DH2323 Lab2 Track2

Transformations and Particle System

1. Introduction

This lab will introduce you to transformations, which allow you to transform objects in 3D environments and relative to each other on Unity3D[1]. An understanding of transformations is fundamental to computer animation and this lab will provide you with the knowledge required to completer further tasks in Lab3.

Before assembling After assembling

Figura 1: This lab is a basis for working toward a tank model.

The lab consist of four related tasks.

- Get familiar with programming on Unity by checking existing ***TankController.cs*** script.
- Investigate translation and rotation transformations for tanks.
- Implement two functions to rotate wheels and turret. Make the wheels rotate in the correct way as tank moves forward or backward. Rotate the turret to make it follow the mouse cursor.
- Use a basic particle system to generate, update and visualize large numbers of blended particles in real-time.

[1] https://unity.com/

2. Transformations

Open up a new project in Unity. Go to ***Asset–>Import Package–>Custom Package***. After lab2 package is imported, you will find in the project window that the assets are sorted into several folders:

- Resources: Holds all resources used in current project.
  - Materials: Holds the shell material.
  - Models: Contains the tank, bonfire, and shell models and their materials.
  - Prefabs: Holds the shell prefab.
- Scenes: Contains the Tank scene, Fireflame scene, and Shell scene.
- Scripts: Holds the initial scripts.

This section will deal with script ***TankController.cs***. This script should be attached to *Tank* gameobject[2] in the scene in order to apply tank movement.

2.1. Tank Hierarchy

In Unity Hierarchy view which shows all objects in current scene, a tank object can be found. This tank contains sub-objects such like wheels, turret, gun, etc as shown in Figure 2. This heirarchical relation can also be called parent-child relation. A parent object causes all children objects to move and rotate the same way the parent object does, although moving children objects does not have any effect on the parent. Children themselves can be parents; e.g., *Tank* object is the parent of *Wheels*, *Body*, *Turret* objects. And *Turret* object is the parent of *Main gun*, *Secondary gun* objects.

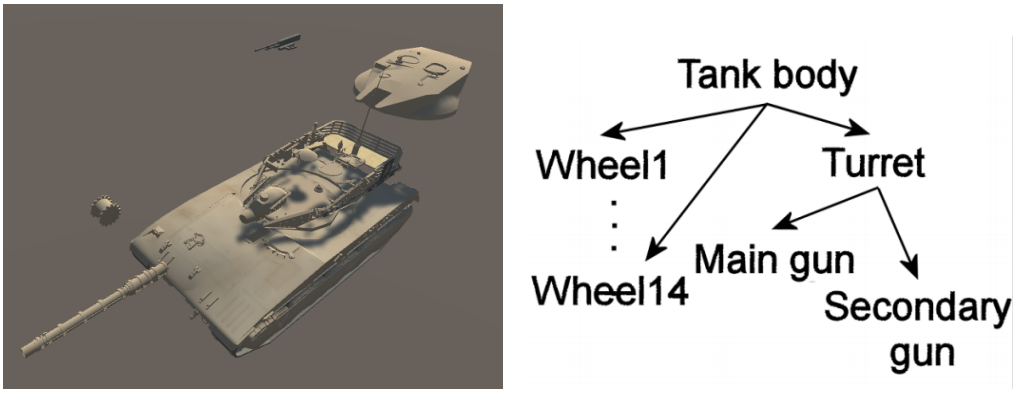


Figura 2: The hierachical structure of the tank.

[2]Almost everything in Unity (models, camera, lights, audio, etc.) is a gameobject.

2.2. Tank Movement

We will use *'W,A,S,D'* or *'up, left, down, up'* in our keyboard to control tank movement and rotation. These keys are called ***directional keys***. In C++, we can use SDL to get access to keyboard, mouse, joystick, etc. However, in Unity, all these inputs can be accessed through ***Edit->Project Setting->Input***, where you can find mouse inputs, keyboard inputs, etc. In this subsection, we use keyboard input, specifically, *'W, A, S, D'* or *'up, left, down, up'* keys. And these directional keys are controlled by two axes: ***Horizontal*** and ***Vertical***. If you want to use other keys, you can refer to *Input.GetKeyDown()* function.

