Coder un dogfight en 3D avec Unity

Gaëtan Blaise-Cazalet – Codeur et GD économique

1. Ouvrir Unity, créer un projet 3D

2. Sol et avion

Créer le sol : clic droit sur la scene, 3D object, plane. Reset sa position. Changer sa scale x et z pour 500. Renommer en sol.

Créer une capsule : clic droit sur la scene, 3D object, capsule. Ce sera l'avion. Reset la position, mettre 10 en Y. Mettre la rotation Z à 90. Renommer en "avion".

En profiter pour régler les contrôles avec Edit/Preferences/Keys. Régler aussi les outils pour les placer sur AZERT

Créer des ailes pour l'avion en lui créer un enfant Cube, que l'on écrase. Mettre la position du cube à 0 puis le placer correctement pour qu'il ressemble à des ailes. Les ailes bougent avec la capsule.

Mettre la taille de l'avion à 5 10 5. Mettre la taille des ailes à 0.05 0.5 6.

Dans la fenêtre assets, créer un material. Lui donner une couleur (albedo) plus agréable que le blanc. Renommer en "FloorMaterial". Déplacer sur le GameObject sol.

Sauvegarder la scene.

Se déplacer autour de l'avion pour prendre un bon point de vue. Sélectionner la caméra. Faire GameObject / Align with view pour donner le bon angle à la camera.

3. Texture pour se repérer.

Sous paint, créer un carré de 256 * 256. Donner une bordure en haut et à gauche.

Importer dans unity.

Glisser la texture sur l'albedo. Mettre tiling à 50 * 50.

4. Faire avancer l'avion

Ajouter un composant RigidBody à l'avion. Tester.

Enlever la gravité.

Créer un composant script ShipMovement sur l'avion.

Déclarer le rigidbody dans le script :

```
Rigidbody rbody;
```

Le récupérer pour pouvoir l'utiliser dans Start() :

```
rbody = GetComponent<Rigidbody>();
```

Donner une vitesse dans start :

```
rbody.velocity = new Vector3(-10f, 0, 0);
```

On aimerait pouvoir régler la vitesse sans aller dans le code. Il suffit de créer un membre public Speed:

```
    public Vector3 Speed;
```

Et de rendre la vélocité égale à cette vitesse.

```
    rbody.velocity = speed;
```

On voit que la vitesse est désormais éditable dans Unity.

5. Contrôles : rotation de l'avion

On veut que l'avion tourne sur lui-même dans un sens ou dans l'autre quand on appuie sur Q ou D.

```
if(Input.GetKey(KeyCode.Q))
{
    // Rotation sur la gauche
}
if (Input.GetKey(KeyCode.D))
{
    // Rotation sur la droite
}
```

Le code pour la rotation :

```
transform.Rotate(Vector3.up * rollSpeed * Time.deltaTime);
```

Avec rotateSpeed une nouvelle variable publique

Maintenant qu'on a la rotation le long de l'axe de l'avion, on veut le faire tourner vers le haut et vers le bas avec Z et S. Cela nous permettra de le controler l'avion. Utiliser cette fois l’axe Vector3.forward.

```
if (Input.GetKey(KeyCode.S))
{
    transform.Rotate(Vector3.forward * -pitchSpeed * Time.deltaTime);
}
if (Input.GetKey(KeyCode.Z))
{
    transform.Rotate(Vector3.forward * pitchSpeed * Time.deltaTime);
}
```

Problème : l'avion n’obéit pas du tout à sa rotation. Il continue à avancer selon l'axe X.

6. L'avion avance selon son angle

Calcul mathématique compliqué. Unity nous simplifie la tache.

```
rbody.velocity = speed * (transform.rotation * Vector3.up) * Time.deltaTime;
```

Quand on fait transform.rotation * Vector3.up, il se déroule un beau calcul matriciel que l'on ne gère pas, et qui permet au devant de l'avion de correspondre à l'angle choisi.

7. L'avion est à peu près contrôlable, mais il n'est pas suivi par la caméra.

Première solution : mettre la caméra comme enfant du vaisseau.

Ca marche mais c'est moche. Sensations de jeu pas intéressantes.

