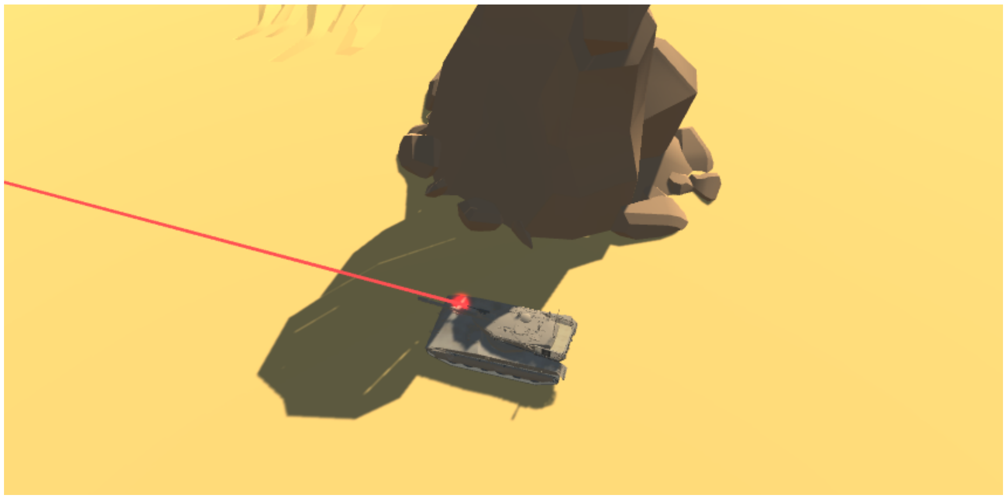


Figura 7: Scene view and game view.

[3]https://docs.unity3d.com/ScriptReference/Physics.OverlapSphere.html

```
void Shoot()
{
    // Reset the timer.
    timer = 0f;

    // Stop the particles from playing if they were, then start the particles.
    gunParticles.Stop();
    gunParticles.Play();

    // Enable the line renderer and set it's first position to be the end of the gun.
    gunLine.enabled = true;
    gunLine.SetPosition(0, transform.position);

    // Set the shootRay so that it starts at the end of the gun and points forward from the barrel.
    shootRay.origin = transform.position;
    shootRay.direction = transform.forward;

    // Perform the raycast against gameobjects on the shootable layer and if it hits something...
    // If it hits something, set the second position of the line renderer to the point the raycast hit, otherwise, set the second position of the line renderer to the maximal raycast range.

    // Your code here.
}
```

Listing 3: SecondaryGunShoot: Shoot()

Note: Only code is not enough, you should justify why you do that.

Hints and tips:
You might think of using these two functions ***Physics.Raycast()*** and ***LineRenderer.setPosition()***.

2.5.3. Firing of the Main Gun

Referring to Lab2, we have a ***ShellScene***. Now we integrate that scene to this lab. An empty gameobject is created and attached to the main gun (Tank/Turret/Maingun). By simply attaching ***ShellEmitter.cs*** is not enough, you also need to rescale the size of shell as shown in Figure 8. You might think of using *transform.localScale*.

2.5.4. Destroy Enemy Tank

- Referring to the Radar Detection task, here you need to spawn enemy tanks when the tank is detected by the radar. The ***Spawn()*** function in RadarAlarm.cs spawns enemy tank which moves towards the tank you controlled. Uncomment ***InvokeRepeating()*** (in OnTriggerEnter()) and ***CancelInvoke()*** (in OnTriggerExit) to enable the spawning process.

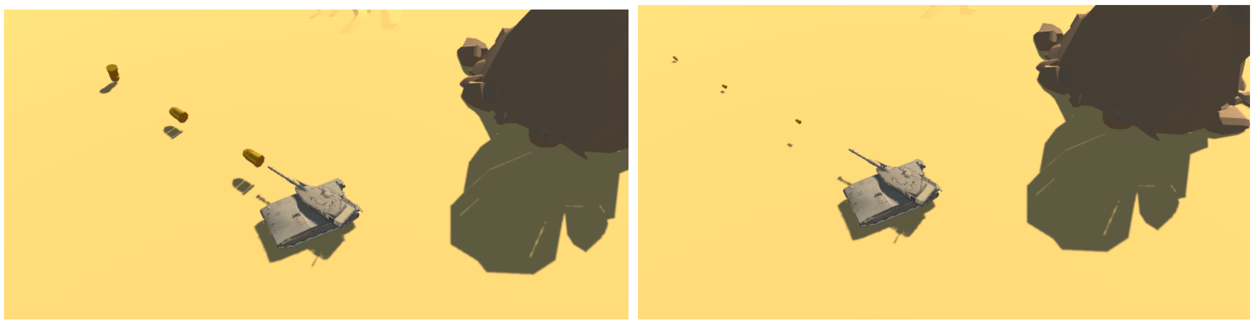


Figura 8: The instantiated shell should be rescaled as shown in the right image.