We use ***Input.GetAxis()*** to get the value of input axes. The value will be in the range -1...1 for keyboard and joystick input. If the axis is setup to be delta mouse movement, the mouse delta is multiplied by the axis sensitivity and the range is not -1...1.

In the following ***Update()***[3] function, the input value of directional keys are obtained.

```
1    private void Update()
2    {
3        // Store the value of both input axes.
4        m_MovementInputValue = Input.GetAxis("Vertical");
5        m_TurnInputValue = Input.GetAxis("Horizontal");
6    }
```

Listing 1: Input.GetAxis() exmple

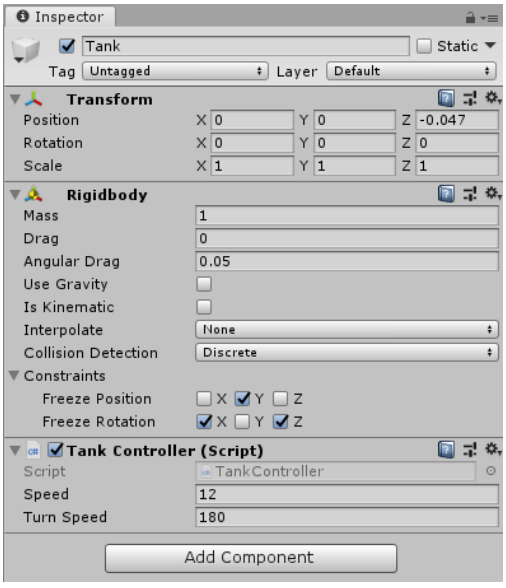


Figura 3: Inspector of Tank.

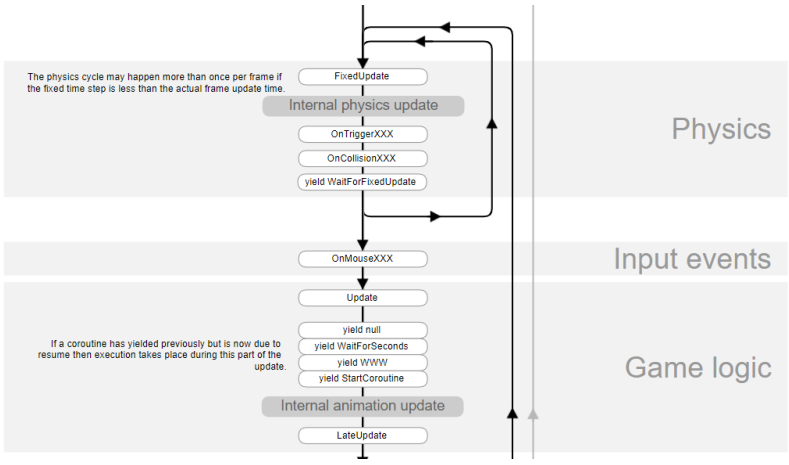


Figura 4: Update order.

Once we have these input values, we can use them to move and rotate the tank as shown in Listing 2 and Listing 3. ***Rigidbody*** is used to control an object's position through physics simulation. Adding a Rigidbody component to an object will put its motion under the control of Unity's physics engine. The Rigidbody also has a scripting API that lets you apply forces to the object and control it in a physically realistic way. Here we use ***MovePosition()*** and

[3]Update() function is called per frame in Unity.

MoveRotation() in order to move and turn the tank. Referring to Unity documentation of MoveRotation(), you will find the input parameter is a quaternion. We bring in the introduction of quaternion here in case you are not familiar with it.

Quaternion

Quaternions are unit vectors on a 3-sphere in 4 dimensional space. Defined like complex numbers but with 4 coordinates. $q[w,(x,y,z)]$ also written $q[w,v]$, where $v=(x,y,z)$. Here, w is real part, and (x,y,z) are imaginary parts. Unit quaternions provide a convenient mathematical notation for representing orientations and rotations of objects in three dimensions.

