Coder un platformer 2.5D avec Unity

Gaëtan Blaise-Cazalet – Professeur de programmation

1. Ouvrir Unity, créer un projet 3D

2. Joueur

Créer un cube, le nommer Player. Supprimer le BoxCollider. Créer un material pour le joueur et l'assigner.

Créer un script Player. Créer un script Controller.

Script Player :

```
[RequireComponent(typeof (Controller))]
public class Player : MonoBehaviour {

    Controller controller;

    void Start () {
        controller = GetComponent<Controller>();
    }
}
```

Script Controller :

```
[RequireComponent(typeof (BoxCollider2D))]
public class Controller : MonoBehaviour {

    BoxCollider2D collider;

    void Start () {
        boxCollider = GetComponent<BoxCollider2D>();
    }
}
```

Ajouter le script Player au cube. Cela ajoute automatiquement les autres composants.

Créer un Layer Player, l’assigner au joueur, ainsi que le tag Player.

3. Tracer des rays sous le cube

Dans le script Controller.

Ajouter une struct qui donnera l'origine des rays, et la variable membre associée :

```
    RaycastOrigins raycastOrigins;
    struct RaycastOrigins {
        public Vector2 topLeft, topRight, bottomLeft, bottomRight;
    }
```

Ajouter les variables :

```
    [SerializeField]
    int horizontalRayCount = 4;

    [SerializeField]
    int verticalRayCount = 4;

    float horizontalRaySpacing;
    float verticalRaySpacing;
    const float skinWidth = 0.015f;
```

Ajouter des méthodes pour mettre à jour les rays :

```
    void UpdateRaycastOrigins() {
        Bounds bounds = boxCollider.bounds;
        bounds.Expand(skinWidth * -2);

        raycastOrigins.bottomLeft = new Vector2(bounds.min.x, bounds.min.y);
        raycastOrigins.bottomRight = new Vector2(bounds.max.x, bounds.min.y);
        raycastOrigins.topLeft = new Vector2(bounds.min.x, bounds.max.y);
        raycastOrigins.topRight = new Vector2(bounds.max.x, bounds.max.y);
    }
```

```
void CalculateRaySpacing() {
    Bounds bounds = boxCollider.bounds;
    bounds.Expand(skinWidth * -2);

    horizontalRayCount = Mathf.Clamp(horizontalRayCount, 2, int.MaxValue);
    verticalRayCount = Mathf.Clamp(verticalRayCount, 2, int.MaxValue);
    horizontalRaySpacing = bounds.size.y / (horizontalRayCount - 1);
    verticalRaySpacing = bounds.size.x / (verticalRayCount - 1);
}
```

Enfin, créer une méthode Update qui dessine les rays sur la scène :

```
void Update() {
    UpdateRaycastOrigins();
    CalculateRaySpacing();

    for(int i = 0; i < verticalRayCount; i++) {
        Debug.DrawRay(raycastOrigins.bottomLeft + Vector2.right * verticalRaySpacing * i, Vector2.up * -2, Color.red);
    }
}
```

Tester le jeu en mode scène. Vous voyez les raycasts de dessiner sous le cube.

4. Créer le sol, mouvement du joueur

Ajouter un cube pour figurer le sol. Remplacer le BoxCollider par un BoxCollider2D.

Ajouter dans Player :

```
    float gravity = -20;
    Vector3 velocity;
```

```
    void Update () {
        velocity.y += gravity * Time.deltaTime;
        controller.Move(velocity * Time.deltaTime);
    }
```

Ajouter dans Controller la fonction Move :

```
    public void Move(Vector3 velocity)
    {



        transform.Translate(velocity);
    }
```

Déplacer l'appel à CalculateRaySpacing dans Start. Déplacer l'appel à UpdateRayCastOrigins dans Move.

