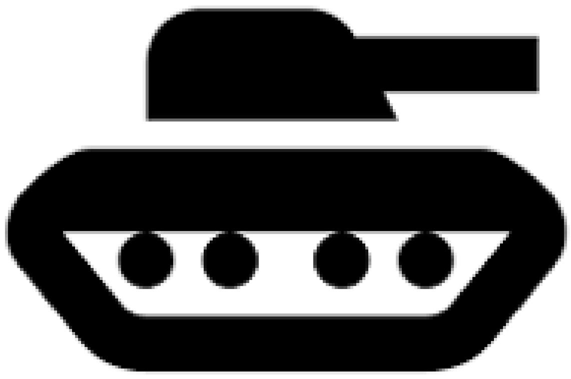
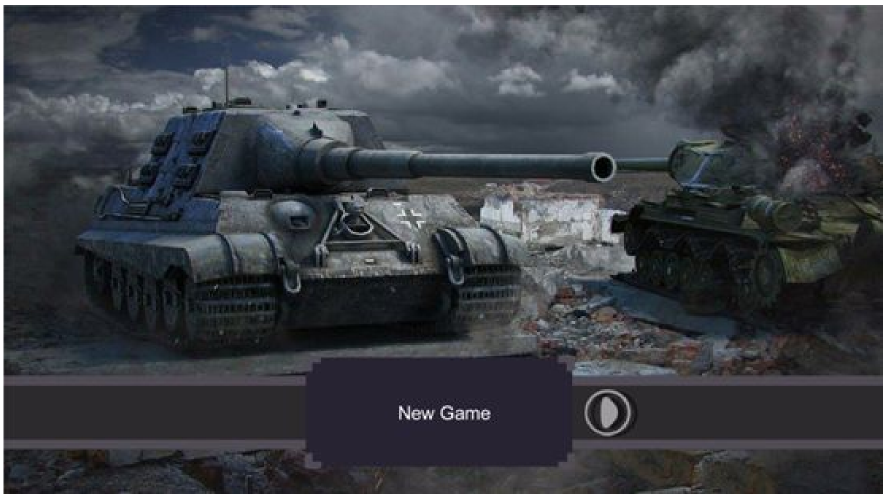
3D Game Programming Final Report



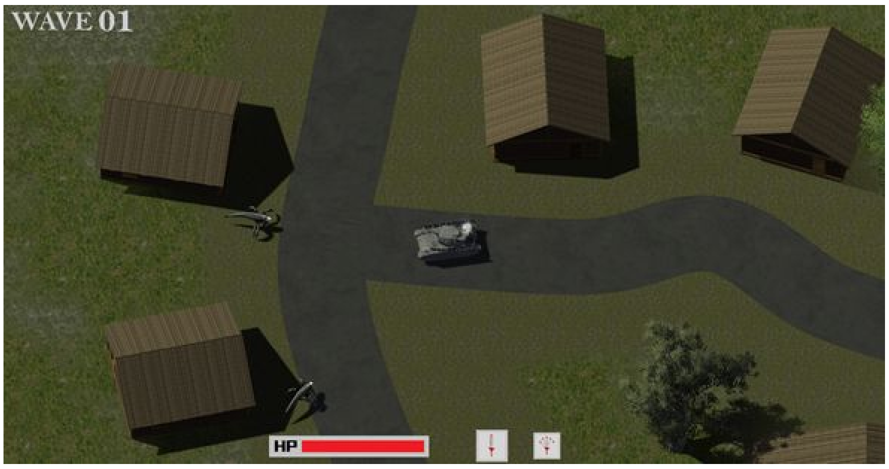
Tank Party

0316235 簡立哲, 0316241 吳玉辰, 0656623 侯家元

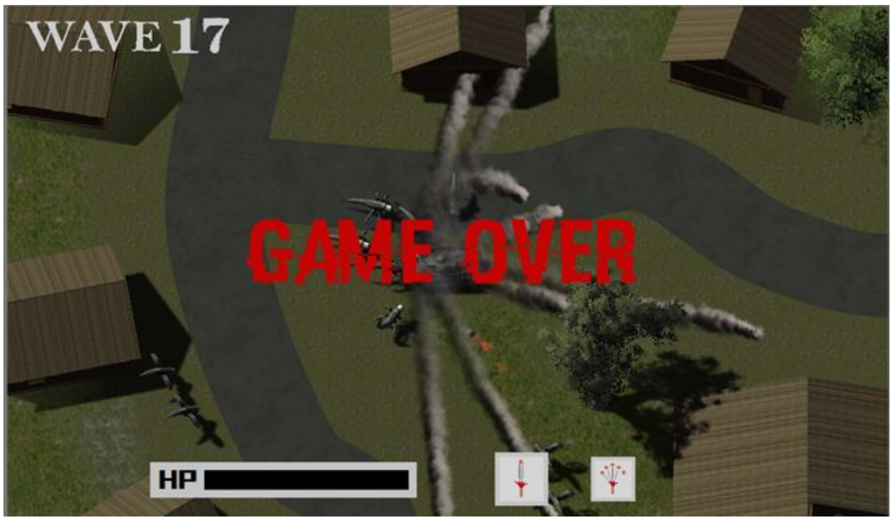
Representative Images :



Title Scene



Game Scene



Game Over Scene

● Introduction

Although most of the games need only one person per machine to play, there are still some games that need multiplayers in one machine, such as “King of Opera”, “Keep talking and nobody explodes”.

We would like to try this type of game, so we thought about something that needs two or more people to control, and we found tanks which need driver, gunner and others.

With this as the topic, we want to create a tank shooting game with multiplayers cooperation on one machine.

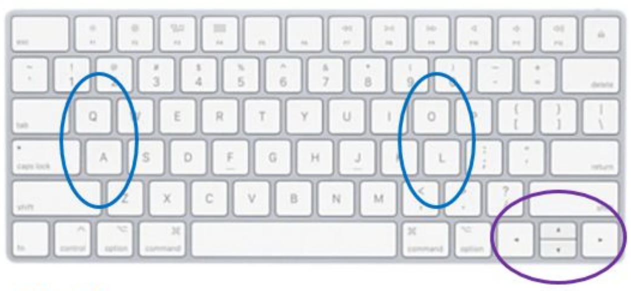
● Game Design

- Game Features

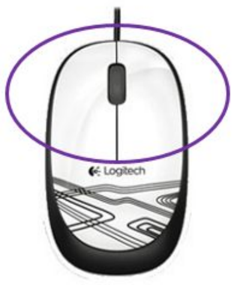
This game is a multiplayer shooting arcade game with a third person look down camera and unlimited waves of enemies.