In Listing 3, the quaternion can be obtained from euler angles (Line 7).

```
1    private void Move()
2    {
3        // Create a vector in the direction the tank is facing with a
     magnitude based on the input, speed and the time between frames.
4        Vector3 movement = transform.forward * m_MovementInputValue * m_Speed
     * Time.deltaTime;
5
6        // Apply this movement to the rigidbody's position.
7        m_Rigidbody.MovePosition(m_Rigidbody.position + movement);
8    }
```

Listing 2: Move tank

```
1    private void Turn()
2    {
3        // Determine the number of degrees to be turned based on the input,
     speed and time between frames.
4        float turn = m_TurnInputValue * m_TurnSpeed * Time.deltaTime;
5
6        // Make this into a rotation in the y axis.
7        Quaternion turnRotation = Quaternion.Euler(0f, turn, 0f);
8
9        // Apply this rotation to the rigidbody's rotation.
10       m_Rigidbody.MoveRotation(m_Rigidbody.rotation * turnRotation);
11   }
```

Listing 3: Turn tank

Unlike C++ project, there is no main() function as an entry point. Unity runs C# scripts attached to the gameobjects. The entry point will be ***Awake()/Start()*** function, followed by ***FixedUpdate()/Update()/LateUpdate()***[4] as shown in Figure 4. Both tank movement and rotation deal with tank physics, they should be triggered in FixedUpdate() as shown below:

```
1    private void FixedUpdate()
2    {
3        // Adjust the rigidbodies position and orientation in FixedUpdate.
4        Move();
```

[4]Execution Order of Event Functions: https://docs.unity3d.com/Manual/ExecutionOrder.html

```
        Turn();
    }
```

Listing 4: Turn tank

2.3. Tasks

2.3.1. Rotate Wheels when Tank Moves

In the script *TankController.cs*, you will find a list variable ***m_wheels***, it contains all the wheels of the tank. The wheels should move with tanks, i.e. when tank moves forward, the wheels should rotate forward; when tank moves backwards, the wheels should rotate backwards. You should use quaternion to rotate the wheels. Note that there is no rigidbody attached to the wheels, you might think of using *transform.localRotation* (as shown below) instead of using *rigidbody.rotation* to rotate the wheels.

```
m_wheels[i].transform.localRotation
```

The code should be written in ***RotateWheels()*** function. In order to make wheels rotate with the same direction as tank's, you can think of using the sign of ***m_MovementInputValue*** to determine if wheels should rotate forward or backward.

```
    private void RotateWheels()
    {
      // Rotate tank wheels. When tank moves forward, the wheels should rotate forward; When tank moves backwards, the wheels should rotate backwards.
      // Your code here
    }
```

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

Hints and tips:
You can put the scene view and the game view side by side as shown in Figure 5. By selecting one wheel, you can see its rotation in the scene view to test if the wheel actually moves and if it moves in the right direction.

2.3.2. Rotate Turret Following Mouse

It is common in the battlefield that rotating turret is always faster than rotating the tank body in order to do quick reaction. In this task, you will implement ***RotateTurret()*** function in order make turret always look at the cursor.

We use ***RayCast*** to shoot a ray from the camera to the ground, and find a hit point on the ground which can be used as a reference to guide the direction of the turret.

Before looking at the code, you should manually add a layer to the ***Ground*** gameobject by creating a new layer named 'Ground'. It is because when you import a Unity package, the customized layers will be missing.

