Manual of
Universal Danmaku Engine
(alpha 1.0.0)

I. Introduction

Universal Danmaku Engine(a.k.a UDE) is a package of several scripts that helps to make bullet hell(Danmaku in Japanese) games with customizable patterns. It contains scripts that runs the game and some examples that help to understand how the engine works. UDE is made mainly for bullet hell games, but it can be used in any other games that handles some patterns(that's the reason why it is called UNIVERSAL Danmaku Engine).

The current version(alpha 1.0.0) of UDE provides these features.

- Scriptable stage pattern
- Scriptable enemy pattern
- Basic bullet, homing bullet
- Basic laser(straight and curve)
- Object pool for bullets
- Engine's own time scale, which is applied differently to enemies and players
- Transition helper similar to iTween but supports engine's time scale(WIP, only supports movement currently)
- Math library that helps calculations
- ECS and Job System(experimental and unstable)

The engine is made in Unity 2019.1.0f2. Thus it is recommended to use this package in 2019.1.0f2 or newer versions. Also, the engine needs following packages.

- Burst 1.1.2
- Collections preview – 0.1.1

- Entities preview – 0.1.1
- Jobs preview – 0.1.1
- Mathematics 1.1.0

Also, the engine is currently in alpha state, so it may be unstable. If you find some bugs, please report to me on e-mail(suboo0308@gmail.com).

II. Structure of the Engine

The engine consists of five kinds of objects

OBJECTS	DESCRIPTION	SCRIPTS
Bullets	Scripts for bullets. All bullets are child of 'UDEAbstractBullet' class.	• UDEAbstractBullet • UDEBaseBullet • UDEBulletECS(ECS) • UDEHomingBullet
Lasers	Scripts for lasers. All lasers are child of 'UDELaser' class.	• UDELaser • UDECurveLaser • UDEStraightLaser
Characters	Scripts for enemies and players. All characters are child of 'UDEBaseCharacter' class	• UDEBaseCharacter • UDEPlayer • UDEEnemy
Patterns	Scripts for shot patterns and stage patterns. Shot pattern determines how enemy shoots bullets. Stage pattern determines when to summon enemies	• UDEBaseShotPattern • UDEBaseStagePattern
Managers	Scripts that manages objects. All managers are singletons.	• UDESingleton • UDEBulletPool • UDEObjectManager • UDETime

Also, there are scripts that are not objects. They are components and systems of Unity ECS. ECS entities and systems are managed separately from 'MonoBehaviour' objects. I will address ECS of the engine at the last chapter of the manual.

For now, we will see how 'MonoBehaviour' side of the engine works.

- All bullets have a component that is a child of 'UDEAbstractBullet' class. This script contains methods that calculate bullet's trajectory.
- The bullets are summoned from 'UDEBulletPool'. Bullet pool finds a bullet to summon from already instantiated bullets. If there is no such bullet, UDEBulletPool creates new pool that contains that bullet.
- 'UDEBaseShotPattern' script get bullets from the bullet pool and initializes them. Initial positions and movements are determined by stage pattern script. When bullets are initialized, they are registered to 'UDEObjectManager'
- 'UDEObjectManager' calls methods of registered bullets to calculate their movements and move them. Object manager call these methods every FixedUpdate.
- Stage patterns are registered to 'UDEEnemy'. Enemy script contains patterns and their health. The enemy runs patterns in sequence. If the health of current pattern becomes below 0 or time runs out, enemy stops the pattern and runs next pattern. If the last pattern ends, enemy dies.
- 'UDEBaseStagePattern' script summons enemies and let them run their patterns at specific time and location. Also, stage pattern script can let enemies move if the movements are simple. For complex movements, it is better to handle in shot pattern of each enemy.

To sum up, UDEBaseStagePattern first runs and summons UDEEnemy. UDEEnemy runs UDEBaseShotPattern and UDEBaseShotPattern gets bullets from UDEBulletPool and initializes them. Then UDEBaseBullet objects appear in the game and move. All bullets are managed in UDEObjectManager.

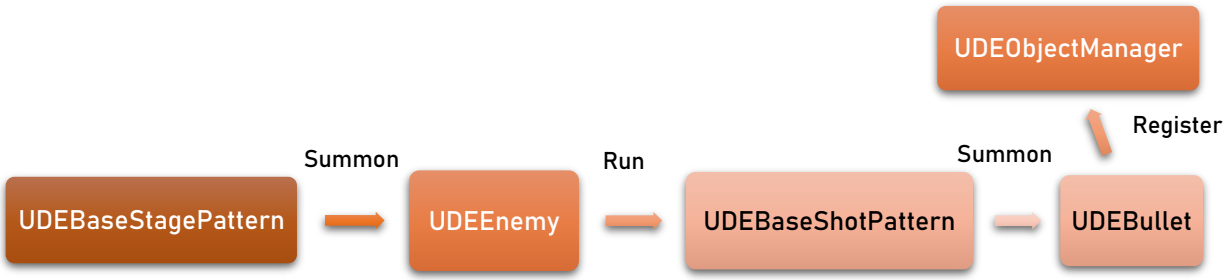

Figure 1 The structure of the engine

III. Getting started

First, create new 2D project. Then import the package.

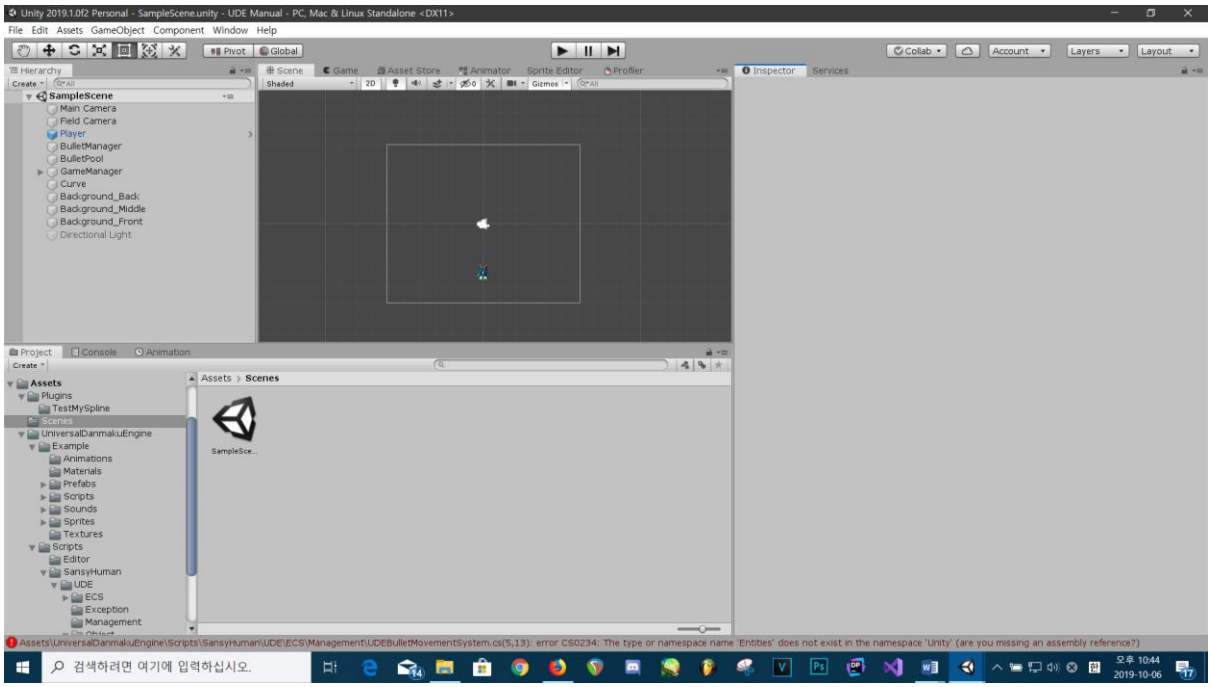

Figure 2 After importing the package

Files in package will be imported. If the editor throws errors like 'The type or namespace name '000' does not exist in the namespace 'XXX", it's because you didn't imported packages needed to the engine. Go to Window-Package Manager and check 'Show preview packages' in the Package Manager dialog box. Then import all packages you need(that are listed in introduction).

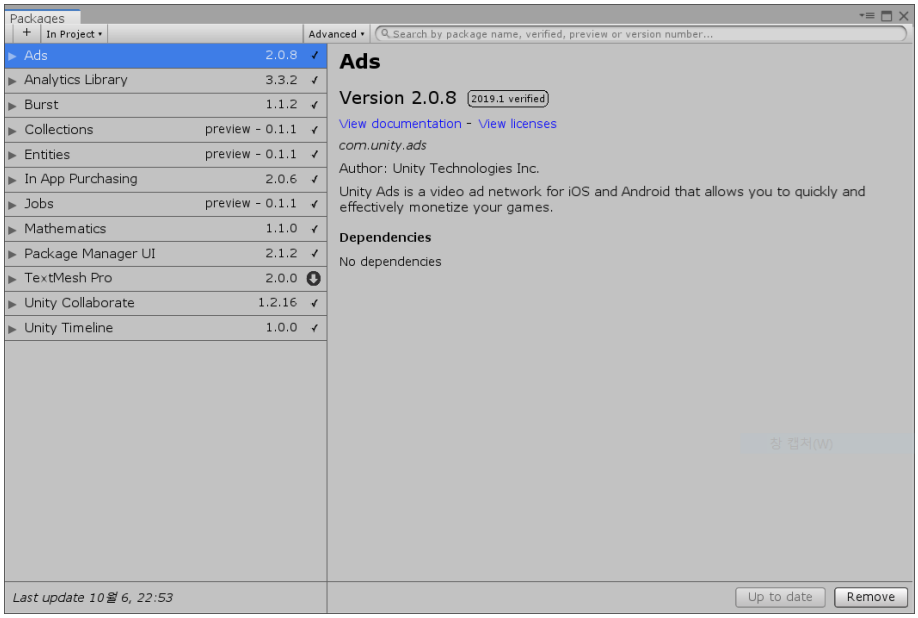

Figure 3 Importing packages

And go to Edit-Project Settings and set Api Compatibility Level in Player-Configuration tab to '.NET 4.x'.

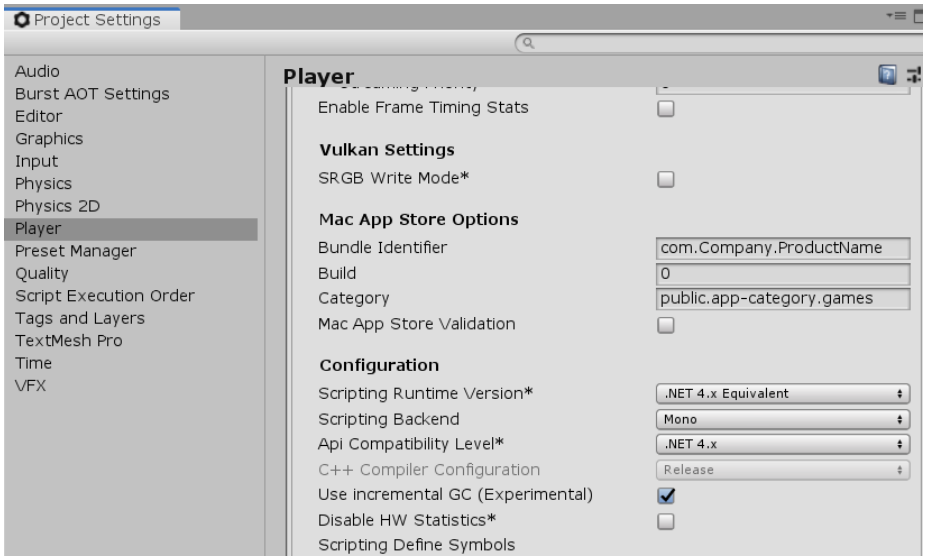

Figure 4 Project setting

Next, you need to add tags. Go to Project Settings-Tags and Layers tab and add these tags. The order is not important.

- Bullet
- Enemy
- Laser

If some of these tags are already exists, you don't need to add them.

You can, but may not, add layers. The engine will work without layers, but layer will reduce the amount of physics calculations. Add these layers in Project Settings-Tags and Layers tab.