The players will control a tank to destroy enemies that come in waves. When a wave of enemies are all destroyed, the wave will progress. The objective of the players is to survives as many waves as they can.

- Game Control



Driver：
Keyboard “Q” “A”：Left track forward and backward.
Keyboard “O” “L”：Right track forward and backward.



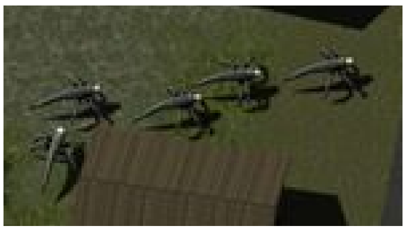
Gunner：
Mouse left click：Fire.
Mouse right click：Trigger Turent Rotation.
Move right：Turret rotate clockwise.
Move left：Turret rotate counterclockwise.
Keyboard “Right” “Left”：Change weapon.

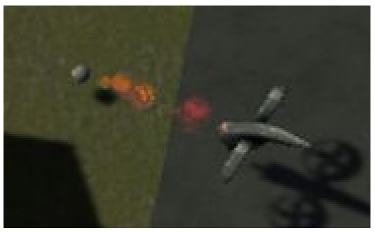
The main idea is to let two people play this game. So we do not use the simple WASD movement control, but use the QAOL for the driver so it is hard for him to control both the driver’s work and the gunner’s work.

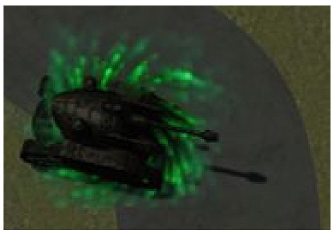
- Game Level

We have three different difficulties, each has a different map. The enemies’ attack value and health are also different. The difficulty can be chosen in the title scene.

There are three type of enemies : **Melee attack robots**, **Ranged attack drones** and **Powerful boss tanks**. Melee attack robots will attack the players when it is right beside them. Ranged attack drones will shoot bullets towards the players when they are in attack range. The boss tanks will shoot three bullets in a row, and also it has a laser gun that needs to charge up and is very hard to dodge. The enemies' type and number that will spawn in the wave are different due to the wave number.


(a) Melee attack robots


(b) Ranged attack drone


(c) Powerful boss tank

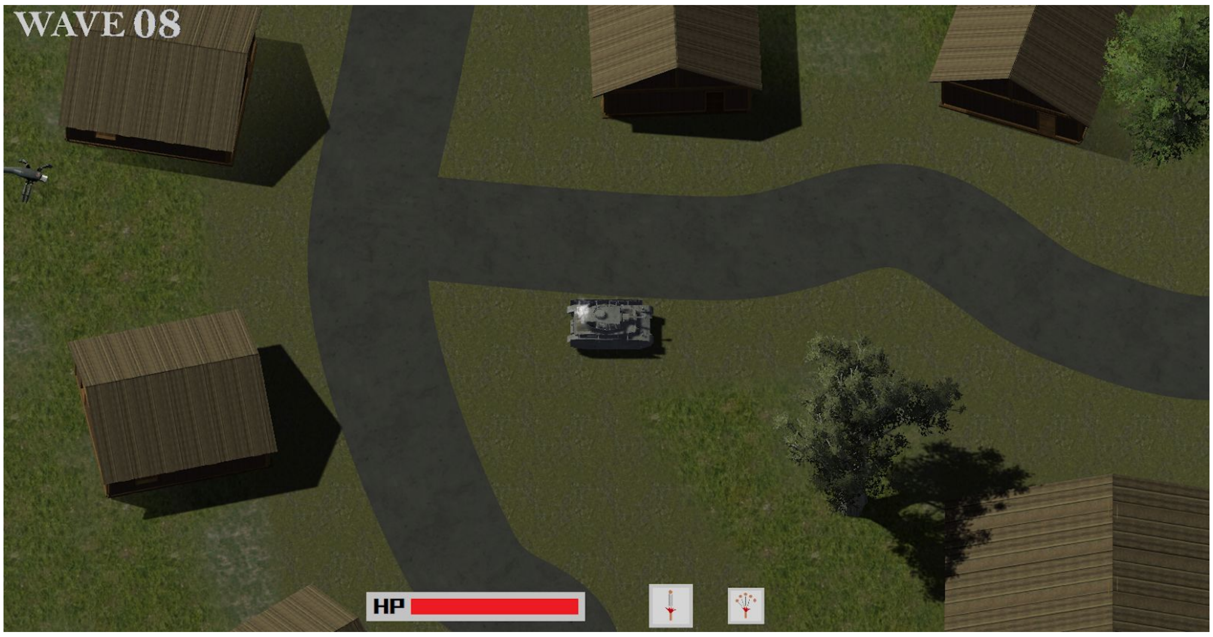
One of the player controls the tank's chassis, which control the movement of the tank. The other player controls the turret, which control the attack direction of the tank.

For players, there are two types of attack, one is the original cannon which has short CD time, the other is the shotgun cannon which shoot multiple bullets but has a long CD time.