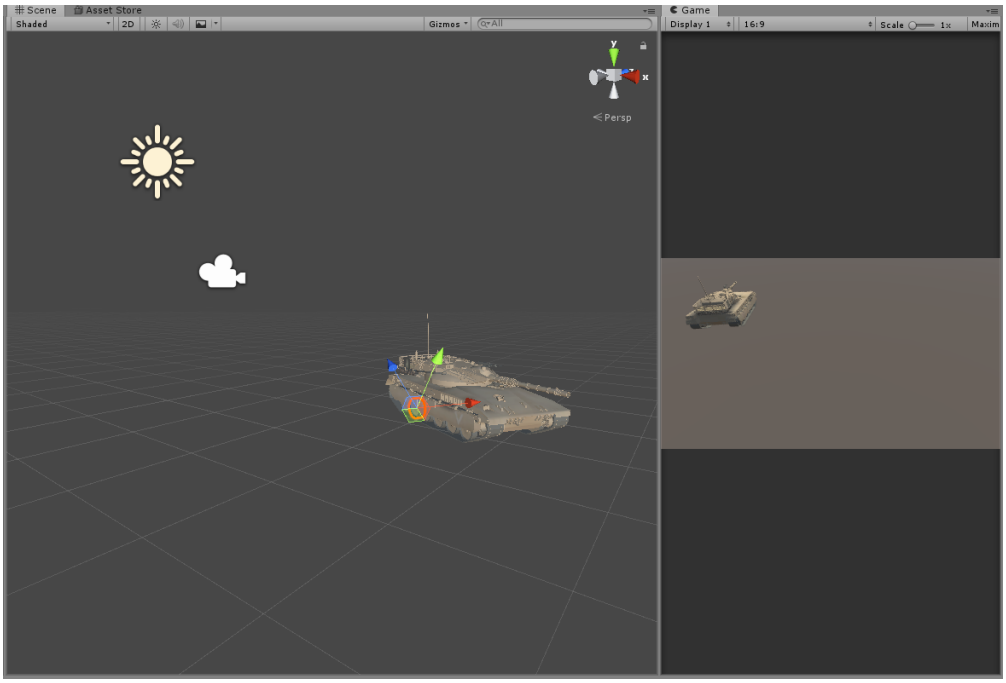


Figura 5: Scene view and game view.

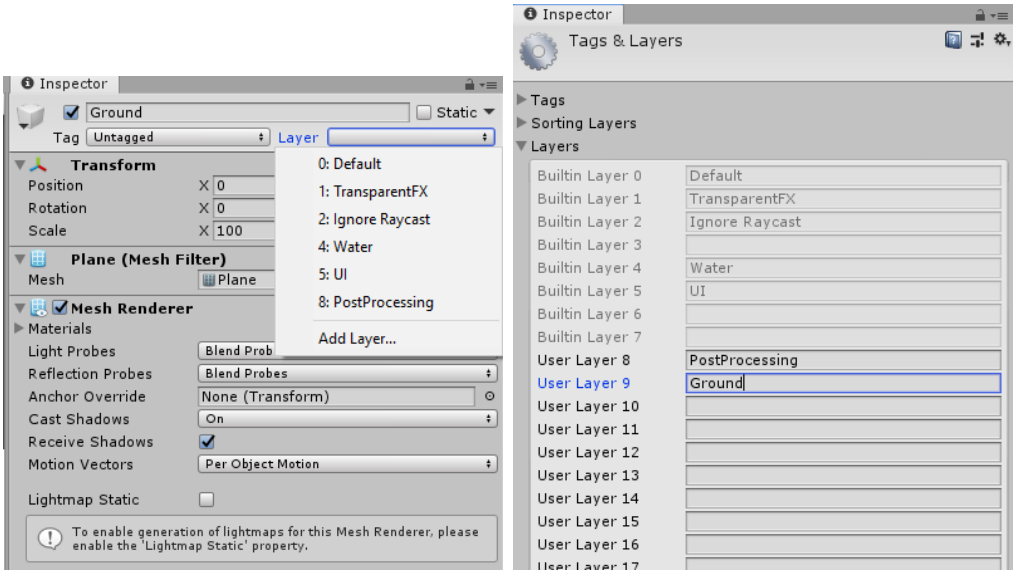


Figura 6: Add a new layer in the scene.

```
1  private void RotateTurret()
2      {
3          // Create a ray from the mouse cursor on screen in the direction of
       the camera.
4          Ray camRay = Camera.main.ScreenPointToRay(Input.mousePosition);
5
6          // Create a RaycastHit variable to store information about what was
       hit by the ray.
```