- Player
- Enemy
- PlayerBullet
- EnemyBullet

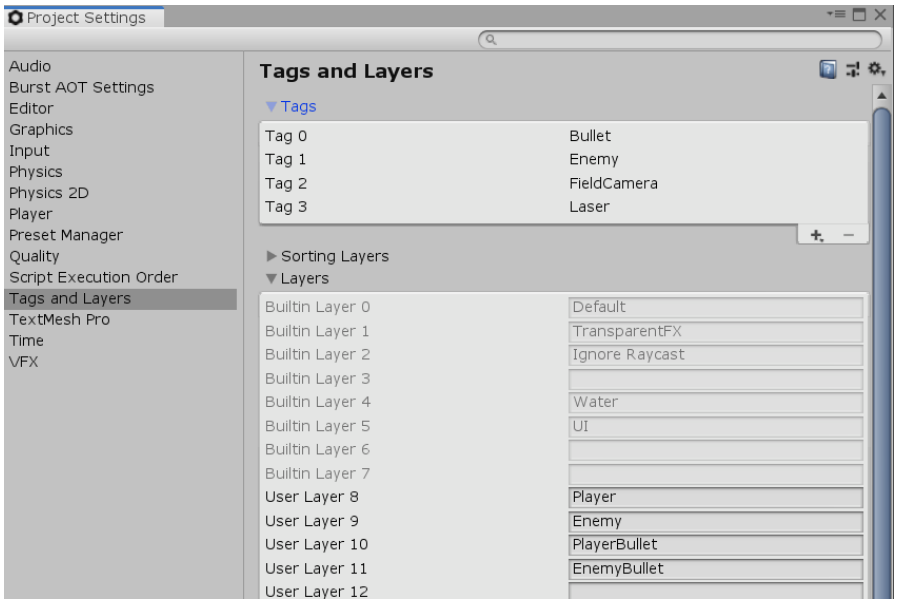

Figure 5 Tags and Layers settings

The player and player bullets will not collide. Also, Enemies and enemy bullets will also not collide. Of course objects in the same layer will not collide each other(if they collide in your game, you can set them collide). So to sum up,

- Player will not collide with other player(if there exists) and player bullets.
- Enemies will not collide with other enemies and enemy bullets.
- Bullets will not collide each other.

Thus in Project Settings-Physics 2D, set Layer Collision Matrix like below.

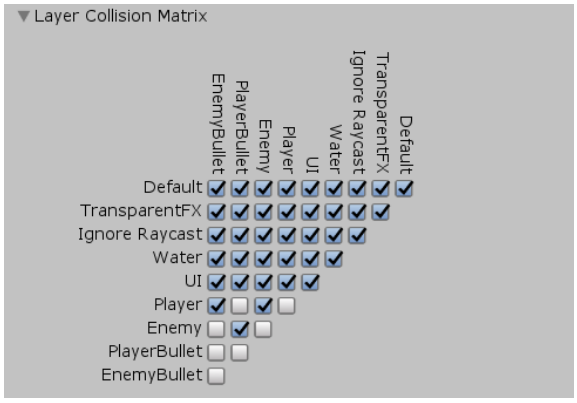

Figure 6 Layer Collision settings

If all settings are done, you can play example scene and it works perfectly.

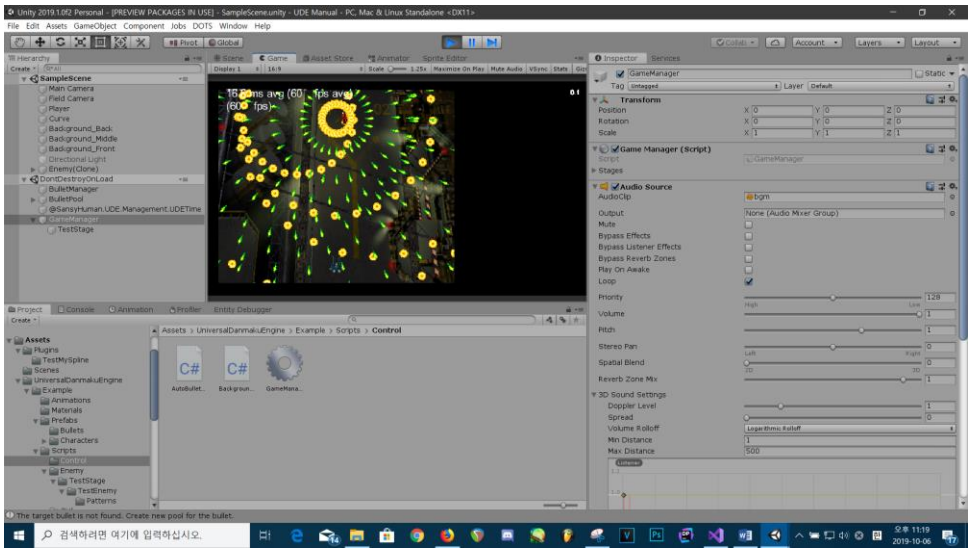

Figure 7 Example is running

IV. Making Bullets

Now we first make bullets. Bullet is the most basic element in bullet hell, so we make bullets first. We will use resources from unity3d-jp-tutorials/2d-shooting-game(https://github.com/unity3d-jp-tutorials/2d-shooting-game).

Before starting, make sure that the sprite of the bullet is oriented in horizontal right direction, in other words, the head of the bullet is on the right and tail is on the left on the horizontal line. That's because the engine uses polar coordinates that sets horizontal right to be 0 degree. The sprite in original tutorial from above link is oriented vertically, so you should turn the sprite using Photoshop or whatever program.

If you made your bullet sprites properly, then drag and drop the sprite to the Hierarchy of the project to make game object of that sprite. Name the bullet and drag the bullet object to prefab folder to make bullet prefab

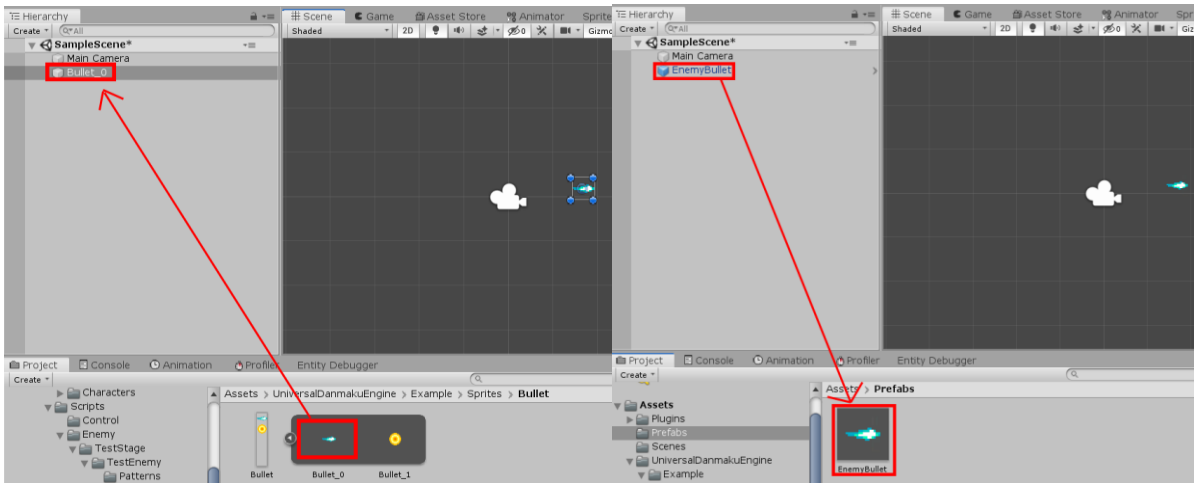

Figure 8 Making bullet prefabs

In this example, I made a bullet named 'EnemyBullet.' If you made a prefab, delete the bullet object in the Hierarchy and open the prefab.

The bullet should have a script that extends 'UDEAbstractBullet' abstract class. This class is the base class of all bullets. There are three bullet scripts that extend UDEAbstractBullet.

Scripts	Descriptions
UDEBaseBullet	Basic bullet script for generic bullets. The bullet moves as user's settings.
UDEHomingBullet	Bullet script for homing bullets. It is only used for player's bullets(since if enemies shoot homing bullets, it is very difficult to avoid).
UDEBulletECS	Basic bullet but for ECS(Not addressed in this chapter).

We will use the prefab for basic enemy bullet. So we add a script UDEBaseBullet to the prefab.

You can set two properties through the inspector.

Properties	Descriptions
Damage	The damage that bullet deals. In default settings, enemy bullets deals always 1 damage to player whatever damage you set. So basically this property is only used for player bullets.
Summon Time	Time amount that the bullet is in summon state. In summon state, the bullet blurs(actually it just activates halo and disables sprite renderer and collider). It is similar to some bullets in Touhou Project games.

If you want enemy bullets deal damage as you set but not always 1, you should revise some codes in UDEPlayer script(the script is in UniversalDanmakuEngine/Scripts/SansyHuman/UDE/Object/Character

folder).

Bullet collision to player is handled in `UDEPlayer.OnTriggerStay2D`.

```
        private void OnTriggerStay2D(Collider2D collision)
        {
            if (invincible)
                return;

            if (collision.CompareTag("Enemy"))
                StartCoroutine(DamageSelf(1));
            else if (collision.CompareTag("Bullet"))
            {
                UDEAbstractBullet bullet =
collision.GetComponent<UDEAbstractBullet>();
                if (bullet != null && bullet.OriginCharacter is UDEEnemy)
                {
                    UDEBulletPool.Instance.ReleaseBullet(bullet);
                    StartCoroutine(DamageSelf(1));
                }
            }
            else if (collision.CompareTag("Laser"))
            {
                UDELaser laser = collision.GetComponent<UDELaser>();
                if (laser != null && laser.OriginCharacter is UDEEnemy)
                {
                    StartCoroutine(DamageSelf(1));
                }
            }
        }

        private IEnumerator DamageSelf(float damage)
        {
            health -= damage;
            invincible = true;
            SpriteRenderer renderer = self.GetComponent<SpriteRenderer>();
            Color col = renderer.color;
            col.a = 0.5f;
            renderer.color = col;
            yield return
StartCoroutine(UDETime.Instance.WaitForScaledSeconds(3f,
UDETime.TimeScale.PLAYER));
            col.a = 1f;
            renderer.color = col;
            invincible = false;
        }
```

So the code will reduces 1 health of the player when the player hits to enemies, bullets, or lasers. If you want to deal damage set to bullet, you can

write this code.

```
        else if (collision.CompareTag("Bullet"))
        {
            UDEAbstractBullet bullet =
collision.GetComponent<UDEAbstractBullet>();
            if (bullet != null && bullet.OriginCharacter is UDEEnemy)
            {
                UDEBulletPool.Instance.ReleaseBullet(bullet);
                StartCoroutine(DamageSelf(bullet.Damage));
            }
        }
```

If you added a script, then you have to add a Rigidbody 2D and Collider 2D. Add a Rigidbody 2D and any Collider 2D you want.

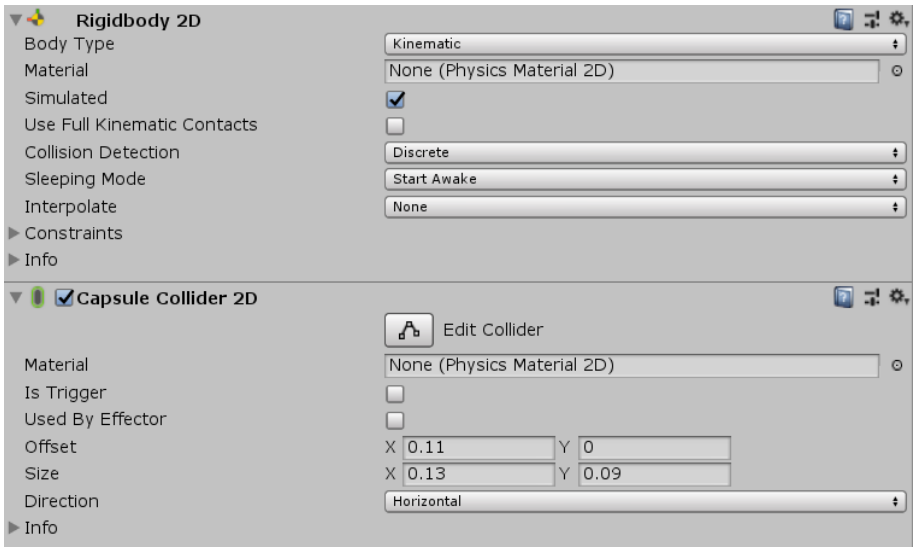

Figure 9 RigidBody and Collider

You should set the body type of the Rigidbody to Kinematic so that collisions between bullets are ignored. Thus all bullets have kinematic collider. Then edit the collider's shape as you want. In bullet hell games, collider is usually a little bit smaller than the sprite.