Deuxième solution, un petit peu de code. On voudrait que la caméra "rattrape" la position et l'angle de l'avion. Créer un script SmoothFollow pour la caméra.

```
public class SmoothFollow : MonoBehaviour
{
    public Transform target;

    public Vector3 velocity = Vector3.one;
    public Vector3 defaultDistance;
    public float distance;


    void LateUpdate()
    {
        if (!target) return;

        Vector3 toPosition = target.position + (target.rotation * defaultDistance);
        Vector3 currentPosition = Vector3.SmoothDamp(transform.position, toPosition, ref velocity, distance);
        transform.position = currentPosition;

        transform.LookAt(target);
    }
}
```

Propositions de valeurs :

Default distance 10 10 0

Distance 1.2

8. Tirer

Créer un objet vide enfant de l'avion, le nommer ShotSpawnPoint et le placer sous l'avion.

Créer un Cylindre, le nommer shoot. Lui donner une taille de 1 5 1.

Faire glisser shoot dans la barre des assets. L'objet devient un prefab. On peut le supprimer de la scène.

Créer un script ShipShooting dans l'avion. On veut tirer en appuyant sur espace (dans la méthode Update). Pour tirer, on instancie notre objet de tir avec Instantiate(shoot, position, rotation);

```
void Update () {
        if (Input.GetKey(KeyCode.Space))
        {
            Instantiate(shoot, position, rotation);
        }
}
```

Mais ne compile pas. Il faut récupérer shoot, la position, la rotation.

Il faut déclarer cet objet shoot, et le recupérer depuis l'éditeur.

```
public GameObject shoot;
```

On peut maintenant glisser le prefab dans le slot shoot du script.

On veut la position de l'objet enfant (le deuxième dans le hiérarchie) et la rotation de l'avion.

```
Instantiate(shoot, transform.GetChild(1).position, transform.rotation);
```

Ca marche, mais les tirs ne bougent pas. On va leur ajouter un script qui leur donne une vélocité. Qui doit être supérieure à celle de l'avion !

Créer un script Shoot dans le prefab Shoot. Créer aussi un Rigidbody pour qu'on puisse lui donner une vélocité. Ce rigidbody n'est pas soumis à la gravité.

Récupérer le rigid body dans le script et lui donner une vélocité, paramétrée par une vitesse publique.

```
    public float speed;


    Rigidbody rbody;

      void Start () {
        rbody = GetComponent<Rigidbody>();
        rbody.velocity = speed * (transform.rotation * Vector3.up) * Time.deltaTime;
    }
```

Trois problèmes !

1. Les tirs s'éparpillent parce qu'ils se rentrent dedans.

2. Les tirs restent indéfiniment dans le jeu. Même si les objets sont simples, on risque de saturer la mémoire.

3. Il y a trop de tirs. On génère une espèce de spaghetti.

9. Régler les problèmes des tirs

Ca commence à être le bazar dans nos assets ! Créet un dossier script et mettre tous les scripts dedans. Unity gère l'association avec les GameObjects.

Problème 1 : les tirs s'éparpillent.

C'est parce que la physique du jeu fait interagir les tirs. Cocher isTrigger dans le composant collider du tir.

Problème 2 : les tirs demeurent.

Ils ont une vitesse rapide donc deviennent très vite petits. On va leur donner une durée de vie de 3 secondes. Dans le script, ajouter :

```
Destroy(gameObject, 3f);
```

On peut aussi créer une variable lifetime publique et l'utiliser dans le destroy. Comme ca vous pouivez régler la durée de vie des tirs depuis unity !

Probleme 3 : les spagetthis.

La solution est de laisser s'écouler du temps entre chaque tir. Occupons nous de ShipShooting.

On va créer un cooldown et un compteur qui s'incrémente avec le temps entre chaque frame. Quand ce temps depasse la durée du cooldown, on peut instancier un tir.

```
public float cooldown = 0.5f;


float counter = 1f;

void Update () {
        if (Input.GetKey(KeyCode.Space))
        {
            if(counter > cooldown)
            {
                Instantiate(shoot, transform.GetChild(1).position, transform.rotation);
                counter = 0f;
            }
        }
        counter += Time.deltaTime;
}
```

Maintenant on aimerait avoir une petite amélioration. On aimerait un viseur devant l'avion (mais qui ne doit pas interagir avec la physique). Et on aimerait que les tirs ne partent non pas d'une mais de deux sources. Gaucher droite gauche droite. Ça serait plus joli et plus réaliste.