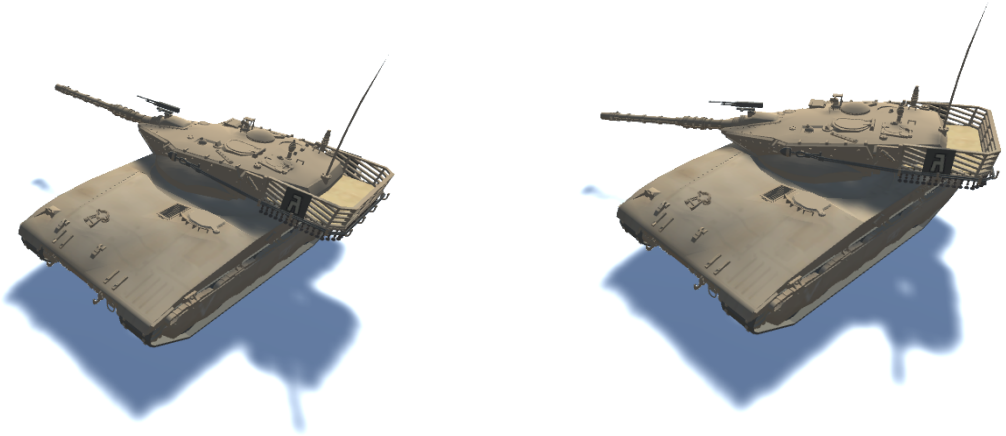
```
        RaycastHit floorHit;

        // Perform the raycast and if it hits something on the floor layer...
        if (Physics.Raycast(camRay, out floorHit, camRayLength, floorMask))
        {
            // Your code here.
        }
    }
```

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

Hints and tips:

- The turret should rotate with y value to be 0 as shown in Figure 7.
- You might think of the difference between rotating wheels and rotating turret by considering the difference between ***localRotation()*** and ***rotation()***.



(a) Right! The turret rotates only on 2D plane. (b) Wrong! An artifact happens with rotating freely on any axis.

Figura 7: The right and wrong cases of turret rotation.

3. Particle System

A particle system is a technique in game physics and computer graphics that simulates the interactions between a large number of small objects in order to model various types of phenomena, ranging from snowfall, rain and sparks, to solar systems and galaxies. In this section, you will make a simple particle system using Unity built-in particle system.

3.1. Fire Flame

This section will deal with another scene ***Fireflame*** Scene in the Scenes folder. Open the scene, you will find a bonfire without fire flame. In the scene, you will find an empty gameobject called ***Fireflame***. Add particle system by clicking ***Add Component*** in the inspect, and type to find Particle System.

Some of you might have the problem that the texture of the added particle system is missing as shown in the left of Figure 8. It can be fixed by following the steps: click the ***Renderer*** section of the particle system, click the Dot next to the ***Material*** field and double-click ***Default-Particle*** in the window that pops up.

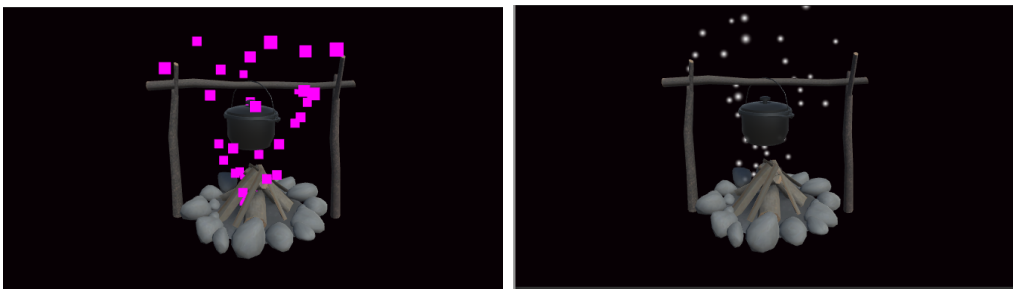


Figura 8: Particle system with texture missing (left) and corrections (right).