Finally, set the tag of the bullet to 'Bullet'. If you set layers too, then set

the layer to ‘EnemyBullet’ if the bullet is for enemies, or to ‘PlayerBullet’ if the bullet is for the player. In this case, I set the bullet’s layer to EnemyBullet since it is for enemies.

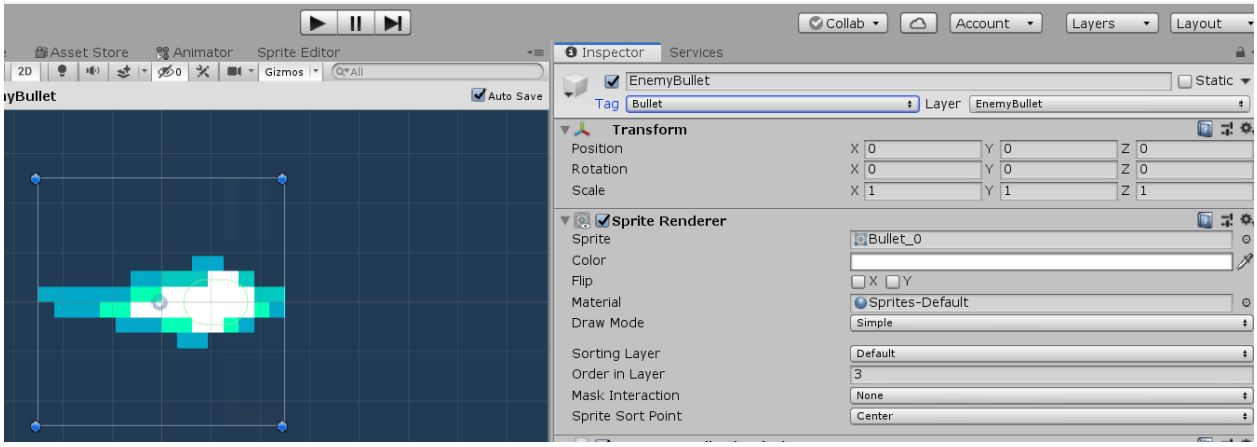

Figure 10 After setting all things

If you want bullet to have summon state(effect like in Touhou, that bullet blurs for few seconds when they first summoned and after that the clear figure appears), you just can add an Halo to the bullet. The size and color of the halo can be set as your own. The Summon Time property is only in effect if there is a halo to the bullet; in other words, it will be ignored if there is no halo.

When the bullet is in summon state, Sprite Renderer and Collider will be disabled and Halo will be enabled. During the summon state, bullet will not move. After Summon Time passes, Halo will be disabled and Renderer and Collider will be enabled again. Also, bullet will start to move.

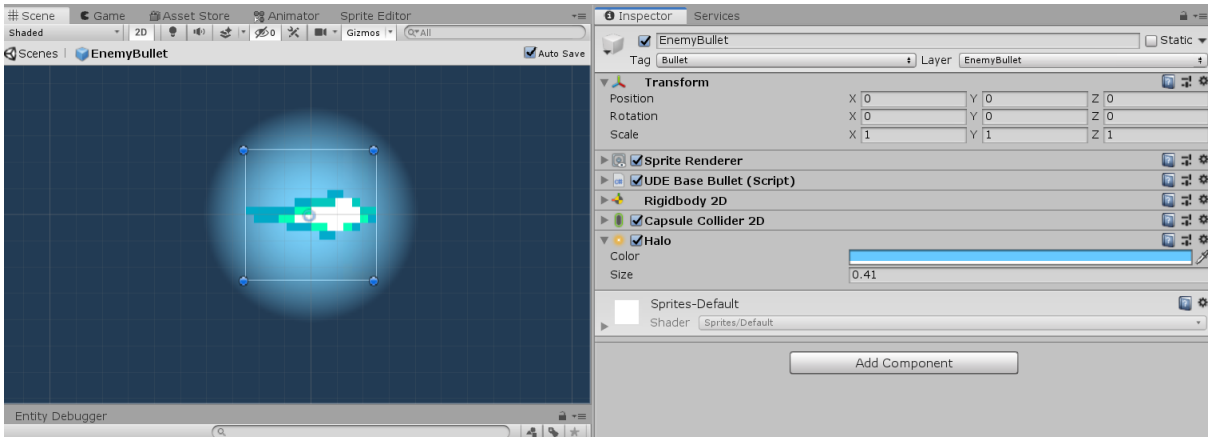

Figure 11 After adding a halo

It is done! You successfully made a bullet. But it will do nothing if you add it to the hierarchy. You have to initialize the bullet in the 'UDEBaseShotPattern' script. Next chapter will see that.

V. Making Shot Patterns

If you made bullets, it's time to make a shot pattern. All shot patterns are managed by a script that extends `UDEBaseShotPattern` in namespace '`SansyHuman.UDE.Pattern`'.

To start, make a script that will extend `UDEBaseShotPattern`. I named it '`TestShotPattern`'. You have to implement one abstract method.

```
protected abstract IEnumerator ShotPattern();
```

This method will return coroutine used in the class that will summon bullets in specific pattern. You can write your own pattern by implementing this method. Let's see an implementation which is also in API document.

```
using System.Collections;
using UnityEngine;
using SansyHuman.UDE.Pattern;
using SansyHuman.UDE.Object;
using SansyHuman.UDE.Management;
using SansyHuman.UDE.Util.Builder;
using SansyHuman.UDE.Util.Math;

public class TestShotPattern : UDEBaseShotPattern
{
    [SerializeField]
    private int NumberOfBullets = 10;
    [SerializeField]
    private float BulletSpeed = 2f;

    protected override IEnumerator ShotPattern()
    {
        float AngleDeg = 360f / NumberOfBullets;
        float AngleRef = 0f;
        float RefOmega = 0f;

        while (true)
        {
            for (int i = 0; i < NumberOfBullets; i++)
            {
                UDEAbstractBullet bullet =
UDEBulletPool.Instance.GetBullet(patternBullets[0]);
                UDEBulletMovement movement =
UDEPolarMovementBuilder.Create().RadialSpeed(BulletSpeed).InitialAngle(i *
AngleDeg + AngleRef).Build();
                Vector2 origin = originEnemy.transform.position;
                var formLocTuple = UDEMath.Polar2Cartesian(0.7f,
movement.angle);
                Vector2 formLocation = new Vector2(formLocTuple.x,
formLocTuple.y) + origin;
                bullet.Initialize(formLocation, origin, 0, originEnemy,
this, movement);
            }
            AngleRef += RefOmega;
            RefOmega += 0.2f;
            if (RefOmega >= 360)
                RefOmega -= 360;
            if (AngleRef >= 360)
                AngleRef -= 360;
            yield return
StartCoroutine(UDETime.Instance.WaitForScaledSeconds(0.07f,
UDETime.TimeScale.ENEMY));
        }
    }
}
```

Let's look at the code more detail.

```
UDEAbstractBullet bullet =
UDEBulletPool.Instance.GetBullet(patternBullets[0]);
```

UDE provides object pool for bullets named 'UDEBulletPool' since there will be hundreds or thousands of bullets to make. UDEBulletPool is a singleton and can access to the instance by a property 'Instance'. Actually, all singleton extending 'UDESingleton<T>' have a property 'Instance'.

You should call method UDEBulletPool.GetBullet(UDEAbstractBullet) to get a bullet from the pool. If you want to remove bullet, then call method UDEBulletPool.ReleaseBullet(UDEAbstractBullet)(but in most case, the bullet will automatically removed when they are out of the screen, so there will be few situation to call the method directly.)

In UDEBaseShotPattern, a list of UDEAbstractBullet, patternBullets, is defined. You should register bullet prefabs to the list in the inspector.

Thus this code will get the copy of the first element in the patternBullets list from the object pool.

```
UDEBulletMovement movement =
UDEPolarMovementBuilder.Create().RadialSpeed(BulletSpeed).InitialAngle(i *
AngleDeg + AngleRef).Build();
```

UDEBulletMovement is a struct that defines the movement of the bullet. There are three movement mode; Cartesian, CartesianPolar, and Polar.

- In Cartesian mode, velocity and acceleration are described as 2-dimensional vectors.
- In CartesianPolar mode, velocity is described with speed and direction. Acceleration is described with tangential and normal acceleration.
- In Polar mode, velocity is described with radial and angular speed. Acceleration is described with radial and angular acceleration.

You can directly instantiate UDEBulletMovement through basic constructor, but there are dozens of parameters to put. So it is recommended to use builder class. There are builder classes for each movement mode. In this case, I used PolarMovement builder.

As you can see, the movement's radial speed is `BulletSpeed` and initial angle in polar coordinate is `i * AngleDeg + AngleRef`. Thus the trajectory of the bullet will be $r = (\text{BulletSpeed})t, \theta = (\text{AngleDeg})i + (\text{AngleRef})$, thus a straight line(t is a time from the initialization.)

In this example, you created only one bullet movement, but you can create several bullet movements so that each movement applies different time. When you do that, you should set start time of each movement so that the bullet automatically change its movement as the time passes.

(Or you can forcefully change movement by using a method `UDEBaseBullet.ForceMoveToPhase(int)`). For more details, please read API document.

```
Vector2 origin = originEnemy.transform.position;
var formLocTuple = UDEMath.Polar2Cartesian(0.7f, movement.angle);
Vector2 formLocation = new Vector2(formLocTuple.x, formLocTuple.y) +
origin;
```

In `UDEBaseShotPattern`, there is a `UDEEnemy` field named '`originEnemy`'. When `UDEEnemy` script runs `UDEBaseShotPattern` script, it automatically registers itself to `originEnemy` field. So `originEnemy.transform.position` is a current position of the enemy who is running the pattern. In this example, the position of the enemy is set to be a origin of the bullet's polar coordinate in world space(In Polar mode, bullet calculates the radius and angle in polar coordinate using this origin.)

Also, I set the position that the bullet will appear. The bullet will appear in the distance of 0.7 from the origin in the direction of the initial angle of the bullet. `UDEMath` contains several functions that helps calculations. `UDEMath.Polar2Cartesian(float r, float deg)` returns cartesian coordinate in distance r from the origin and angle deg from the right horizon. It returns cartesian coordinate in tuple `(float x, float y)`, so you have to transform it to vector.

```
bullet.Initialize(formLocation, origin, 0, originEnemy, this, movement);
```

You should initialize the bullet to make it to move. You can initialize the bullet with a method `UDEAbstractBullet.Initialize`.

```
public abstract void Initialize(Vector2 initPos,
                                Vector2 origin,
                                float initRotation,
                                UDEBaseCharacter originCharacter,
                                UDEBaseShotPattern originShotPattern,
                                UDEBulletMovement movement,
                                bool setOriginToCharacter = false,
                                bool loop = false);
```

PARAMETERS	DESCRIPTIONS
initPos	Initial position of the bullet in world space.
origin	Origin of the bullet in polar coordinate.
initRotation	Initial rotation angle of the bullet. It is used when the first bullet movement's field 'faceToMovingDirection' is false. Otherwise, the bullet will rotate automatically to the direction of the movement.
originCharacter	Character who shot the bullet.
originShotPattern	Pattern who summoned the bullet. If it is a player bullet, it can be null.
movement	Movement of the bullet. If the bullet has several movement in order, then put an array of movements instead.
setOriginToCharacter	Sets whether set the origin of the polar coordinate of the bullet to the position of the origin character automatically. Default value is false.
loop	Sets whether loop the movements; in other words, whether go back to first movement when the last movement ends. Default value is false.

If you set 'setOriginToCharacter' true, then the bullet will follow the character if it's in Polar mode. And if you set 'loop' true, the bullet will go back

to the first movement when the last movement ends(in that case, the last movement should have a field 'hasEndTime' true.

```
yield return StartCoroutine(UDETime.Instance.WaitForScaledSeconds(0.07f,
UDETime.TimeScale.ENEMY));
```

UDETime is a singleton that controls time scale of enemies and the player. Thus the instance can be accessed through a property Instance(as other singleton classes do). UDETime.WaitForScaledSeconds(float time, UDETime.TimeScale scale) method returns coroutine that pauses time under the time scale. There are three types of time scale.

VALUES	DESCRIPTIONS
UDETime.TimeScale.ENEMY	Time scale for enemies.
UDETime.TimeScale.PLAYER	Time scale for the player.
UDETime.TimeScale.UNSCALED	Unscaled time; thus 1.