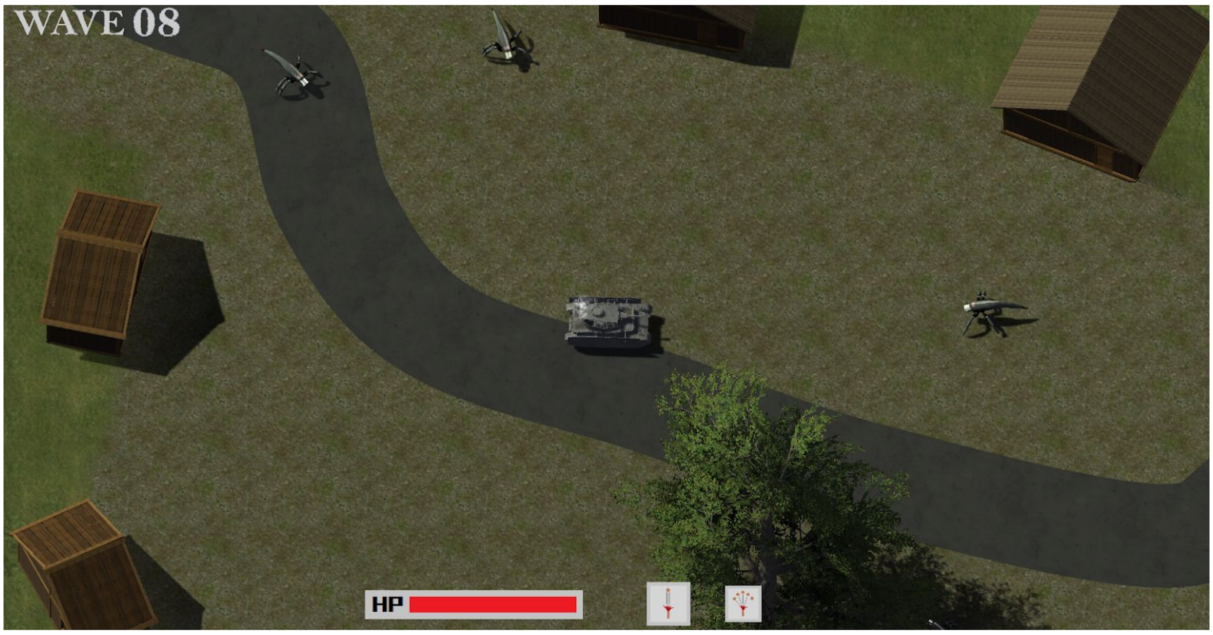
When destroying an enemy, there is a chance that a repair kit will drop. Taking it will restore health of the tank.

Houses and Trees will block the bullets and the movement of every units. There are also muds on the terrain that will slow down the players movement.

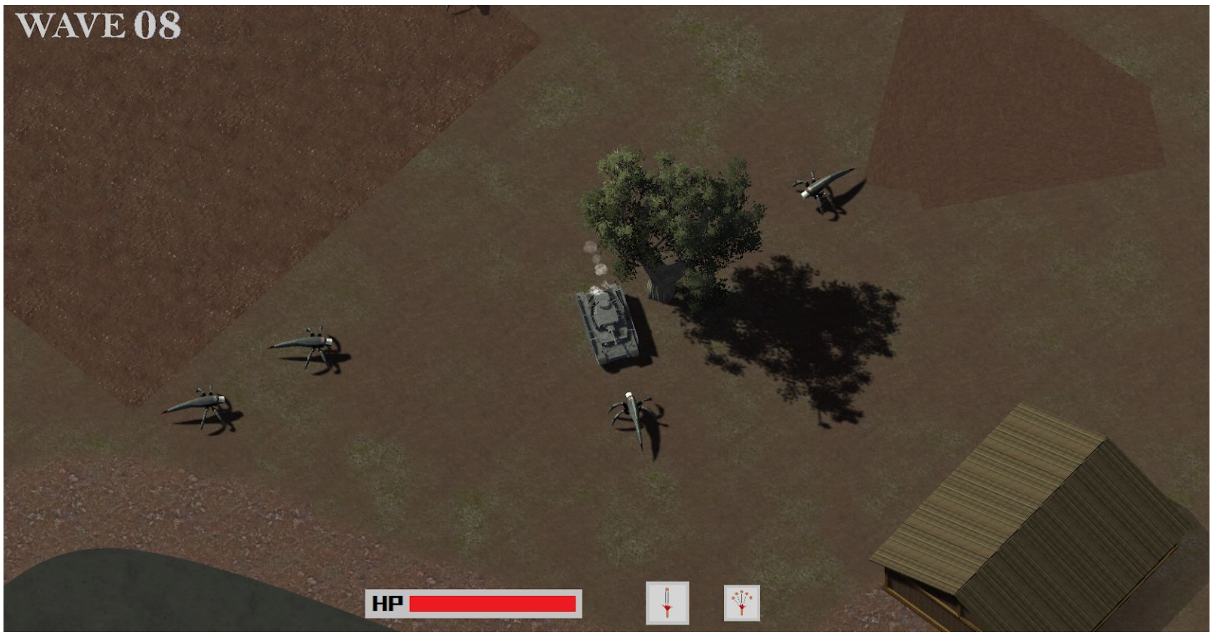
We created 3 scenes and changed the game setting for different difficulties.



(d) **Difficulty Easy**, enemies have low health and low attack value. In Easy scene, there are many obstacles to help player avoid from being attacked.

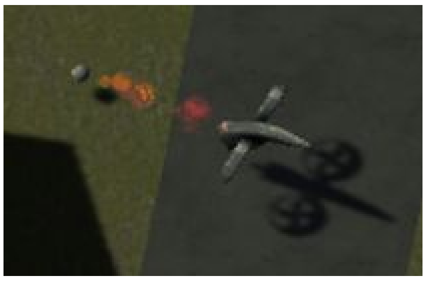


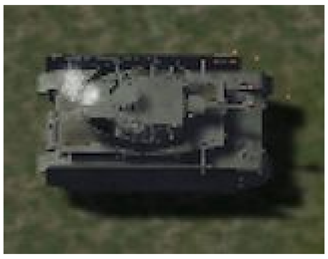
(e) **Difficulty Normal**, enemies have medium health and medium attack value. In Normal scene, the numbers of houses and trees are decreased.

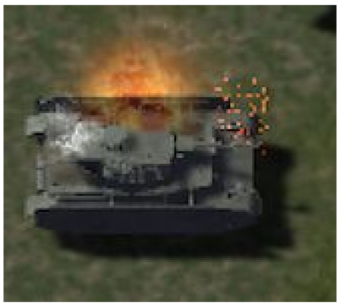


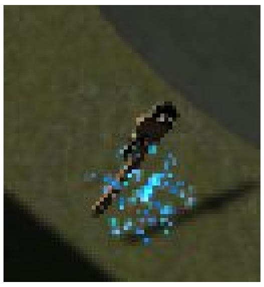
(f) **Difficulty Hard**, enemies have high health and high attack value. In Hard scene, there are few houses and trees. There are also some muds to slow down Player.