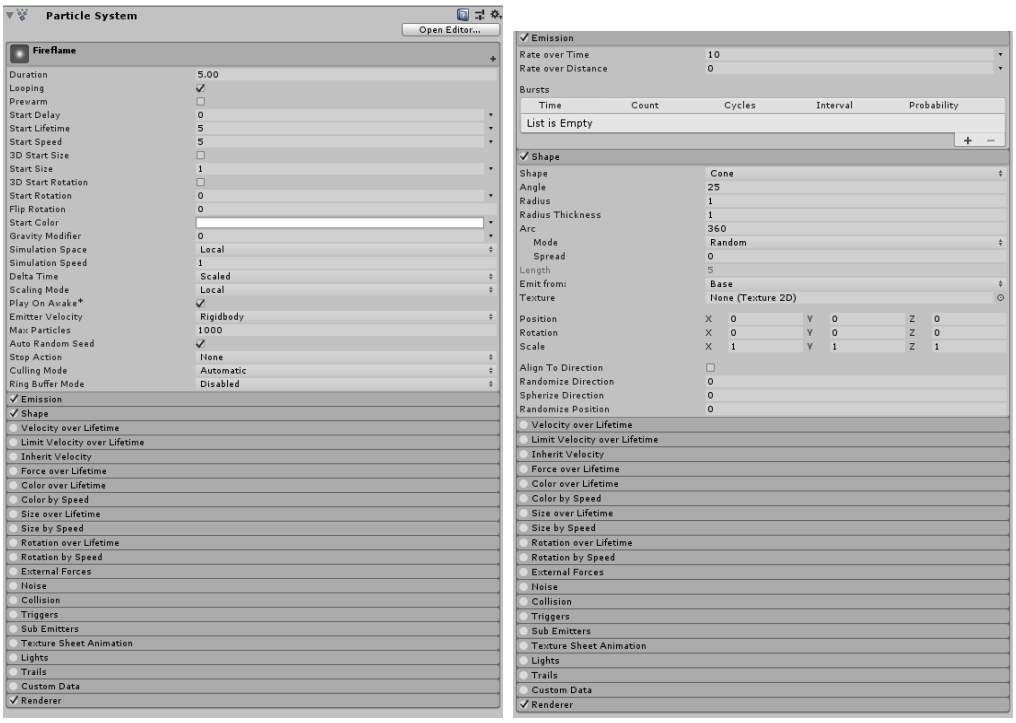


Figura 9: Modules in a partile system.

Take a look at the Inspector. You'll notice the particle system Component you added has several subsections. Each of these subsections is called a ***Module***. These Modules contain

the settings for the particle system. The Module expanded by default is known as the Main module as shown in the left of Figure 9. The most common particle settings live here:

- Duration: The length of time in seconds for the particle system to run.
- Looping: Repeatedly emit particles until the particle system stops. The cycle restarts once the Duration time is reached. The fire needs to burn continuously, so leave this enabled.
- Prewarm: Only used when Looping is enabled. The Particle System will act as if it's already completed a full cycle on start-up.
- Start Delay: The delay in seconds before the particle system starts emitting.
- Start Lifetime: The initial lifetime in seconds for the particles. The particle is destroyed after this elapsed time.
- Start Speed: The initial speed of the particles. The greater the speed of the particles, the more spread out they will be.

Figura 10: Fire flame under the pot.

The ***Emission*** and ***Shape*** are commonly used as well. You can play around with all the modules to get a nice fire. The parameter used in Figure 10 is as below:

- Looping checked
- Start Lifetime = 2
- Start Size = 3
- Emission –>Rater over Time = 15

- Shape –>Angle = 7
- Shape –>Radius = 0.2
- Renderer –>Material is set to be FireMaterial which can be found in Resources/Models/Fire/Materials.

3.2. Shooter

This section will deal with another scene ***ShellScene*** in the Scenes folder. After ShellScene is loaded, you will find a shooter and a target in the scene. The shooter is an empty gameobject with script ***ShellEmitter.cs*** attached to it. If left mouse key is clicked, the emitter will shoot out a shell towards the target, and holding left mouse key for a while will charge energy and fire the shell at a higher speed.

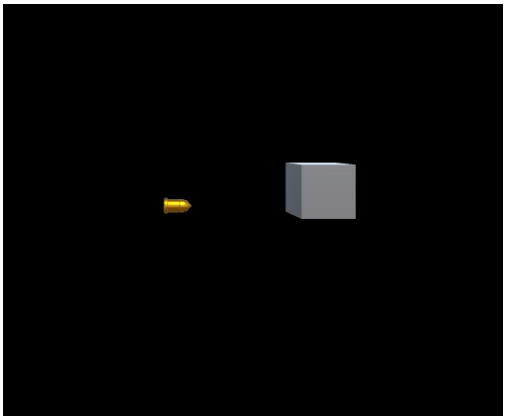


Figura 11: Fire a shell to the target.