10. Améliorer les tirs

Pour le viseur, créer un sprite et la placer loin devant l'avion, face à l'avion. Choisir un sprite dans ceux proposés par défaut.

Par exemple :

Position 0 10 0

Rotation 90 0 0

Taille 2 2 1

Pour le second tir, dupliquer l'objet ShootSpawnPoint avec Ctrl + D. Attention à leur ordre dans la hiérarchie. Cet ordre est important pour le script.

On les voit pas trop, sauf en les sélectionnant dans la hiérarchie. On va ajouter des guizmos pour voir cet objets invisibles. Il suffit de cliquer sur l'icone a coté du nom dans linspecteur, et d'attribuer un petit rond ou un petit losange. Maintenant on peut les placer facilement.

Dans notre script, on veut signaler si c'est le spawn de gauche qui a tiré ou si c'est le droite. Si c'est le gauche on tire avec le droite (et on dit que c'est le gauche qui a tiré) et si c'est le droite on tire avec le gauche.

On crée la variable :

```
bool spawnLeftShot = false;
```

On l'utilise dans le update. A vous !

```
void Update () {
        if (Input.GetKey(KeyCode.Space))
        {
            if(counter > cooldown)
            {
                if(spawnLeftShot)
                {
                    Instantiate(shoot, transform.GetChild(1).position, transform.rotation);
                    spawnLeftShot = false;
                }
                else
                {
                    Instantiate(shoot, transform.GetChild(0).position, transform.rotation);
                    spawnLeftShot = true;
                }
                counter = 0f;
            }
        }
        counter += Time.deltaTime;
}
```

11. Faire exploser les tirs quand ils rencontrent un obstacle (le sol)

Prepation de l'explosion. Créer un Particule system. Mettre une shape sphere, augmenter l'émission et la vitesse des particules. Jouer avec les paramètres pour que ca fasse une petite explosion.

Renommer en ShootExplosion. En faire un Prefab.

Principe du trigger : on peut déclencher un script quand il se passe quelque chose.

Créer un box collider assez fin autour du sol. En faire un trigger.

Créer un script ShootTarget qui va s'activer quand un trigger passera dedans. Lui associer un objet public shootExplosion.

Instancier cet objet quand un trigger avec le tag shoot (qu'il faut créer et associer à l'objet shoot) passe à travers.

```
public GameObject shootExplosion;

private void OnTriggerEnter(Collider other)
{
    if(other.tag == "Shoot")
    {
        Instantiate(shootExplosion, other.transform.position, other.transform.rotation);
        Destroy(other.gameObject);
    }

}
```

Problème : les explosions demeurent.

Prévoir leur destruction au bout d'une seconde avec :

```
Destroy(explosion, 1f);
```

On peut récupérer la référence de l'objet à detruire à partir d'Instantiate.

```
public GameObject shootExplosion;

private void OnTriggerEnter(Collider other)
{
    if(other.tag == "Shoot")
    {
        GameObject explosion = Instantiate(shootExplosion, other.transform.position, other.transform.rotation);
        Destroy(other.gameObject);
        Destroy(explosion, 1f);
    }

}
```

12. Objets à détruire

Créer un cube qui soit à la fois une shooting target, et qui ait un autres script avec un certain nombre de point de vie.

Quand le cube est touché par un tir (dans le script shooting target), on appelle le composant qui gère la vie pour retirer des points de vie.

```
Life targetLife = GetComponent<Life>();
if(targetLife != null)
{
    targetLife.Hurt(1);
}
```

A zero points de vie, le composant vie déclenche la mort du cube, avec une explosion.

Attention ! Le cube ne doit pas mourir plusieurs fois :)

```
public class Life : MonoBehaviour {

    public int hp;
    public GameObject deathExplosion;
    bool hasDied = false;

    public void Hurt(int lostHp)
    {
        hp -= lostHp;
        if(hp <= 0 && !hasDied)
        {
            Die();
            hasDied = true;
        }
    }

    void Die()
    {
        GameObject explosion = Instantiate(deathExplosion, transform.position, transform.rotation);
        Destroy(gameObject);
        Destroy(explosion, 1.5f);
    }

}
```

13. Génération de cibles

Commencer par faire du cube cible un prefab. C'est le bazar, donc créer un dossier Prefab pour les ranger, ainsi qu'un dossier Vfx pour les explosions.