Each time scale can be set through the inspector or properties named UDETime.EnemyTimeScale and UDETime.PlayerTimeScale. Thus the above code will pause the pattern for 0.07 seconds in enemies' time scale every cycle of the pattern.

To sum up,

1. Get copy of bullet by passing prefab to UDEBulletPool using a method UDEBulletPool.Instance.GetBullet(UDEAbstractBullet).
2. Create bullet's movements using builders.
3. Set origin in polar coordinate and initial position of the bullet with help of several functions in UDEMath.
4. Initialize bullet using a method UDEAbstractBullet.Initialize(…).
5. Pause the pattern using UDETime.Instance.WaitForScaledSeconds(…).

If you made a pattern script, add an empty game object to the hierarchy and name it to the script's name. Then add the script to the object and make object to prefab.

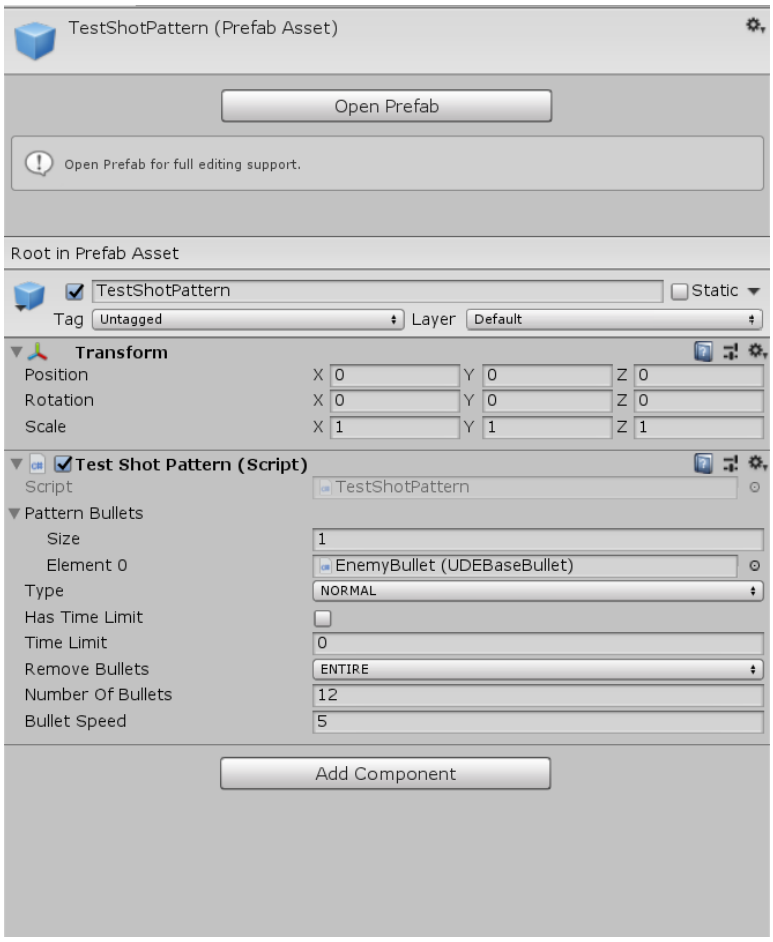

Figure 11 Pattern prefab

Then set the properties of the pattern.

PROPERTIES	DESCRIPTIONS
Pattern Bullets	Bullets that are used in the pattern. Must contain children of `UDEAbstractBullet` as its component

Type	Type of the pattern. There are two types. ● NORMAL: General pattern ● SPECIAL: Special pattern If you are not going to implement some logic that works differently between normal and special patterns, this property is meaningless.
Has Time Limit	Sets whether the pattern will automatically end after some time even if the health of the enemy doesn't go below zero.
Time Limit	If 'Has Time Limit' is true, time limit of the pattern in seconds.
Remove Bullets	Sets whether remove bullets when the pattern ends. There are three options. ● NONE: Do not remove bullets. ● PATTERN: Remove bullets shot by the pattern. ● ENTIRE: Remove all bullets in the game.

The last two properties are user-defined properties in the custom pattern script.

VI. Making Enemy

When you made patterns, you should make enemies who will run those patterns. All enemy has `UDEEnemy` as its component.

First, make a prefab of enemy. It can have some animations. Then add a `Rigidbody2D`, a `Collider2D`, and an `UDEEnemy` script. Set the body type as 'Kinematic' and set collider to trigger. And set the score on death(this value currently does nothing, but you can add some codes to add points when the enemy dies) and enemy type. There are two types of enemy; Invincible, and Destroyable. The enemy can be killed when it is destroyable. Finally, set the tag of the prefab to 'Enemy' and set layer to 'Enemy', if you added layers.

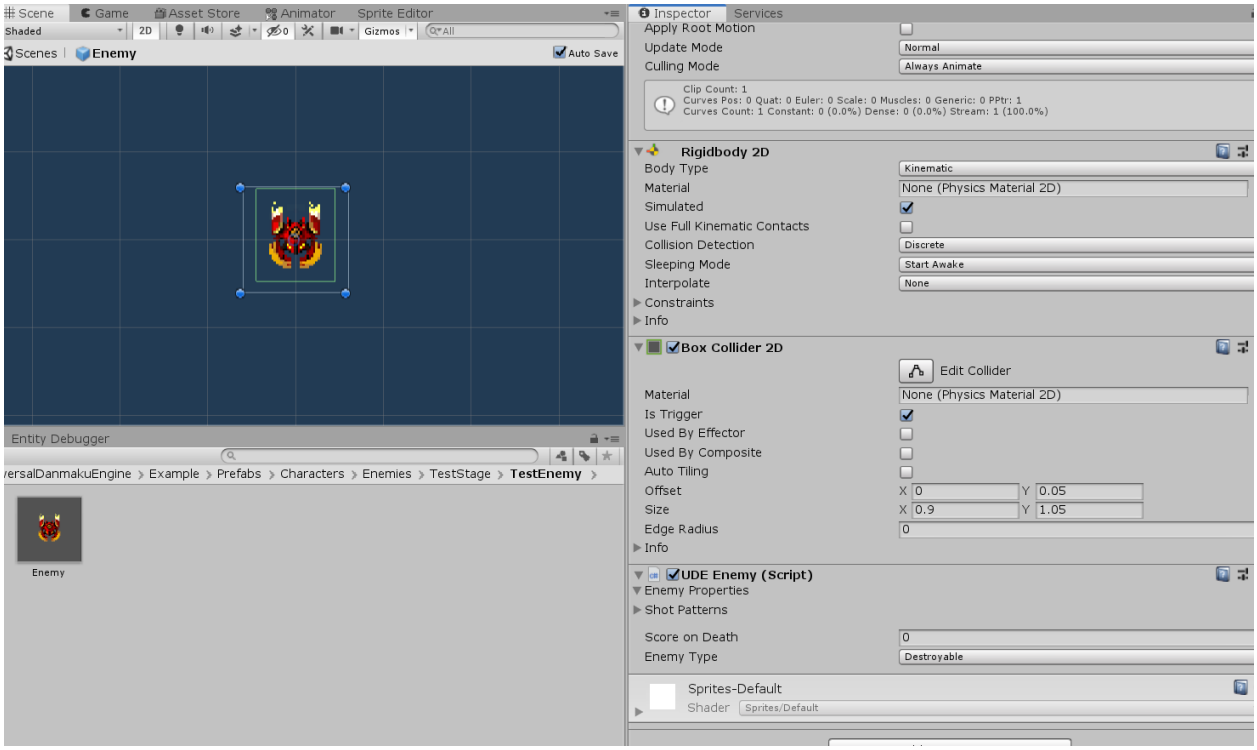

Figure 12 Enemy prefab

In this state, the hitbox that the enemy will take and deal damage is same. When the player's bullet is in the trigger, it will damage the enemy. And if the player is in the trigger, the player will get damage.

But you would want to make hitboxes of two different(for example, the range that the player will get damage by enemy can be smaller than the range

that the enemy will get damage by player's bullets). In this case, you can add an empty game object as a child of the enemy and add another collider.

First, set the tag of the enemy to 'Untagged'. Then add an empty game object as a child of the enemy. Then set the tag of the game object to 'Enemy' and add a collider.

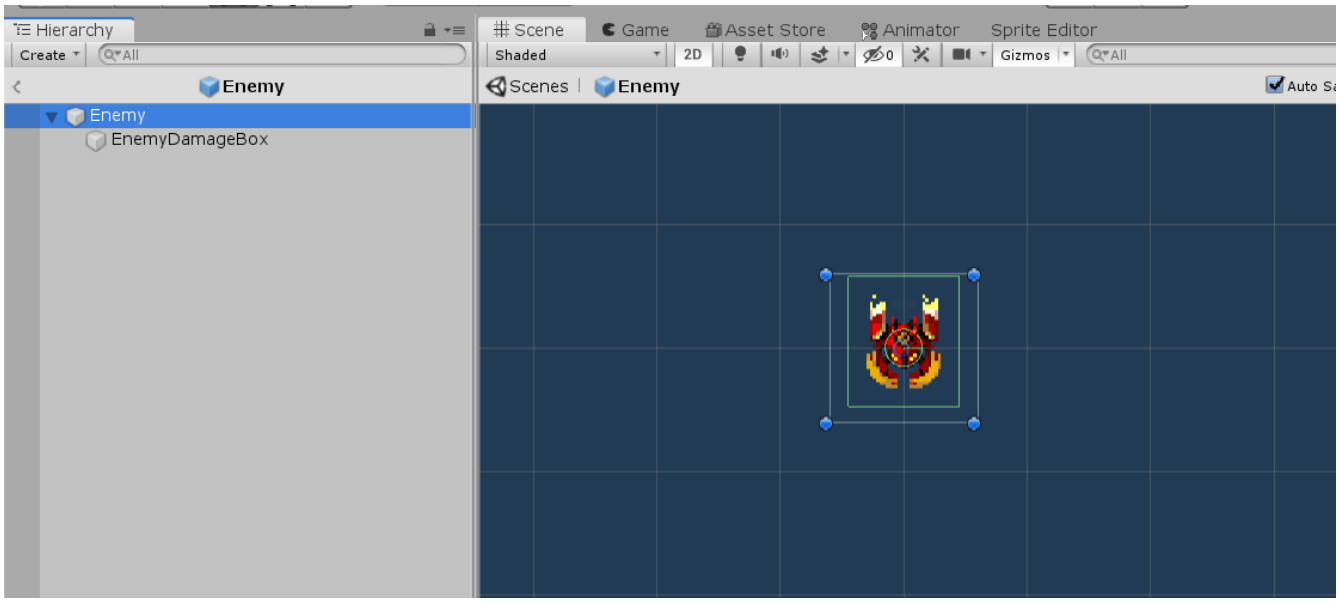

Figure 12 Enemy prefab

In this example, I made another smaller circle collider. The player will get damage if it collide with a circle collider, but not damaged when the player is in the larger box collider. But the enemy will get damage when player's bullet is in the box collider.

Then add shot pattern prefabs that the enemy will run as its children. And set the number of patterns in Shot Patterns property in UDEEnemy script. Then drag the shot pattern prefabs you added as children of the enemy and set health of each pattern. The pattern will be executed in sequence.

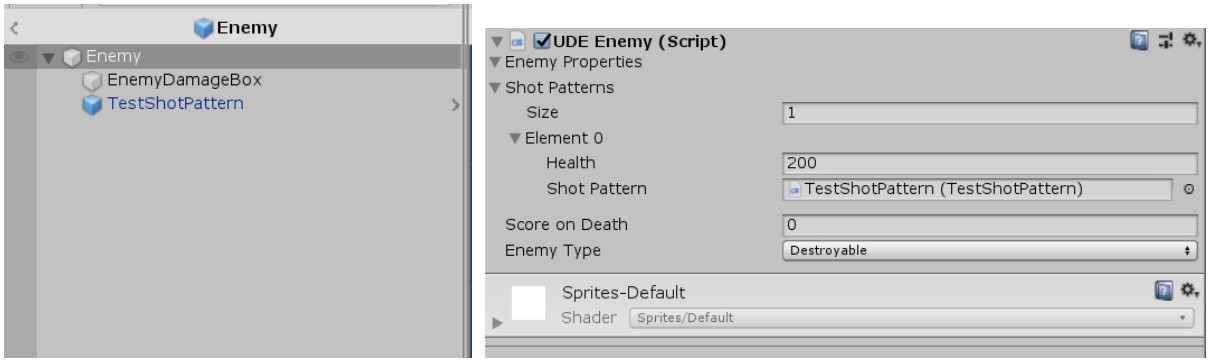

Figure 13 Setting the shot pattern of the enemy

You should not drag the original prefab of the pattern in the Shot Patterns property! It will cause the error. You should put the copy of the shot pattern in the children of the enemy.