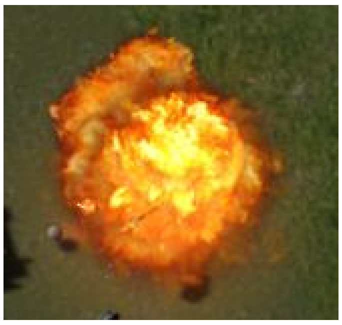
- **Particle Systems**

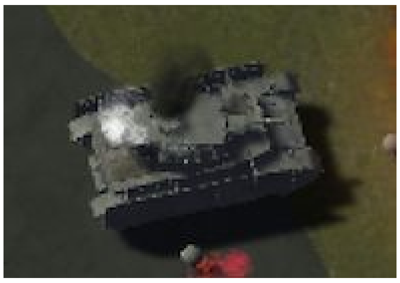

1. The flame behind the enemy’s bullet.

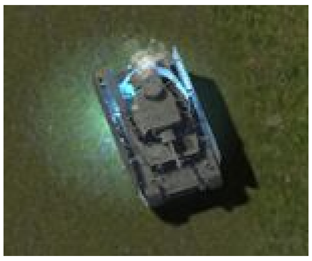

2. White engine smoke of the players and the shooting spark.

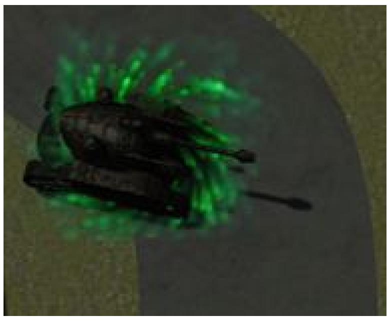

3. Bullet explosion.

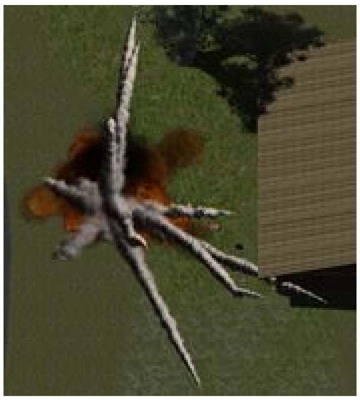

4. Blue spark on repair kit.


5. Enemy destroyed explosion.


6. Black smoke when the players are in low health.


7. Laser explosion.


8. Boss tank’s charging effect.


9. Players destroyed explosion.

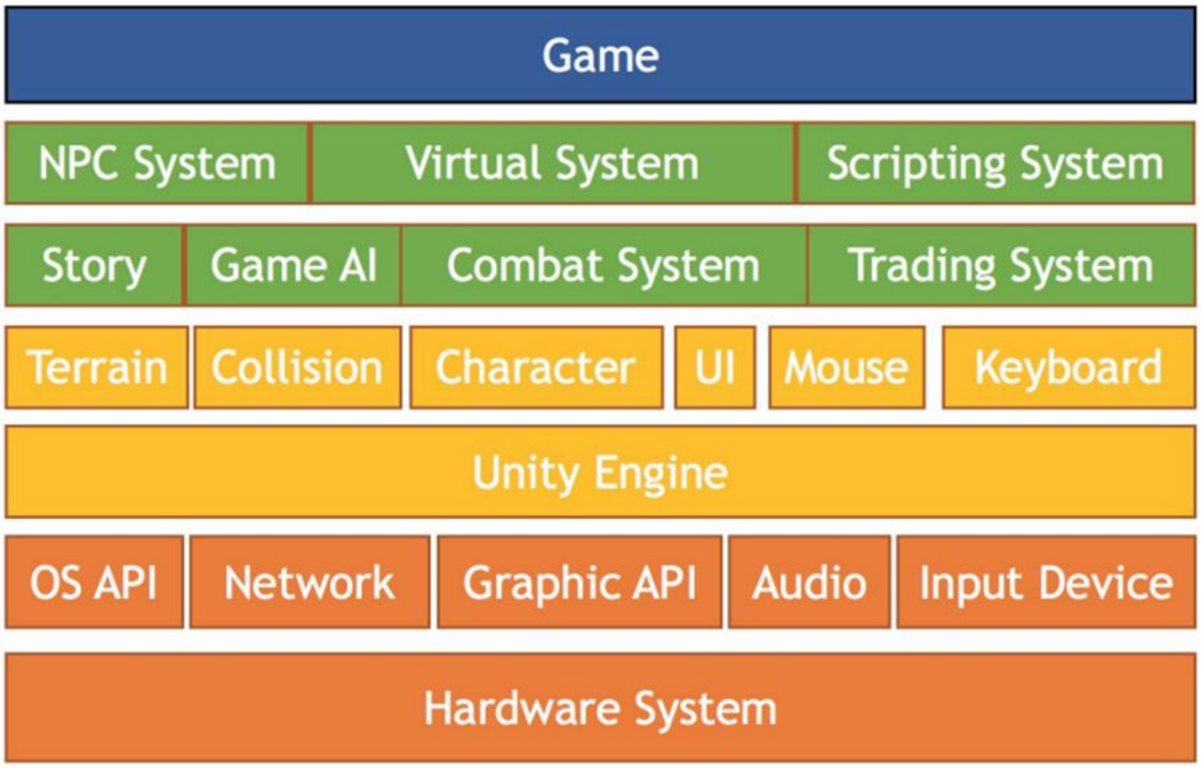
- **Sound Effects**
  1. Cannon Fire
  2. Engine Roaring
  3. Cannon Explode
  4. Enemy Explode
  5. Laser Charging
  6. Laser Fire
  7. Enemy Cannon Fire
  8. Enemy Melee Attack
  9. Menu Bottom Push
  10. Background Music

- **SWOT**

  S : A game that can play together with only one computer. Good for small party or friends’ social time.
  W : It need two players to play this game, it’s difficult for players to play alone.
  O : This kind of cooptation games isn’t common.
  T : There are a lot of party games online, so the players have many choices.