We look into the ***ShellEmitter.cs*** to see how the emitter fire a shell. The key is the ***Fire()*** function as shown in Listing 5. It brings an important concept ***Prefab***[5] in Unity. The prefab acts as a template from which you can create new prefab instances in the scene. When you want to reuse a gameobject configured in a particular way – like a non-player character (NPC), prop or piece of scenery – in multiple places in your scene, or across multiple scenes in your project, you should convert it to a prefab. This is better than simply copying and pasting the gameobject, because the prefab system allows you to automatically keep all the copies in sync. For example, in this lab, the shell is made as prefab since the emitter will shoot multiple times.

We create a prefab by simply drag an gameobject in the scene and drop it to an empty place in the asset folder. And we use ***Instantiate()*** to instantiate an instance from the prefab.

```
1    private void Fire()
2    {
3        if (shellPrefab != null)
4        {
```

[5] https://docs.unity3d.com/Manual/Prefabs.html

```
5             // Create an instance of the shell and store a reference to it's
    rigidbody.
6             GameObject shellInstance = Instantiate(shellPrefab, transform.
    position, transform.rotation);
7
8             // Set the shell's velocity to the launch force in the fire
    position's forward direction.
9             shellInstance.GetComponent<Rigidbody>().velocity =
    m_CurrentLaunchForce * transform.forward;
10
11            // Set the shell's life time to be 3 second. After that, the shell
     will be destroyed.
12            Destroy(shellInstance, 3f);
13        }
14    }
```

Listing 5: Fire a shell

Refer to the previous sections about getting Input, we use

```
1 // 0 for left button, 1 for right button, 2 for the middle button
2 Input.GetMouseButton(0)
```

to get response from left mouse key. In this script, you will also find ***GetMouseButtonDown()*** and ***GetMouseButtonUp()*** which detect the response when the mouse key is push down and pop up.

3.3. Task

3.3.1. Fire the Target!

- Create a particle system to show the explosion effect. What parameters you used to create this particle system (default values should not be reported)?
- Make the particle system you just created to be a prefab. Drag and drop it to the ***Explosion.cs*** which is attached to the Target as shown in Figure 12. You should see something similar in Figure 13. Take screenshots of what you get (sequential images like Figure 13 would be good).

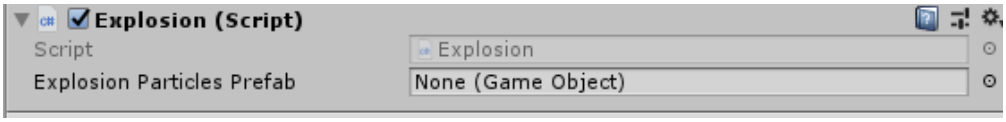


Figura 12: Place to drag and drop particle system prefab.

3.3.2. Create Your Scene

Create a new scene and import a free 3D model (find it online), such as .OBJ or .FBX. Tink of how to use particle system to add dynamic effects onto the model you imported. For example, Section 3.1 is created by adding a fire flame particle system on a bonfire model.

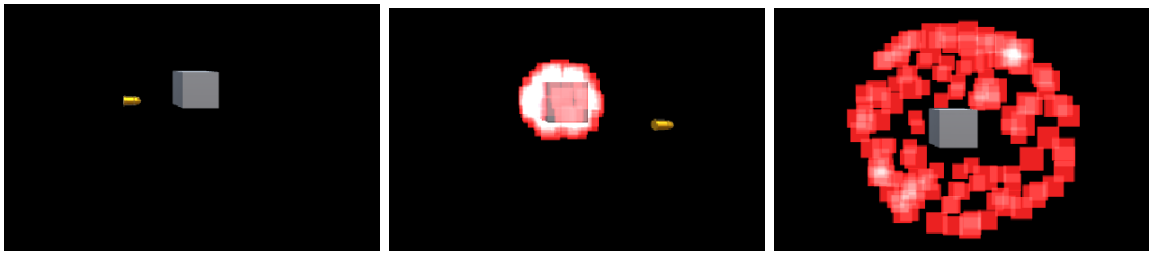


Figura 13: An explosion will shown when the shell hits the target.

You might think of use particle system to simulate rains, snow flare, splash, etc. Write down the process, keep record of the final image and also report related parameter you used.