You successfully made an enemy! But it will do nothing currently. You have to add stage pattern that will summon and initialize the enemy. We will take a look at that in the next chapter.

VII. Making Stage Pattern

We need to make stage pattern script to summon enemies. All stage patterns extends UDEBaseStagePattern class in the namespace SansyHuman.UDE.Pattern.

First, create a script file. In this example, I named it 'TestStagePattern'. Then let the script to extend UDEBaseStagePattern. In this script, enemy is summoned at the upper part of the screen and wait until the enemy dies.

```
using System.Collections;
using UnityEngine;
using SansyHuman.UDE.Pattern;
using SansyHuman.UDE.Util;
using SansyHuman.UDE.Object;
using SansyHuman.UDE.Management;

public class TestStagePattern : UDEBaseStagePattern
{
    protected override IEnumerator StagePattern()
    {
        while (true)
        {
            UDEEnemy enemy = SummonEnemy(enemies[0]);
            enemy.transform.position = Camera.main.Viewport-
ToWorldPoint(new Vector3(0.5f, 1.15f, 0f));
            enemy.CanBeDamaged = false;

            Vector2 dest = Camera.main.ViewportToWorldPoint(new Vec-
tor3(0.5f, 0.85f, 0f));
            var result = UDETransitionHelper.MoveTo(enemy.gameObject,
                                    dest,
                                    1.3f,
                                    UDETransitionHelper.easeOutCubic,
                                    UDETime.TimeScale.ENEMY,
                                    true);
            yield return new WaitUntil(() => result.EndTransition);

            enemy.CanBeDamaged = true;
            enemy.Initialize();
            yield return new WaitWhile(() => enemy.Alive);
        }
    }
}
```

Let's see more details of the code.

```
protected override IEnumerator StagePattern()
```

UDEBaseStagePattern class has an abstract method named StagePattern that returns coroutine of the stage pattern. The pattern starts and stops in StartStage and StopStage methods in the same class.

```
UDEEnemy enemy = SummonEnemy(enemies[0]);
```

SummonEnemy method is defined in UDEBaseStagePattern class. It takes UDEEnemy instance of the enemy prefab and returns the copy of the enemy. enemies is a list of UDEEnemy.

Actually, there are three lists of enemies; enemies, subBoss, and boss. enemies takes normal enemy characters. subBoss and boss take boss characters. But in this example, I only used enemies. If you have sub-boss or boss character in the stage, you can use subBoss and boss lists. All of these lists can be initialized in the inspector.

```
var result = UDETransitionHelper.MoveTo(enemy.gameObject,
                                        dest,
                                        1.3f,
                                        UDETransitionHelper.easeOutCubic,
                                        UDETime.TimeScale.ENEMY,
                                        true);
yield return new WaitUntil(() => result.EndTransition);
```

UDETransitionHelper is a class to help animate game objects. It is similar to the famous asset 'iTween', but UDETransitionHelper supports the time scale in UDE engine.(The current version of the UDE, 1.0.0 alpha, only supports movement, but it will support other properties in upcoming updates)

There are several methods for moving game objects. In this code, I used UDETransitionHelper.MoveTo() method.

```
public static TransitionResult MoveTo(GameObject target,
                                       Vector2 dest,
                                       float duration,
                                       UDEMath.TimeFunction easeFunc,
                                       UDETime.TimeScale timeScale,
                                       bool isPhysics)
```

PARAMETERS	DESCRIPTIONS
target	Game object to animate.
dest	Final position of the game object.
duration	Duration of the transition.
easeFunc	Ease function of the transition. There are several ease functions defined in UDETransitionHelper class, but you can also define a new ease function. The ease function is a delegate of type UDEMath.TimeFunction, which takes a float and returns a float. Ease function should start from 0 and end to 1 (mathematically, $f(0) = 0$ and $f(1) = 1$.)
timeScale	Time scale to use.
isPhysics	If true, the transition is handled in FixedUpdate. If false, the transition if handled in Update.

Thus the enemy will move to the dest for 1.3 seconds with ease type of easeOutCubic and enemy time scale. Also, the transition will be handled in FixedUpdate.

The method returns an instance of type TransitionResult. It is a class that holds a state of the transition. If EndTransition property of the instance is true, the transition have ended. Thus the coroutine stops until the transition ends.

```
enemy.Initialize();
yield return new WaitWhile(() => enemy.Alive);
```

To let enemy start its patterns, you have to call `Initialize()` method. This method will set enemy to be alive and start the shot patterns of the enemy. Alive property of the enemy indicated whether the enemy's last shot pattern ends(in other words, whether the last pattern's health becomes below 0). Thus the coroutine will be paused until the enemy is dead. After the enemy dies, a new enemy will be summoned again.

Now, make an empty game object and add the stage pattern script. Then add the enemy prefab from the previous chapter in the 'Enemies' list in inspector. Then add the following code in the stage pattern script.

```
private void Start()
{
    StartStage();
}
```

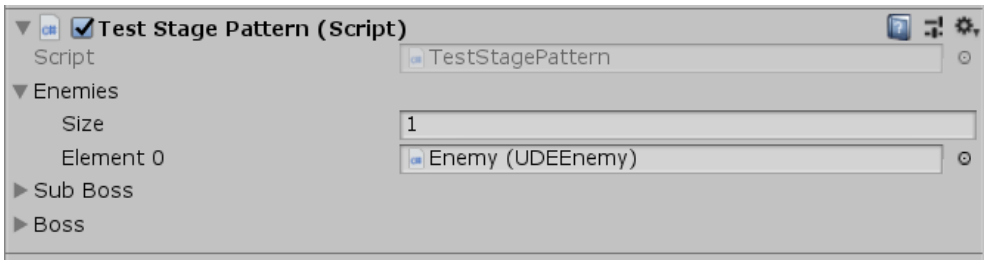

Figure 14 Setting the stage pattern

`StartStage()` method is defined in `UDEBaseStagePattern`. It starts the stage pattern. If you want to pause the stage, call `StopStage()` method. In this state, if you again call `StartStage()`, it will start from the paused part of the stage. If you want to restart the whole stage, call `ResetStage()` before calling `StartStage()`.

If you run the game, you will see the enemy appears from the top of the screen and shoots bullets.

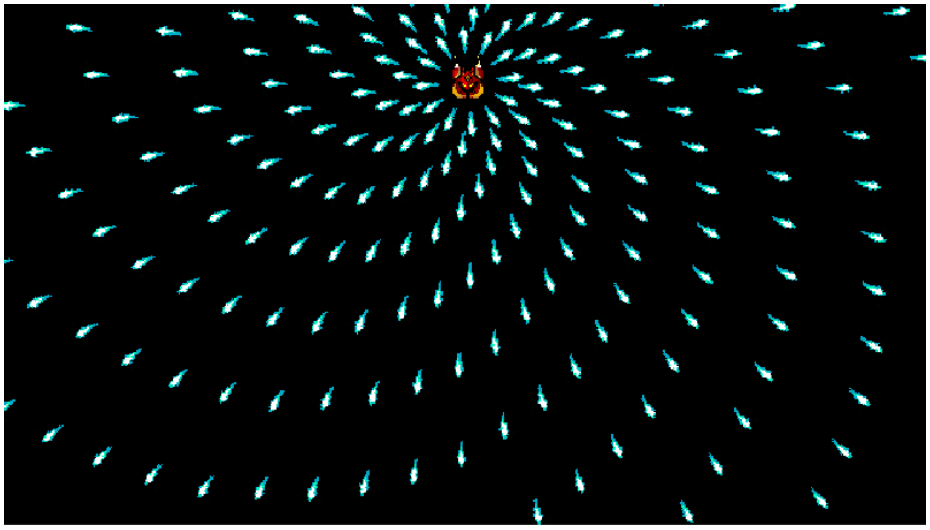

Figure 15 Enemy is shooting bullets

But in this state, bullets outside of the camera are not automatically removed. So you should add some scripts that removes bullets. When you remove the bullet, use `ReleaseBullet(UDEAbstractBullet)` method. In this example, I added a trigger to the main camera whose size is larger than the size of the camera and added the script to the main camera.

```
using UnityEngine;
using SansyHuman.UDE.Object;
using SansyHuman.UDE.Management;

public class AutoBulletRelease : MonoBehaviour
{
    private void OnTriggerExit2D(Collider2D collision)
    {
        if (collision.CompareTag("Bullet"))
        {
            UDEAbstractBullet bullet = collision.GetComponent<UDEAbstract-
Bullet>();
            if (!bullet.gameObject.activeSelf || bullet == null)
                return;
            UDEBulletPool.Instance.ReleaseBullet(collision.GetCompo-
nent<UDEAbstractBullet>());
        }
    }
}
```

If we release a bullet that is already released(thus the game object is not active), it will cause some bugs. Thus it is important to check whether the bullet is active.

VIII. Making Player

You successfully made an enemy. Now, you should make a player. All player have a script that extends `UDEPlayer`. You can customize weapons and shots of the player.

First, make a script that extends `UDEPlayer` class in `SansyHuman.UDE.Object` namespace. You have to implement ShootBullet() method that returns coroutine that shoots bullets when pressing the fire key. Here is an example.

```
using System.Collections;
using UnityEngine;
using SansyHuman.UDE.Object;
using SansyHuman.UDE.Management;
using SansyHuman.UDE.Util.Builder;

public class Player : UDEPlayer
{
    public UDEAbstractBullet bullet;

    public override IEnumerator ShootBullet()
    {
        UDEBulletMovement move1 =
UDECartesianPolarMovementBuilder.Create().Speed(10).Angle(90).Build();
        UDEBulletMovement move2_1 =
UDECartesianPolarMovementBuilder.Create().Speed(10).Angle(45).Build();
        UDEBulletMovement move3_1 =
UDECartesianPolarMovementBuilder.Create().Speed(10).Angle(135).Build();
        UDEBulletMovement move2_2 =
UDECartesianPolarMovementBuilder.Create().Speed(10).Angle(67).Build();
        UDEBulletMovement move3_2 =
UDECartesianPolarMovementBuilder.Create().Speed(10).Angle(113).Build();

        UDEBulletPool bulletPool = UDEBulletPool.Instance;
        Transform tr = gameObject.transform;

        while (true)
        {
            if (Input.GetKey(KeyCode.Z))
            {
                UDEAbstractBullet bullet1 = bulletPool.GetBullet(bullet);

                bullet1.Initialize(tr.position, tr.position, 0, this, null,
move1);

                UDEAbstractBullet bullet2 = bulletPool.GetBullet(bullet);
                UDEAbstractBullet bullet3 = bulletPool.GetBullet(bullet);
                if (Input.GetKey(KeyCode.LeftShift))
                {

                    bullet2.Initialize(tr.position, tr.position, 0, this,
null, move2_2);
                    bullet3.Initialize(tr.position, tr.position, 0, this,
null, move3_2);
                }
                else
                {

                    bullet2.Initialize(tr.position, tr.position, 0, this,
null, move2_1);
                    bullet3.Initialize(tr.position, tr.position, 0, this,
null, move3_1);
                }
            }
```

```
        yield return
StartCoroutine(UDETime.Instance.WaitForScaledSeconds(BulletFireInterval,
UDETime.TimeScale.PLAYER));
            }
        }
}
```

It is similar to enemy pattern, but the difference is bullets are fired when the fire key is pressed(in this example, Z). Also, when the player is moving slowly(by default, when pressing left shift), the bullets are fired in a narrower angles.

In `UDEPlayer` script, there are several keys defined in default in `Move()` method. Arrow keys are move, and left shift is slow down. Also, BulletFiredInterval is defined as a property. You can change them in the inspector

Now, make a game object of the player and add Rigidbody 2D, Collider 2D, and the player script. Set Rigidbody to kinematic and collider to trigger. Also, set the tag and layer to 'Player'.