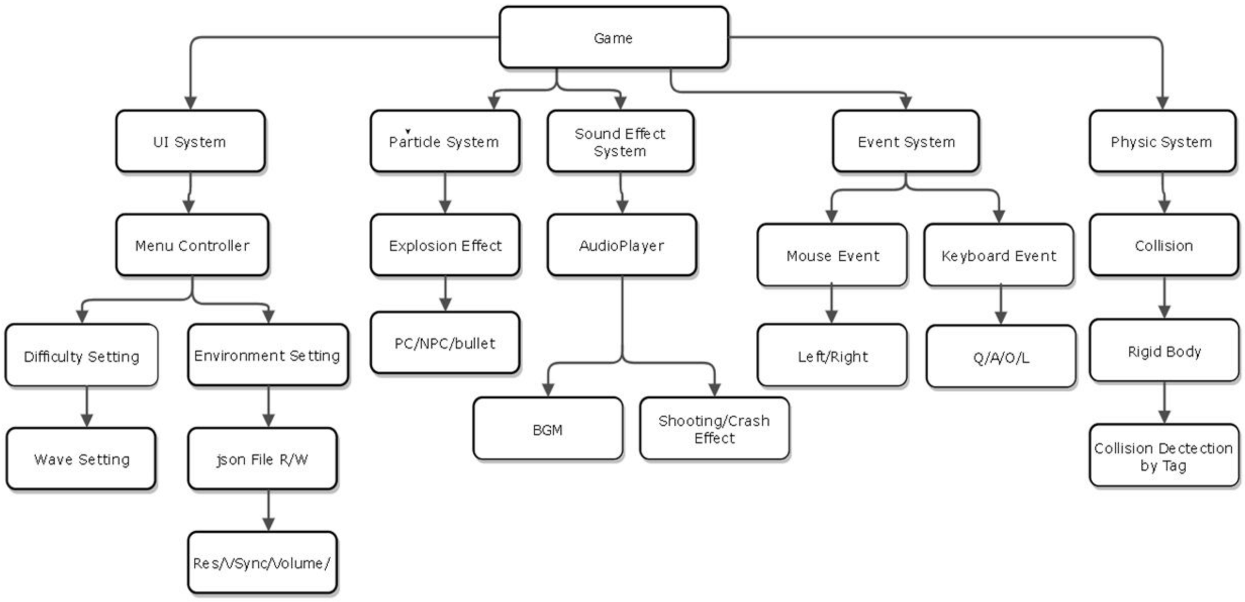
- **System Architecture**

  Framework :



We build the system in Unity platform, with the non player character and player from top to bottom.

Detailed :



This is the detailed architecture version of how we implement in Unity. Basically we have five system as followed.

Within the UI system, we record the user game setting by a json file, with resolution, Vsync, volume and difficulty game attributes.

Particle and sound system are introduced before.

For physics system, we implement our physical behavior above rigid body, and we use Tag to distinguish different object.

● Techniques

1. Singleton

We used the design pattern taught by the professor during the class, as the game program running, there's a static class instance processing in the background, recording static value such as game status, health point, difficulty...etc. also derive functions that need to be used for many separate classes.

2. Menu

We design a menu that provide game settings such as resolution, Vsync, volume ...etc. Everytime after user setting properly and click 'save', the system will automatically generate a json file to record the user's preference setting. Also, the user could select the difficulties and wave number before game start.

3. Collision Events

In the game, we had to handle many conditions about collision, such as bullet collision, hitting obstacles on terrain.

For players, they would be blocked when hitting on other objects in scene and be damaged when being hit by enemies' bullets. Moreover, if players stand on the mud, they would be slowed down. If players meet a tool kit, they would be healthed 20% health.

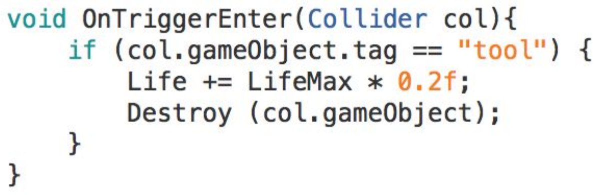
For enemies, they couldn't go through the houses and trees also, and would be damaged by player's bullet, too.

For static objects in scene, each of the objects in scene was tagged. For instance, the houses are tagged as "house." This setting was done before executing the game.

For bullets, each of them would be tagged by the shooter. For example, if the shooter is player, it would be tagged as "playerBullet." According to the tag, the bullet would know if it causes damages to collider or not. Because the setting is dynamic, so it have to be set in scripts.

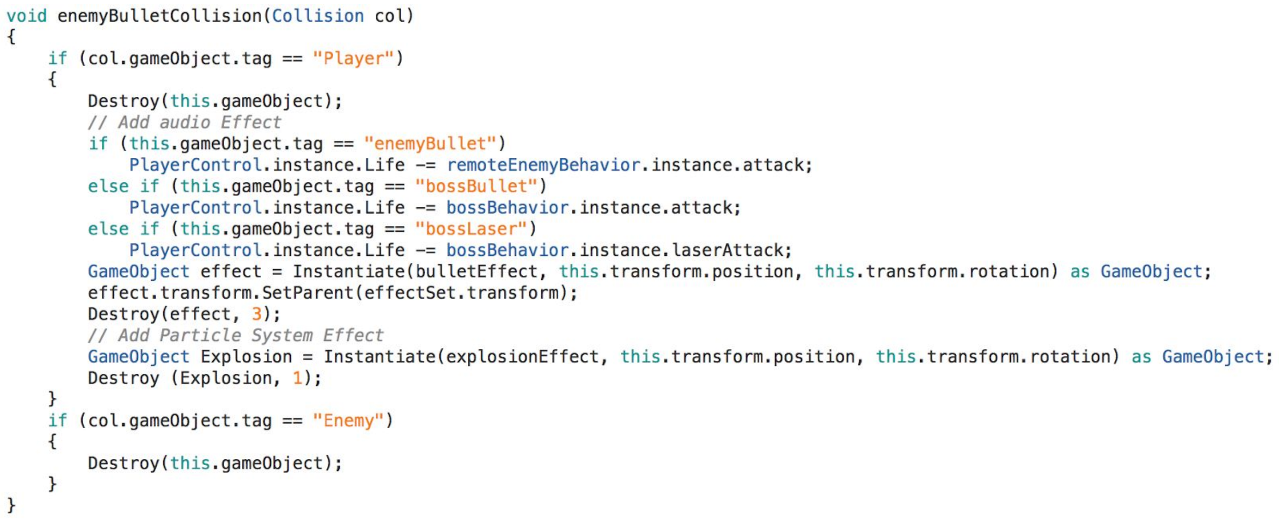
For tool kit, there were some differences between this condition and others. If we used the same way as the above, when we bump into it, it will give a opposite force like physics. So, we figured out the other way, **Trigger**.