Modifier la taille du plateau pour qu'il soit plus petit. 200 * 200.

Faire en sorte que quand un cube est chargé (méthode awake), il choisisse une direction au hasard, et se déplace dans cette direction à partir d'une extrémité du plateau.

Fonction Awake :

```
void Awake () {

}
```

Choix au hasard d'une direction. On utilise un enum pour indiquer plus facilement la direction.

```
enum Direction
{
    Left, Down, Right, Up
}
```

Dans Awake() :

```
    int direction = Random.Range(0, 4);
    switch (direction)
    {
        case (int)Direction.Left:
            case (int)Direction.Left:

                break;
            break;
        case (int)Direction.Down:

            break;
        case (int)Direction.Right:

            break;
        case (int)Direction.Up:

            break;
    }
```

Pour chaque direction, on veut :

- Mettre la cible sur un bord de l'ecran

- Faire aller la cible vers l'autre bord.

Solution pour la coté gauche :

```
transform.position = new Vector3(1000, transform.lossyScale.y / 2, Random.Range(-1000f, 1000f));
rbody.velocity = new Vector3(-speed, 0, 0);
```

Solution complete :

```
public float speed;
public GameObject boardObject;

Rigidbody rbody;

  void Awake () {
    int direction = Random.Range(0, 4);
    rbody = GetComponent<Rigidbody>();
    switch (direction)
    {
        case (int)Direction.Left:
            transform.position = new Vector3(1000, transform.lossyScale.y / 2, Random.Range(-1000f, 1000f));
            rbody.velocity = new Vector3(-speed, 0, 0);
            break;
        case (int)Direction.Down:
            transform.position = new Vector3(Random.Range(-1000f, 1000f), transform.lossyScale.y / 2, -1000);
            rbody.velocity = new Vector3(0, 0, speed);
            break;
        case (int)Direction.Right:
            transform.position = new Vector3(-1000, transform.lossyScale.y / 2, Random.Range(-1000f, 1000f));
            rbody.velocity = new Vector3(speed, 0, 0);
            break;
        case (int)Direction.Up:
            transform.position = new Vector3(Random.Range(-1000f, 1000f), transform.lossyScale.y / 2, 1000);
            rbody.velocity = new Vector3(0, 0, -speed);
            break;
    }

}
```

On va maintenant utilise une fonction de unity, la coroutine, pour generer régulièrement des cibles.

Créer un objet vide GameController dans lequel on met un script GameController.

Ce GameController a un GameObject public Target, qu'il peut générer grace à la méthode suivante :

```
IEnumerator SpawnTarget()
{
    while(true)
    {
        Instantiate(target);
        yield return new WaitForSeconds(0.5f);
    }
}
```

Cette méthode s’exécute en boucle toutes les demi secondes.

Pour lancer le cycle, on commence la coroutine dans Start() :

```
void Start () {
  StartCoroutine(SpawnTarget());
}
```

Enlever la gravité des cubes pour qu'ils bougent mieux.

14. Problèmes de notre génération de cubes.

Les cubes existent indéfiniment. On pourrait leur donner une durée de vie, mais on va faire autre chose. On va les supprimer quand ils sortent du plateau.

Augmenter legèrement les tailles x et y du boxcollider du plateau. 10 -> 10.01.

Ajouter un tag Target aux cubes.

Ajouter un script TargetDestructor au plateau, sur le meme modele que ShootTarget, mais avec la méthode OnTriggerExit.

```
private void OnTriggerExit(Collider other)
{
    if(other.tag == "Target")
    {
        Destroy(other.gameObject);
    }
}
```

Deuxieme probleme : il est difficile d'ajuster la trajectoire pour viser les cubes. On va ajouter les palonniers pour améliorer le contrôle de l'avion.

Faire tourner l'avion sur le dernier axe quand on appuie sur les fleches gauche et droite.

```
if (Input.GetKey(KeyCode.LeftArrow))
{
    transform.Rotate(Vector3.right * -yawSpeed * Time.deltaTime);
}
if (Input.GetKey(KeyCode.RightArrow))
{
    transform.Rotate(Vector3.right * yawSpeed * Time.deltaTime);
}
```

Option : accélération et decelleration (jusqu'à une vitesse max et une vitesse min) quand on appuie sur haut et bas.