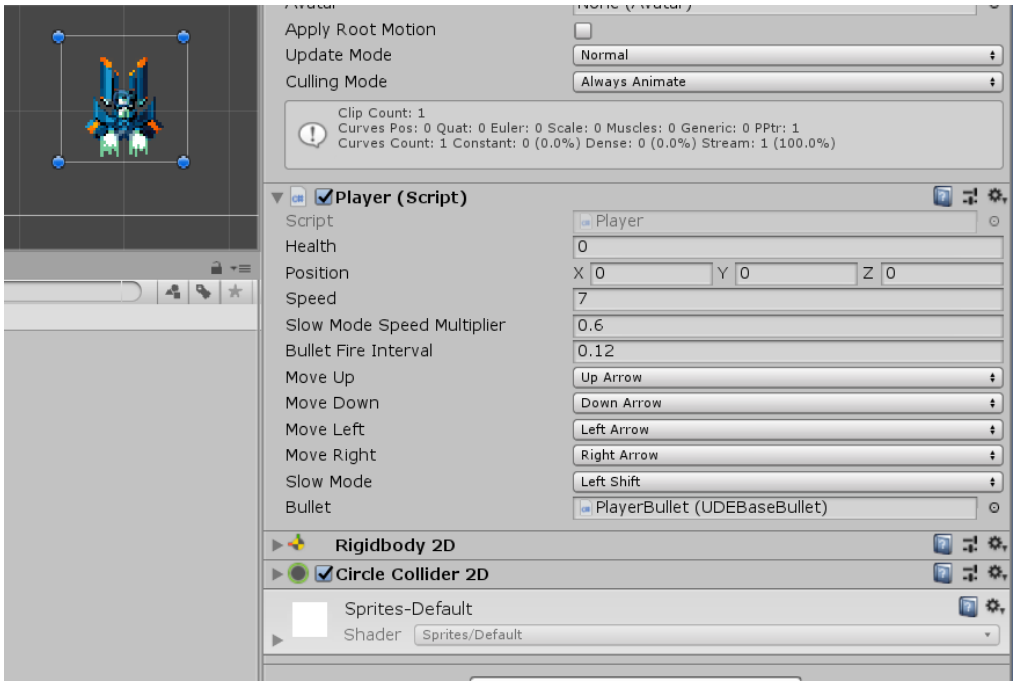

Figure 16 Setting the player

There are several basic properties to set.

PROPERTIES	DESCRIPTIONS
Health	Health of the player. In default settings, player gets 1 damage from all enemies' weapons and sets to be invincible for 3 seconds.
Speed	Speed of the player.
Slow Mode Speed Multiplier	Multiplier to the player's speed when the player is in slow mode(default key is left shift). For example, if the player's speed is 7 and multiplier is 0.6, the player moves in speed 4.2 when pressing slow mode key.
Bullet Fire Interval	Time interval to fire the bullets.

Also, you can change the key mapping at the inspector.

Now, make a player's bullet and add it in the Bullet property in the inspector. If you run the game, you can move the player and fire bullets.

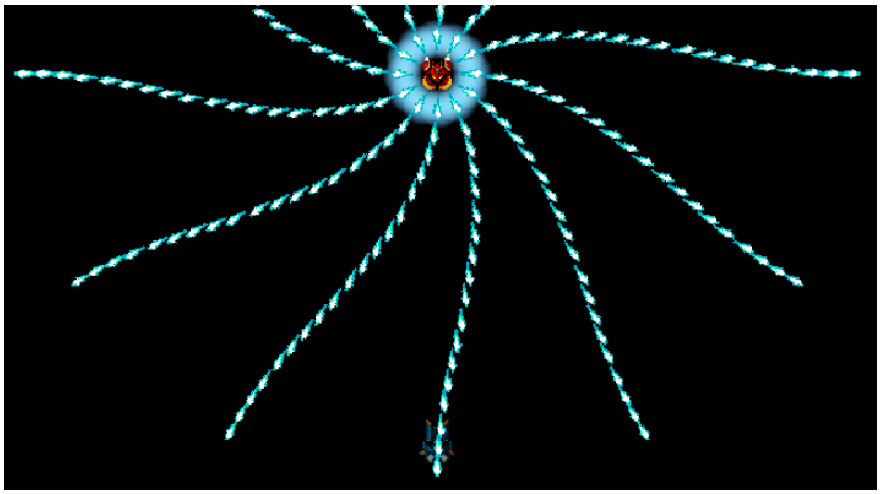

Figure 17 Player in the game

You can also make the player to shoot homing bullets. Open the prefab of the player's bullet and replace `UDEBaseBullet` script to `UDEHomingBullet` script.

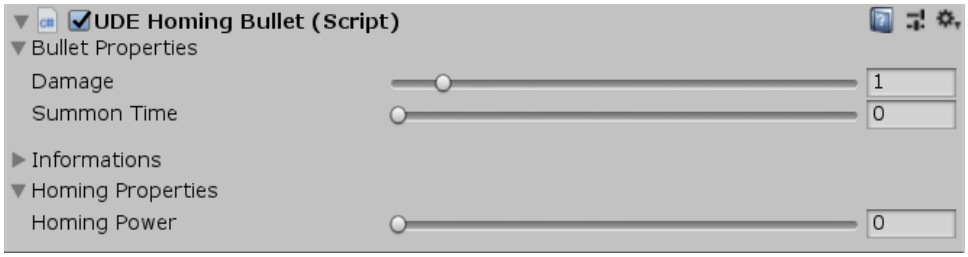

Figure 18 UDEHomingBullet script in inspector

You can set homing power of the bullet in the inspector. Homing power has a value between 0 to 20. 0 means the bullet will not track the enemies. After setting the homing power, fix some codes in the player's script.

```
UDEHomingBullet bullet1 = (UDEHomingBullet)bulletPool.GetBullet(bullet);

bullet1.Initialize(tr.position, this, move1);

UDEHomingBullet bullet2 = (UDEHomingBullet)bulletPool.GetBullet(bullet);
UDEHomingBullet bullet3 = (UDEHomingBullet)bulletPool.GetBullet(bullet);
if (Input.GetKey(KeyCode.LeftShift))
{

    bullet2.Initialize(tr.position, this, move2_2);
    bullet3.Initialize(tr.position, this, move3_2);
}
else
{

    bullet2.Initialize(tr.position, this, move2_1);
    bullet3.Initialize(tr.position, this, move3_1);
}
```

`UDEHomingBullet` has a simpler version of the method `Initialize()`. It only takes initial position, the player instance, and the initial movement. Homing bullets automatically turn the direction toward to the closest destroyable enemies(they will not track invincible enemies). Thus the homing bullet can only be used by the player.

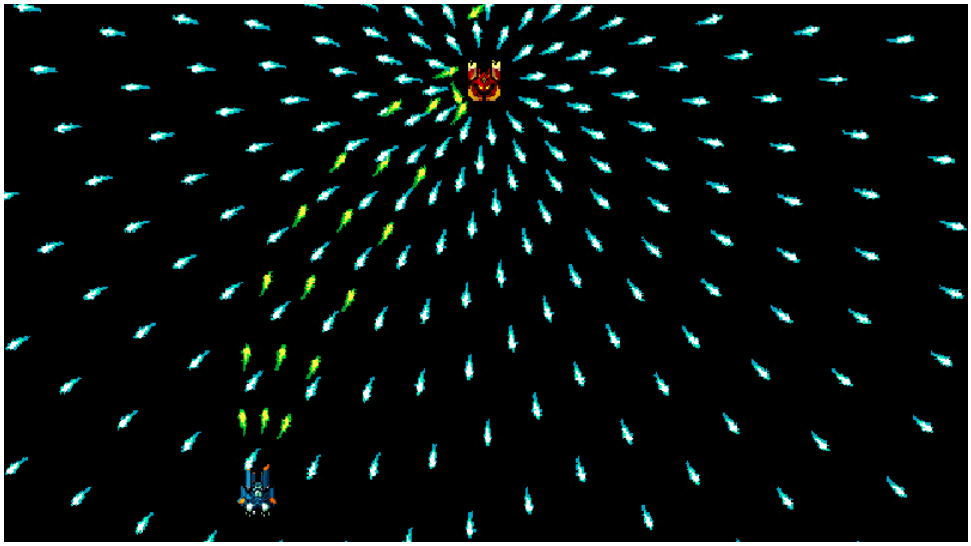

Figure 18 Homing bullet

IX. Making Laser

You can also make lasers. All lasers extends `UDELaser` class. There are two kinds of lasers; Straight laser, and curved laser.

Straight laser

Straight laser has a script `UDEStraightLaser`, which extends `UDELaser`. To make a laser, first make a material of the laser. The material has a shader 'Legacy Shaders/Particles/Alpha Blended Premultiply'.

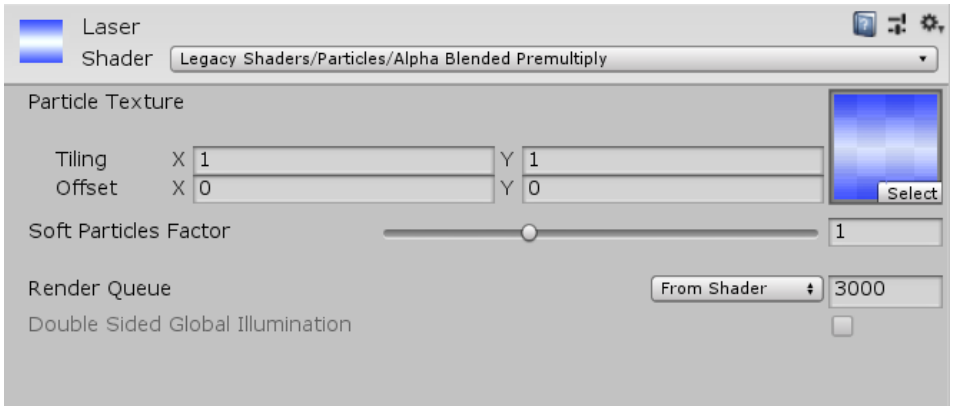

Figure 19 Material of the laser

Then, make a prefab of line renderer. The line has a material you just made. Set corner and end cap vertices properly. If done, add EdgeCollider2D and UDEStraightLaser script. Set the collider to be a trigger.

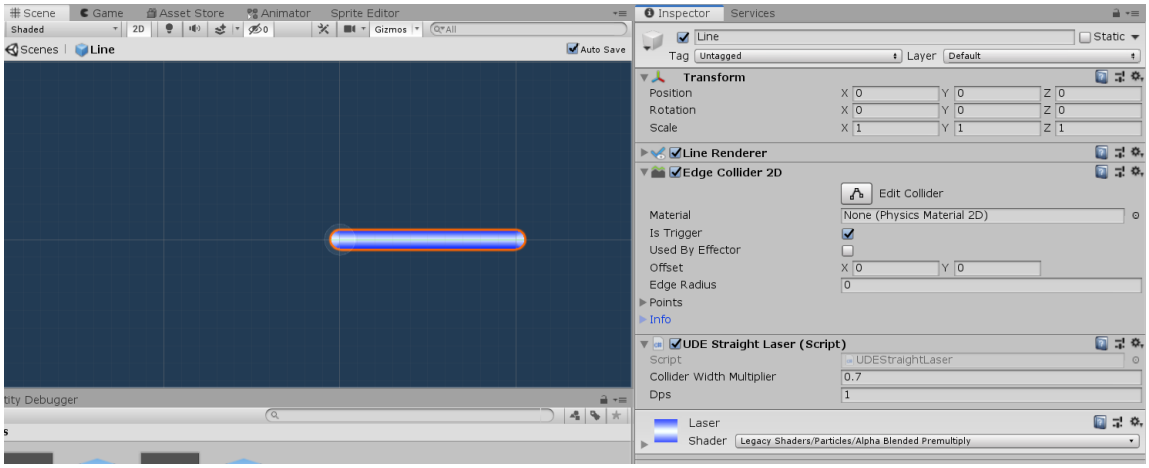

Figure 19 Laser prefab

There are two properties you can set in the inspector.

PROPERTIES	DESCRIPTIONS
Collider Width Multiplier	Determines what width the edge collider will have. The edge collider will have a width of the line renderer multiplied by the multiplier.
Dps	Damage per second. Damage that the laser deals in 1 seconds. Only applied to the damage to the enemy.

Also, you can set the width of the laser in the Line Renderer component.