It also needs to compare with the tag to do something, but it won't generate any physic condition.



```
void OnTriggerEnter(Collider col){
    if (col.gameObject.tag == "tool") {
        Life += LifeMax * 0.2f;
        Destroy (col.gameObject);
    }
}
```

(a) The codes including trigger comparation

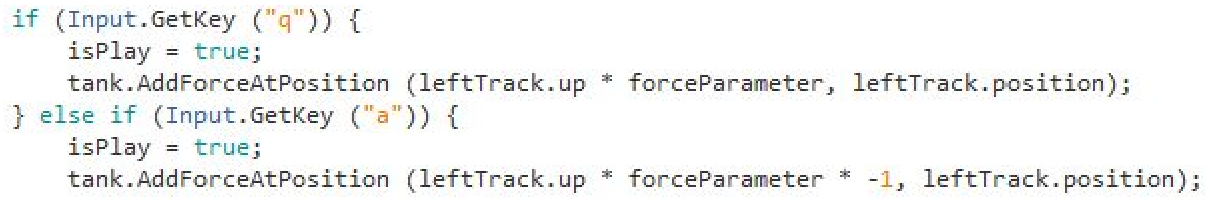


```
void enemyBulletCollision(Collision col)
{
    if (col.gameObject.tag == "Player")
    {
        Destroy(this.gameObject);
        // Add audio Effect
        if (this.gameObject.tag == "enemyBullet")
            PlayerControl.instance.Life -= remoteEnemyBehavior.instance.attack;
        else if (this.gameObject.tag == "bossBullet")
            PlayerControl.instance.Life -= bossBehavior.instance.attack;
        else if (this.gameObject.tag == "bossLaser")
            PlayerControl.instance.Life -= bossBehavior.instance.laserAttack;
        GameObject effect = Instantiate(bulletEffect, this.transform.position, this.transform.rotation) as GameObject;
        effect.transform.SetParent(effectSet.transform);
        Destroy(effect, 3);
        // Add Particle System Effect
        GameObject Explosion = Instantiate(explosionEffect, this.transform.position, this.transform.rotation) as GameObject;
        Destroy (Explosion, 1);
    }
    if (col.gameObject.tag == "Enemy")
    {
        Destroy(this.gameObject);
    }
}
```

(b) The codes including enemies' bullet collision and do damage on player according different type of the bullet.

4. **Tank Tracks' motion**

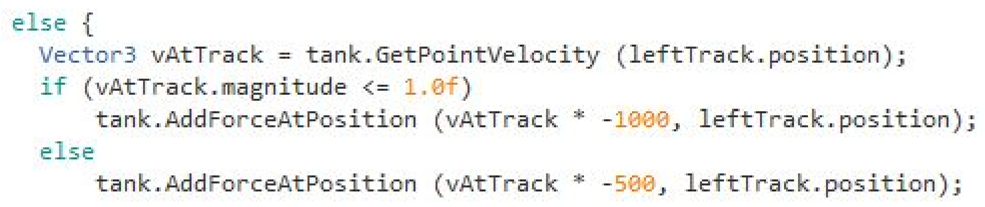
Instead of just use translation to move the tank, we assign force to it. When pressing Q or O, we assign a forward force to the left or the right track, when pressing A or L, we assign a backward force. By doing this, we can go forward or backward when both track has same direction force on it, or turn when only one track has force or two tracks has different direction force.



```
if (Input.GetKey ("q")) {
    isPlay = true;
    tank.AddForceAtPosition (leftTrack.up * forceParameter, leftTrack.position);
} else if (Input.GetKey ("a")) {
    isPlay = true;
    tank.AddForceAtPosition (leftTrack.up * forceParameter * -1, leftTrack.position);
```

(c) The codes to assign force to the moving track.

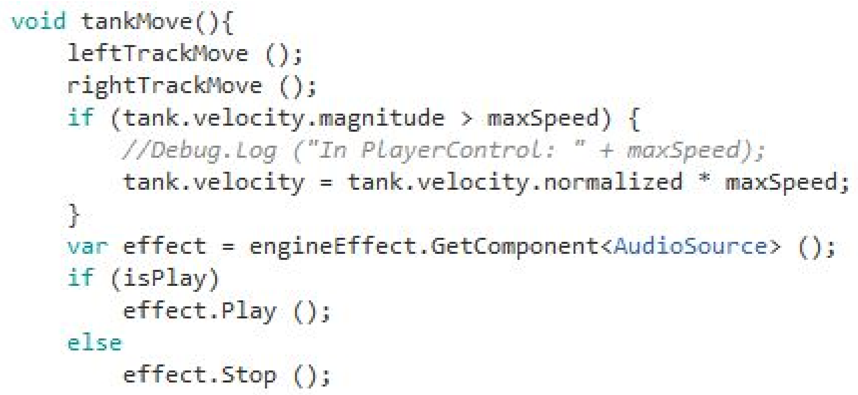
However, due to the incorrect set of friction, when only one track has a force on it, the tank sometimes don't turn but still going straight because the opposite force of friction isn't enough to stop the other track. To fix this, we assign a force to the non-moving track which has a direction opposite to the current point velocity. This will pull the non-moving track back and make the tank turn.



```
else {
  Vector3 vAtTrack = tank.GetPointVelocity (leftTrack.position);
  if (vAtTrack.magnitude <= 1.0f)
      tank.AddForceAtPosition (vAtTrack * -1000, leftTrack.position);
  else
      tank.AddForceAtPosition (vAtTrack * -500, leftTrack.position);
```

(d) The codes that assign opposite force to non-moving track.