- You will find ***ShellCollision.cs*** attached to Shell prefab. Add collision detection in this script in order to destroy the shell if it hits something (e.g. rock, ground, enemy tank, etc.). If the shell hits enemy tank, the enemy tank will also be destroyed.
- Define your own particle system when the shell hits something. A good reference can be found in Resources/Prefabs/ShellExplosion.
- Enable the secondary gun to destroy the enemy tanks.

Finally, you will see similar images as shown in Figure 9

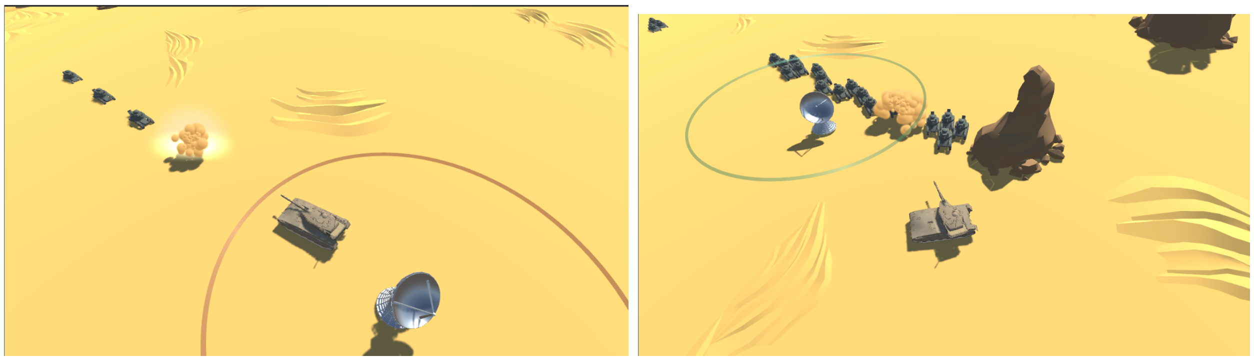


Figura 9: The enemy tanks are spawn when the tank is detected by the radar. And enemy tanks can be destroy by main gun.

Note: Only code is not enough, you should justify why you do that.

3. Level of Details

Early video games had very noticeable changes in the level of detail, with objects in the background visibly gaining definition (and polygons) as they entered the foreground. This was a simple shortcut to save processing power for other tasks while still presenting a passable background to the player. Modern video games still use the same procedure, but the base level from which they render has a much higher polygon count, meaning few people can spot the difference between foreground and background with the naked eye.

In Unity, when a GameObject in the Scene is far away from the Camera, you can't see very much detail, compared to when the GameObject is close to the Camera. And even though you can't see the detail on a distant GameObject, Unity uses the same number of triangles to render it at both distances. To optimize rendering, you can use the Level Of Detail (LOD) technique. It allows you to reduce the number of triangles rendered for a GameObject as its distance from the Camera increases.

3.1. Task

Creating LOD is quite straightforward in Unity[4]. Please create a LOD model for enemy tank in order to display low level of details when the enemy is away from the camera and display high level of details when near. Low level of details model can be found in Resources/Prefabs/enemytanklowmesh.

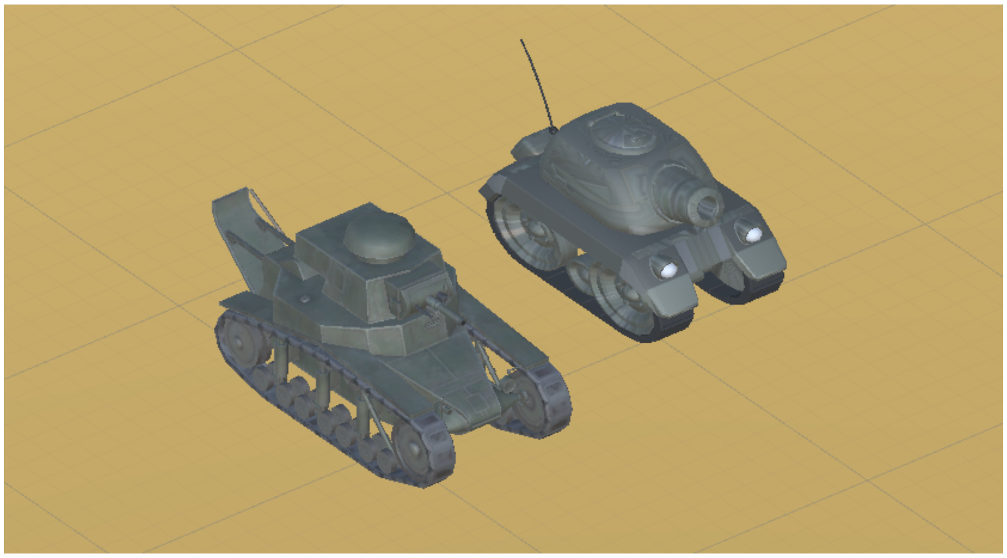


Figura 10: Left enemy tank is of high details and the right one is of low details.

[4]https://docs.unity3d.com/Manual/class-LODGroup.html