The laser can be used by both enemy and player. Set the tag of the laser to 'Laser'. If the laser is used by the player, set the layer to 'PlayerBullet'. If the laser is used by enemies, set the layer to 'EnemyBullet'. In this example, I will add laser to the player. Open the player script from the previous chapter and add some codes.

```
…
public class Player : UDEPlayer, IUDELaserFirable
{
    public UDEAbstractBullet bullet;
    public UDEStraightLaser laser;

    private UDEStraightLaser firedLaser;
    private bool laserFirable = true;

    public void FireLaser()
    {
        firedLaser = Instantiate<UDEStraightLaser>(laser);
        firedLaser.Initialize(transform.position, this, true, Vector3.zero,
UDETime.TimeScale.PLAYER, new Vector2(0, 1000), 10);
        laserFirable = false;
    }

    public void RemoveLaser()
    {
        firedLaser.DestroyLaser();
        laserFirable = true;
    }

    public override IEnumerator ShootBullet()
    {
        …

        while (true)
        {
            if (Input.GetKey(KeyCode.Z))
            {
                …
                if (laserFirable)
                    FireLaser();
            }
            else
            {
                if (!laserFirable)
                    RemoveLaser();
            }

            yield return
StartCoroutine(UDETime.Instance.WaitForScaledSeconds(BulletFireInterval,
UDETime.TimeScale.PLAYER));
        }
    }
}
```

IUDELaserFirable interface defines methods that fires and removes the laser. When you fire the laser, call FireLaser() method. When you stop firing the laser, call RemoveLaser() method.

UDEStraightLaser have a method Initialize().

```
public void Initialize(Vector2 origin,
                       UDEBaseCharacter originCharacter,
                       bool followOriginCharacter,
                       Vector2 initialLocalHeadLocation,
                       UDETime.TimeScale timeScale,
                       Vector2 headVelocity,
                       float maxLaserLength)
```

PARAMETERS	DESCRIPTIONS
origin	Origin of the laser's local space.
originCharacter	The character who shot the laser.
followOriginCharacter	If true, the origin of the laser is set to the position of the origin character.
initialLocalHeadLocation	Start position of the laser in the laser's local position. If the laser starts at the origin, then initial local head location will be zero vector.
timeScale	Time scale to use.
headVelocity	Velocity of the head in the local space.
maxLaserLength	Maximum length of the laser.

If you rotate the laser in the world space, the velocity vector and the start point of the laser also rotates. Thus you don't need to calculate velocity vectors and start points of lasers with different directions. You can just rotate the laser.

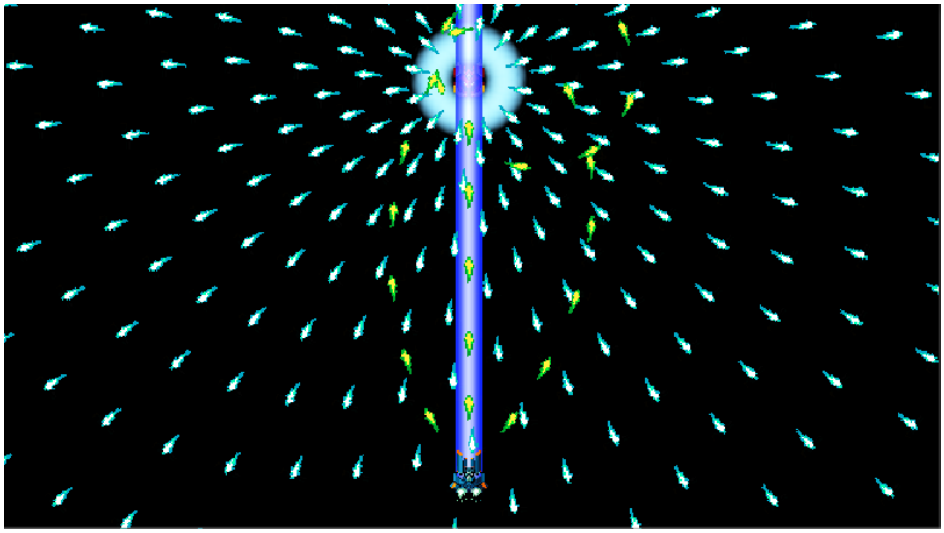

Figure 20 Player is firing the laser

Curved laser

Curved laser have a script `UDECurveLaser`, which extends `UDELaser`. To make a curved laser, you have to define the path of the laser using `UDEMath.CartesianTimeFunction` or `UDEMath.PolarTimeFunction` delegates. These delegates are functions that takes a time and returns the cartesian or polar coordinates at that time.

You can define the function directly, but it is easier to use `UDECurve` script. `UDECurve` is a class that represents various curves. There are currently 3 types of curves.

Types	Descriptions
`BEZIER`	Bezier curve
`CUBIC_SPLINE`	Natural cubic spline curve
`CATMULL_ROM`	Catmull-Rom spline curve. Unlike other curves, it starts from the 2nd control point and ends at (n-1)th point, when the number of control points is n.

For example, let's make a path of Bezier curve. First, create an empty game object named 'LaserPath'. Then add UDECurve script to the object.

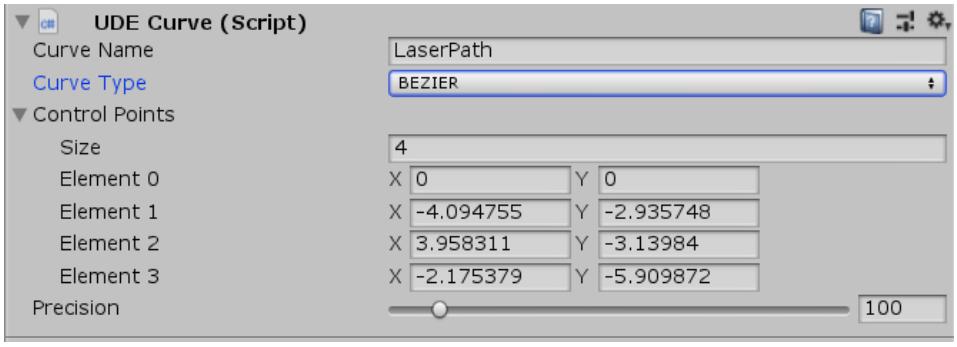

Figure 21 UDECurve script in inspector

There are four properties of the curve.

PROPERTIES	DESCRIPTIONS
Curve Name	Name of the curve. This name is used when you access to the curve in the script.
Curve Type	The type of the curve.
Control Points	The list of control points. The position of these points can be adjusted either on the scene or in the inspector.
Precision	The number of intervals used to preview the path on the scene. It does not affect the real path's precision.

Name your curve and set the type. In this example, I set the name to be 'LaserPath' and the type to be BEZIER. Then add control points as you want. In this example, there are 4 control points.

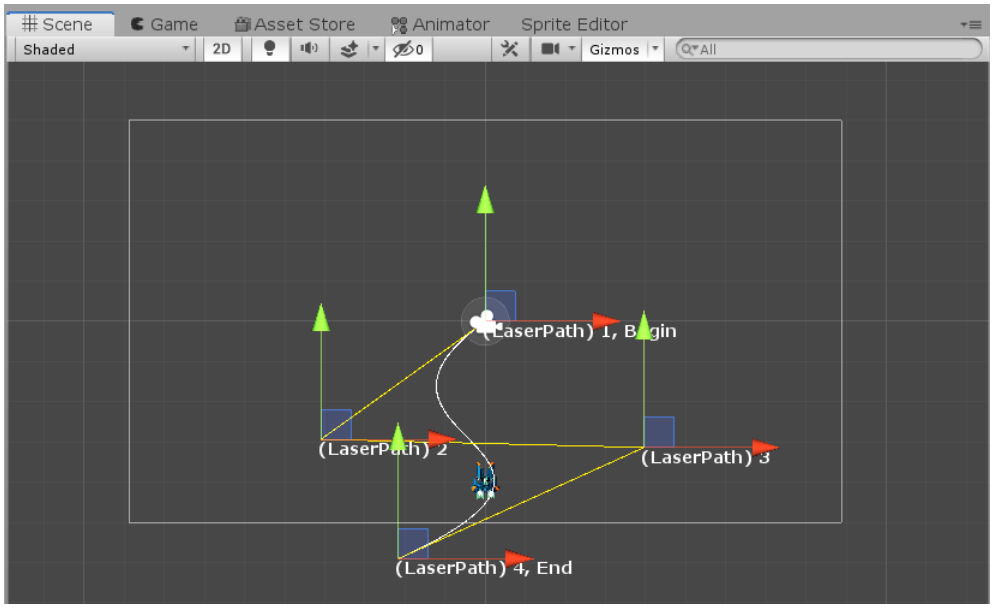

Figure 22 UDECurve on the scene

You can adjust the positions of the control points on the scene. Or you can put the coordinate of the point directly in the inspector. Control points are connected with yellow line and named with the name of the curve and the order of the point. The real path is the white line. The precision of the white preview path is different by the Precision parameter in the inspector.

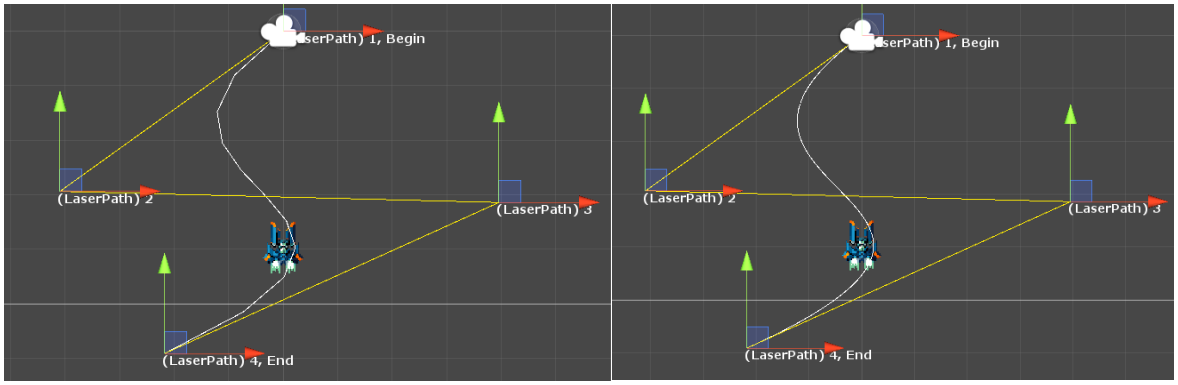

Figure 23 The curve of precision 10(left), the curve of precision 100(right)

If you made a curve, then you can access to the curve object using `UDECurve.GetCurveByName(string name)` static method. You can get the curve by putting the name of the curve you set in the inspector(Not the game object name in the hierarchy!).

Now, let's make a curve laser. You can just copy the straight laser prefab and replace `UDEStraightLaser` to `UDECurveLaser`.

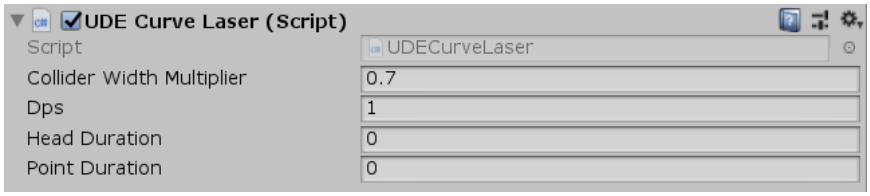

Figure 24 UDECurveLaser script in inspector

Comparing to `UDEStraightLaser`, there are two more properties.

PROPERTIES	DESCRIPTIONS
Head Duration	Duration of the laser head. Time to extend the laser
Point Duration	Duration of each points on the path of the laser head.

In the curve laser, the laser head moves on the path. Every fixed time interval, the laser adds points of line renderer and edge collider at the position of the head at that time. If the time from the point added to the current time exceeds the duration of the point, then the point is removed from the laser. Also, after the duration of the head, the laser head is removed and the laser no longer extends.

Now, let the enemy to fire the laser we made. First, change the layer of the laser to EnemyBullet since it will be fired by the enemy. Then add a new parameter to the previous `TestShotPattern` script.

```
[SerializeField]
private UDECurveLaser laser;
```

In the current version of the UDE, `UDEBaseShotPattern` does not support the laser by default. Thus you should add laser variable in the implemented script if you need.

Next, add some codes that fires the laser. In this case, I let the enemy

fires 5 lasers every 7 seconds.

```
…

using SansyHuman.UDE.Util;

public class TestShotPattern : UDEBaseShotPattern
{
    …

    [SerializeField]
    private UDECurveLaser laser;
    [SerializeField]
    private int NumberOfLasers = 5;

    protected override IEnumerator ShotPattern()
    {
        …

        int laserFlag = 100;
        float laserDeg = 360f / NumberOfLasers;
        var laserPathFunc =
UDECurve.GetCurveByName("LaserPath").GetFunctionOfCurve().Composite(UDETra
nsitionHelper.easeInOutCubic).Composite(t => t / 1.5f);

        while (true)
        {
            …

            if (laserFlag == 100)
            {
                laserFlag = 0;
                for(int i = 0; i < NumberOfLasers; i++)
                {
                    UDECurveLaser laser =
Instantiate<UDECurveLaser>(this.laser);
                    laser.transform.rotation = Quaternion.Euler(0, 0, i *
laserDeg);
                    laser.Initialize(originEnemy.transform.position,
originEnemy, false, UDETime.TimeScale.ENEMY, 1.5f, 0.35f, laserPathFunc);
                }
            }
            else
                laserFlag++;

            yield return
StartCoroutine(UDETime.Instance.WaitForScaledSeconds(0.07f,
UDETime.TimeScale.ENEMY));
        }
    }
}
```

In this example, I set the duration of the head to be 1.5 seconds and each point to be 0.35 seconds. Thus the laser will extend for 1.5 seconds. When the

last point of the laser dissapears, the laser destroies itself, thus you don't need to destroy the laser manually.