At last, if we just simply use force to move the tank, the tank will go super fast eventually, so we need to set a maximum speed for the tank.



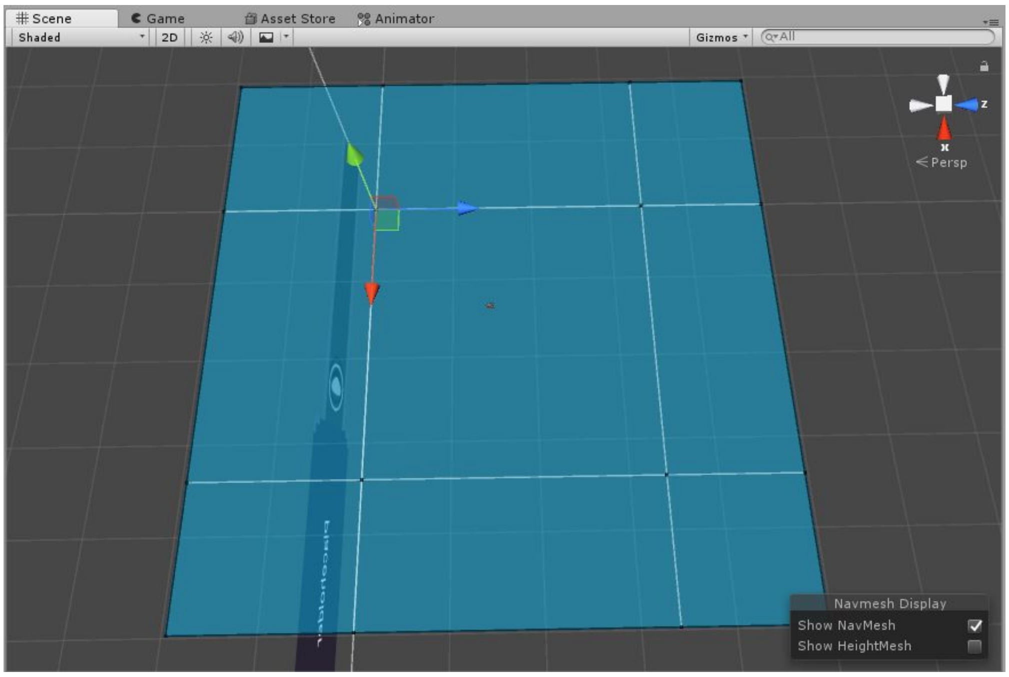
```
void tankMove(){
    leftTrackMove ();
    rightTrackMove ();
    if (tank.velocity.magnitude > maxSpeed) {
        //Debug.Log ("In PlayerControl: " + maxSpeed);
        tank.velocity = tank.velocity.normalized * maxSpeed;
    }
    var effect = engineEffect.GetComponent<AudioSource> ();
    if (isPlay)
        effect.Play ();
    else
        effect.Stop ();
```

(e) The codes to make tank move, including the maximum speed setting.

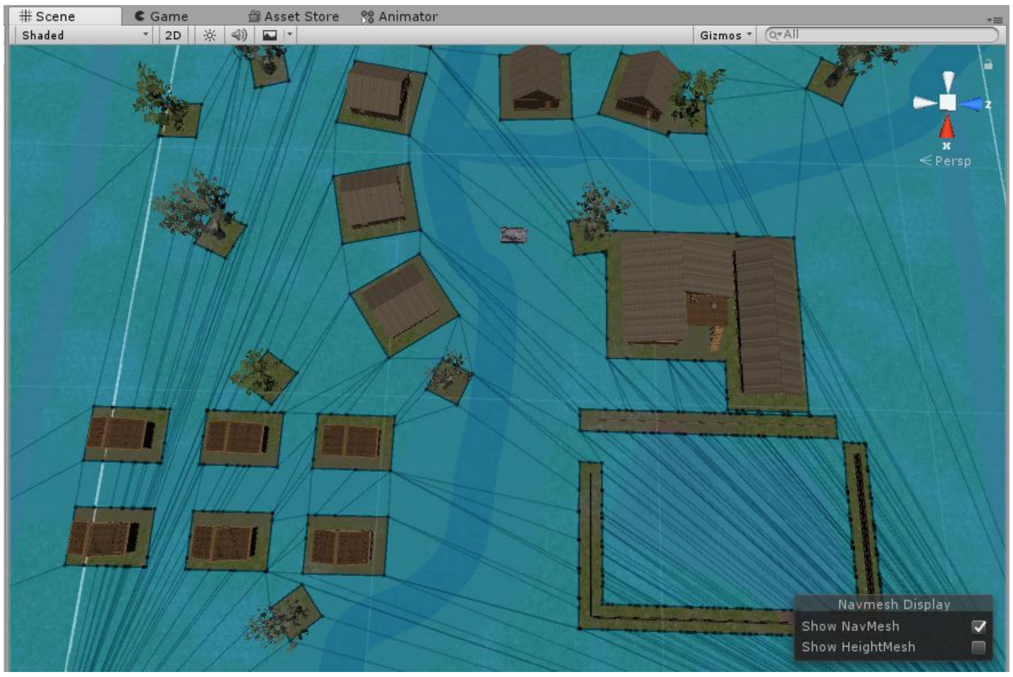
5. Path Finding

We want the enemies to avoid crashing into obstacles and go toward the players, and in Unity, there is a powerful tool called NavMesh. It can scan the whole scene and bake a map of movable places. When we have a NavMesh, we simply add a Nav Mesh Agent to the enemy and set a destination, and it will do the path finding for us.

However, because the NavMesh can only be baked in edit mode, we cannot bake the NavMesh in game. And because in this game we have three different maps, but we simply construct them in the same scene and use visible/invisible to change the map, we cannot acquire three different NavMesh and change it in game. To fix this, we do not bake the NavMesh on the game map, but bake a completely plane terrain where every place is movable. Then, we add a new component Nav Mesh Obstacle on every single house, tree and fence. With this component, the obstacles will carve a hole in the original NavMesh when the obstacles are visible. This is how we can get a different NavMesh in different maps.