`GetFunctionOfCurve()` method gets the `UDEMath.CartesianTimeFunction` delegate that receives a float value and returns the coordinate in cartesian coordinate system. Mathematically, it is a function of the form $\mathbf{x} = f(t) \in \mathbb{R}^2$.

But `GetFunctionOfCurve()` returns a function whose domain is from 0 to 1; $f(0)$ is a start point and $f(1)$ is an end point. The laser puts the passed time to the function, thus we should make the function to have a domain from 0 to 1.5.

`Composite()` is an extension method defined on function delegates whose domain are $\mathbb{R}$. Suppose that there are two functions $f: \mathbb{R} \to \mathbb{R}^2$ and $g: \mathbb{R} \to \mathbb{R}$(In UDE, f is an `UDEMath.CartesianTimeFunction` and g is an `UDEMath.TimeFunction`. If we do `f.Composite(g)`, it will return a new function $f \circ g: \mathbb{R} \to \mathbb{R}^2 = f(g(t))$. If you want to learn more about `Composite()`, see the API document on `UDEMath`.

Thus the code

```
UDECurve.GetCurveByName("LaserPath").GetFunctionOfCurve().Composite(UDETra
nsitionHelper.easeInOutCubic).Composite(t => t / 1.5f);
```

Composites the ease function and a function that stretches the domain. There are several predefined ease functions in `UDETransitionHelper` class. If you want to see more ease functions, see the API document. Every ease functions have a domain and range from 0 to 1. Thus the new function first compresses the domain from 0 to 1.5 to 0 to 1 and put the value to the ease function. And the ease function returns the value between 0 to 1 and again put it in the curve function. Mathematically,

$$f: [0, 1] \to \mathbb{R}^2: \ curve \ \ function$$

$$g: [0, 1] \to [0, 1]: \ ease \ \ function$$

$$\mathrm{h}: [0, 1.5] \to [0, 1]: \ domain \ \ stretch$$

$$f \circ g \circ h: [0, 1.5] \to \mathbb{R}^2 = f(g(h(t)), 0 \leq t \leq 1.5$$

One thing to remember is that the coordinate from the curve function is the relative coordinate of the head; If the coordinate is (0, 0), the head is on the origin of the laser. Thus if you want the laser to start from the origin which you

set, you have to set the start point of the curve to be (0, 0).

You should initialize the laser to make the laser start extension.

```
public void Initialize(Vector2 origin,
                       UDEBaseCharacter originCharacter,
                       bool followOriginCharacter,
                       UDETime.TimeScale timeScale,
                       float headDuration,
                       float pointDuration,
                       UDEMath.CartesianTimeFunction headFunction)
```

The last parameter can be `UDEMath.PolarTimeFunction` which returns polar coordinates. Except for that, those two methods are identical.

PARAMETERS	DESCRIPTIONS
`origin`	Origin of the laser's local space.
`originCharacter`	The character who shot the laser.
`followOriginCharacter`	If true, the origin of the laser is set to the position of the origin character.
`timeScale`	Time scale to use.
`headDuration`	Duration of the head. The time to extend the laser.
`pointDuration`	Duration of each points on the path of the laser head.
`headFunction`	Function of the curve on which the head of the laser will move.

You can just rotate the laser object to make it to extend to different angle. Thus you don't need to create multiple curve function.

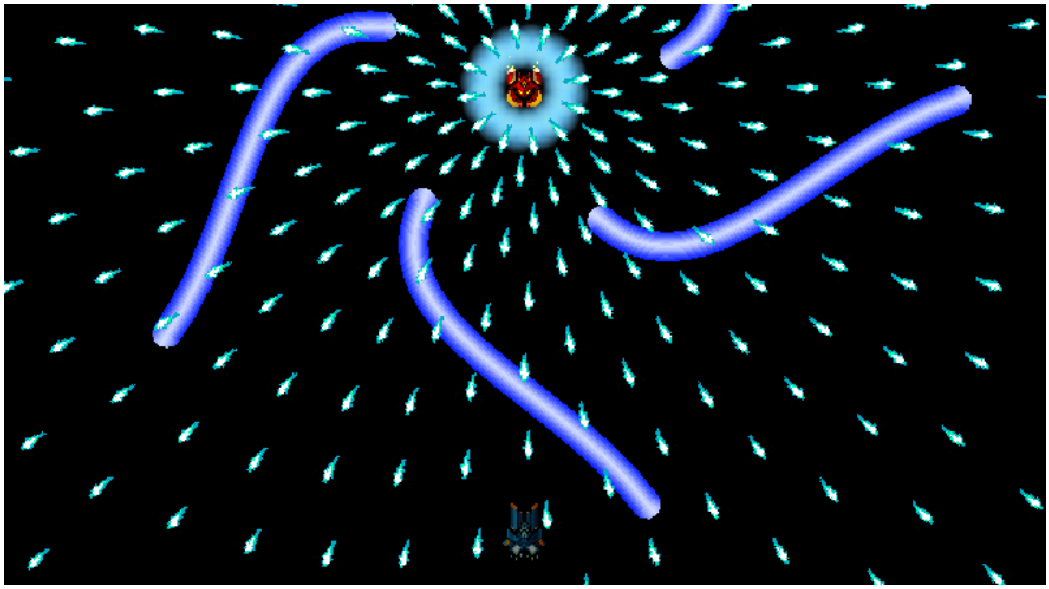

Figure 25 Curve laser in game

The path of the laser can be different by the shape of the curve.

X. Multithreading

Cautions: Current API using job system and ECS is beta. Thus it can be removed or revised in the later version of the engine. Also, it can be unstable.

All objects we have seen are all MonoBehaviour-based. The most serious problem of MonoBehaviour is that it is thread-unsafe; we cannot use multithreading on MonoBehaviour. Thus movements of all objects, such as bullets, lasers, and characters are calculated on a single thread.

Fortunately, Unity recently started to support limited multithreading using job system and ECS. If you don't know what job system and ECS are, see the documents on the official site of Unity.

The current version of UDE support the bullet that uses hybrid ECS; it used both ECS and MonoBehaviour. The below is a list of scripts related to ECS.

Scripts	Description
UDEBulletECS	Script for ECS bullets. All ECS bullets are child of 'UDEAbstractBullet' class.
UDEBulletMovementECS	Struct that contains the movement of the ECS bullet. Unlike UDEBulletMovement, it does not support acceleration function since the job system does not support the managed objects.
UDEBulletMovementSystem	Component system that calculates movements of ECS bullets.
UDEBulletMovements	Component data struct that contains the current movement of the bullet.

You don't need to know other scripts, but `UDEBulletECS`. To use ECS, you only have to add `UDEBulletECS` script to the bullet prefab instead of `UDEBaseBullet`. Other than that, all things such as getting bullet from the pool, initialization, and returning to the pool are same. You don't need to fix `TestShotPattern` script.

After you replaced the bullet in the shot pattern to ECS bullet, open the entity debugger in Window-Analysis-Entity Debugger. If you run the game, you will see this on the entity debugger.

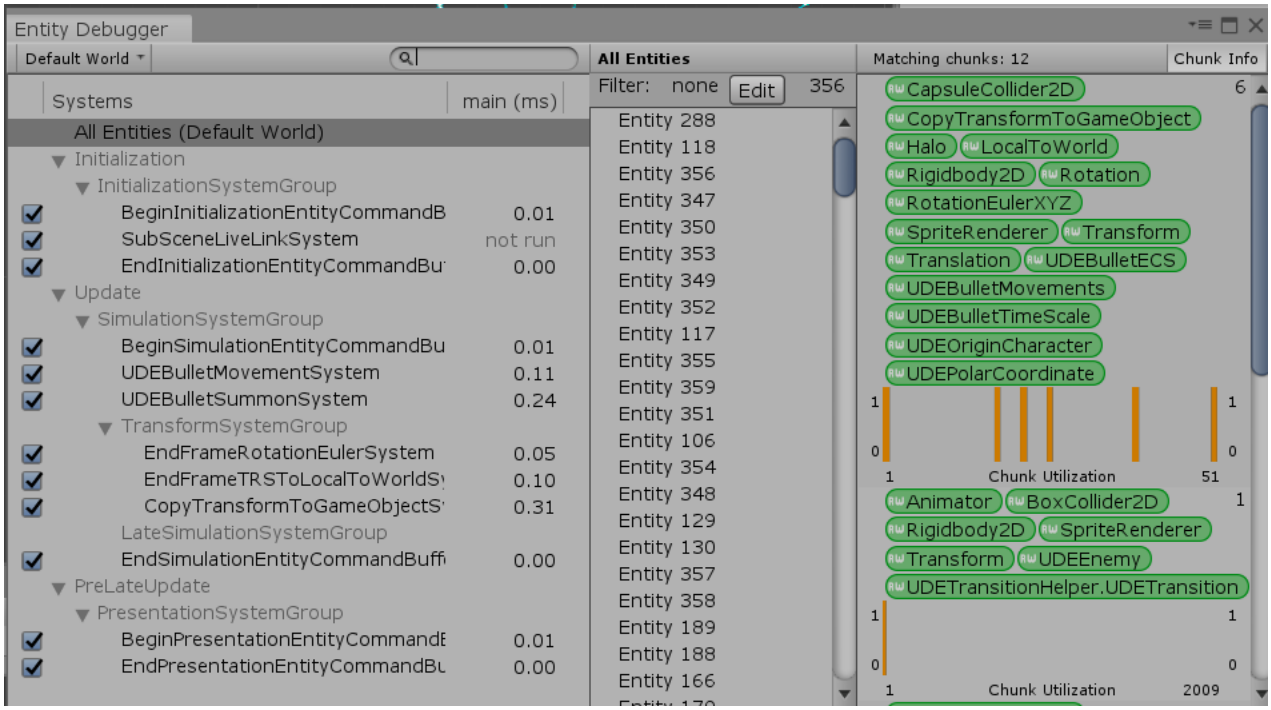

Figure 26 Entity Debugger

The left part shows the systems that are running in the world. You can see two UDE-related systems, `UDEBulletMovementSystem` and `UDEBulletSummonSystem`. `UDEBulletMovementSystem` calculates movements of bullets, and `UDEBulletSummonSystem` controls bullet's summon phase, same to non-ECS bullets. `UDEBulletMovementSystem` supports Burst compile, which makes the system running faster. You can turn on Burst compile in Jobs-Burst-Enable Compilation.

The middle part shows the list of entities in the world. You can see 300 to 400 entities, which are bullets. When initializing bullet, new entity is created of the bullet. When removing bullet, the entity of the bullet is destroyed.

The right part shows the components on the entity. All components starting with ‘UDE’ are UDE-related components.

XI. Conclusion

Congratulations! You finished the tutorial of the Universal Danmaku Engine! But this tutorial didn't see all features supported by UDE. There are more features you can use. If you want to see more features, see API documents in the package.

Finally, there are some points to remember.

1. Initialize the bullet immediately after you got it from the pool.

The bullet pool sets the bullet active before returning it. Thus it would cause unexpected result if you don't initialize it right after you got it(for example, the bullet can appear in the wrong place).

2. Initialize all initializable objects before you do something with them.

All objects that has a method `Initialize()` should be initialized before you use them. Or they will not work properly.

3. Do not initialize objects twice.

If you initialize an object you already initialized, it will cause unexpected result(for example, bullets will ignore second initialization). All initializable objects are children of `UDEInitializable`. In this class, a property `UDEInitializable.Initialized` is defined which returns whether the object is initialized. So check the object before you initialize it.

4. Do not return bullet to the pool that already returned.

All bullets that are returned to the pool are not active. So you can easily check whether you returned the bullet by just checking if the bullet is active. The pool will ignore the return of the bullet that is not active, so it will not cause serious problems. But in terms of performance, it is better not to return the bullet twice.