(f) The completely plane NavMesh, where the blue part are movable places.



(g) When the map is set to visible, the obstacles will carve holes in the NavMesh.

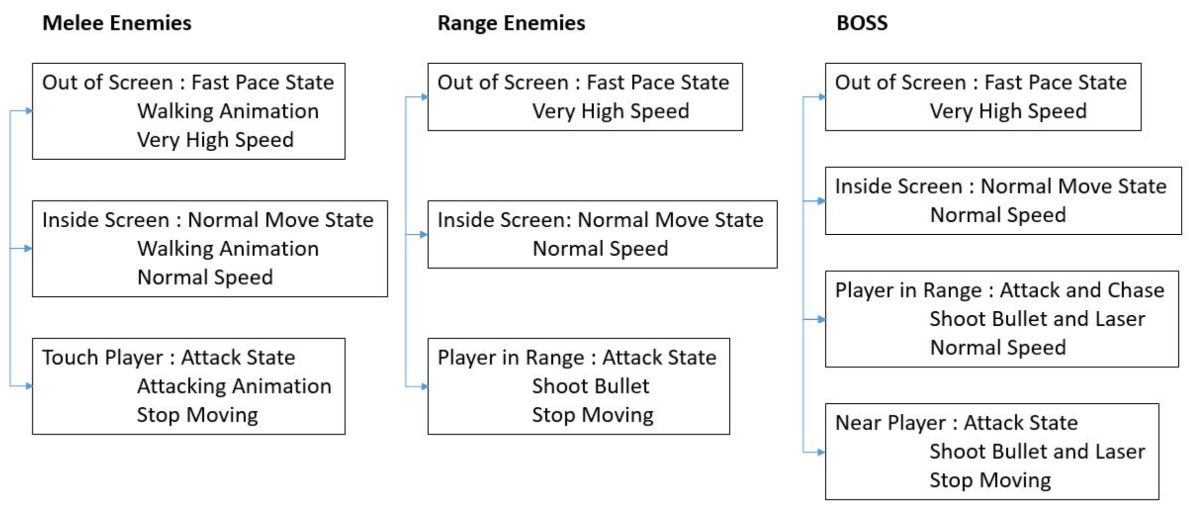
6. Finite State Machine

Every type of enemy has a finite state machine to decide it's movement. The parameter to decide which state to enter is mostly the distance between the enemy and the players.

For the melee attack enemy, when it is far away from the players, which means it is outside the screen, it will enter the state that move super fast towards the players because we don't want the players to wait for too long for the enemy to come. When it is close to the players, which almost inside the screen, it will enter the state that move in normal speed, a walking animation is also applied. When it is right beside the players, it will enter the attack state, which it will stop moving and start the attack animation while dealing damage to the players.

For the ranged attack enemy, when it is far away, it will also move in a super fast speed to come close faster. When it is inside the screen, it will turn back to normal speed. When it is close enough to the players, it will stop moving and start shooting.

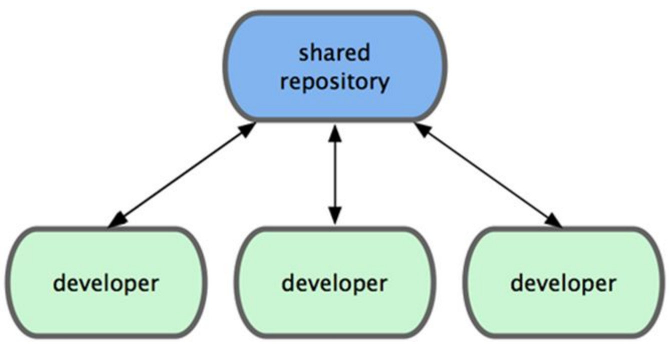
For the boss enemy, when it is far away, it will also move in super fast speed. When it is inside the screen, it will move in normal speed. When the players is in it attack range, which is longer than the ranged enemy, it will start shooting and also moving towards the player. At last, when it is close enough, it will stop moving and keep attacking.


Melee Enemies
Out of Screen : Fast Pace State
Walking Animation
Very High Speed
Inside Screen : Normal Move State
Walking Animation
Normal Speed
Touch Player : Attack State
Attacking Animation
Stop Moving
Range Enemies
Out of Screen : Fast Pace State
Very High Speed
Inside Screen: Normal Move State
Normal Speed
Player in Range : Attack State
Shoot Bullet
Stop Moving
BOSS
Out of Screen : Fast Pace State
Very High Speed
Inside Screen : Normal Move State
Normal Speed
Player in Range : Attack and Chase
Shoot Bullet and Laser
Normal Speed
Near Player : Attack State
Shoot Bullet and Laser
Stop Moving

(h) The FSM state definition.

7. Git

In the project, we used gitlab to be our platform. We could work together parallel, and modify and push to the cloud repository.



- **Contributions**

  •侯家元：

  •GamePlay Design, Scene, Path Finding, Enemy Generator, Debug

  •吳玉辰：

  •Physics, Player Control, Enemies Attack, Particle Systems, Debug

  •簡立哲：

  •Git server, Design Pattern, UI, Sound Effects, Scene Manager, Debug

- **Milestones**

Work	Time Stamp	Time (hrs)
Structure Design	10/20 17:00~	6
Scene / Character construct	10/22 ~ 10/30	17
Player Control	11/1 ~ 11/5	10
Physics	11/1 ~ 11/20	22
Enemy Generator	11/15 ~ 12/2	13
UI, Game Attribute	10/25 ~ 12/20	15
Particle System	12/4 ~ 1/10	17
Sound	10/20 ~ 11/20	4
Debug / Build	12/20 ~ 1/12	17
Total	10/20 ~ 1/12	121

- **Conclusion**

  We've spent much time on this project, and wrote 1,500 lines of codes approximately. We learned lots of skills during programming and debugging. Moreover, we know how to cooperate with teammates and implement same project with different function at the same time, which I consider the most important thing that we have to learn for our future career.