5. Collisions verticales

Ajouter un champ collisionMask, de type LayerMask :

```
    [SerializeField]
    LayerMask collisionMask;
```

Créer une méthode VerticalCollisions. L'appeler dans Move quand la vélocité verticale est différente de 0.

```
        if(velocity.y != 0) {
            VerticalCollisions(ref velocity);
        }
```

```
    void VerticalCollisions(ref Vector3 velocity) {
        float rayDirectionY = Mathf.Sign(velocity.y);
        float rayMagnitude = Mathf.Abs(velocity.y) + skinWidth;

        for(int i = 0; i < verticalRayCount; i++) {
            Vector2 rayOrigin = (rayDirectionY == -1)? raycastOrigins.bottomLeft : raycastOrigins.topLeft;
            rayOrigin += Vector2.right * (verticalRaySpacing * i);
            RaycastHit2D hit = Physics2D.Raycast(rayOrigin, Vector2.up * rayDirectionY, rayMagnitude, collisionMask);

            Debug.DrawRay(rayOrigin, Vector2.up * rayDirectionY * rayMagnitude, Color.red);
            if (hit) {
                velocity.y = (hit.distance - skinWidth) * rayDirectionY;
                rayMagnitude = hit.distance; // Limit future rays magnitudes
            }
        }
    }
```

Maintenant, ajouter un nouveau Layer, appelé Obstacles. Assigner le layer à l'objet qui représente le sol. Mettre le collisionMask du script Controller du joueur sur Obstacles. Tester le jeu, la collision sur le sol devrait fonctionner.

6. Collisions horizontales

Ajouter une variable dans Player :

```
    float moveSpeed = 6f;
```

Et mettre à jour le Update :

```
    void Update () {
        velocity.x = moveSpeed * Time.deltaTime;
        velocity.y += gravity * Time.deltaTime;
        controller.Move(velocity * Time.deltaTime);
    }
```

Dans Controller, copier et adapter la fonction de collision.

```
    void HorizontalCollisions(ref Vector3 velocity) {
        float rayDirectionX = Mathf.Sign(velocity.x);
        float rayMagnitude = Mathf.Abs(velocity.x) + skinWidth;

        for(int i = 0; i < horizontalRayCount; i++) {
            Vector2 rayOrigin = (rayDirectionX == -1)? raycastOrigins.bottomLeft : raycastOrigins.bottomRight;
            rayOrigin += Vector2.up * (horizontalRaySpacing * i);
            RaycastHit2D hit = Physics2D.Raycast(rayOrigin, Vector2.right * rayDirectionX, rayMagnitude, collisionMask);

            Debug.DrawRay(rayOrigin, Vector2.right * rayDirectionX * rayMagnitude, Color.red);
            if (hit) {
                velocity.x = (hit.distance - skinWidth) * rayDirectionX;
                rayMagnitude = hit.distance; // Limit future rays magnitudes
            }
        }
    }
```

La fonction Move finale sera :

```
    public void Move(Vector3 velocity)
    {
        UpdateRaycastOrigins();
        if(velocity.x != 0) {
            HorizontalCollisions(ref velocity);
        }
        if(velocity.y != 0) {
            VerticalCollisions(ref velocity);
        }
        transform.Translate(velocity);
    }
```

Ajouter un obstacle horizontal pour tester le système.

On remarque que la magnitude des rays orientés vers le bas grandit. On va régler ce problème.

7. Lieu de la collision et réparation de la gravité

Ajouter une public struct CollisionInfo dans le Controller.

```
public struct CollisionInfo {
    public bool above, below, left, right;

    public void Reset() {
        above = below = left = right = false;
    }
}
```

Ajouter un membre collision, de type CollisionInfo, qui permettra de savoir quelle collision est valable en ce moment. Lui donner la Property associée.

```
public CollisionInfo Collisions {
    get {
        return collisions;
    }
}
CollisionInfo collisions;
```

Les collisions sont réinitialisées à chaque fois que Move est appelée.

```
public void Move(Vector3 velocity)
{
    collisions.Reset();
    …
}
```

Dans Horizontal et VerticalCollisions, on active la collision quand les rays touchent un objet sur le coté correspondant.

```
    void HorizontalCollisions(ref Vector3 velocity) {
        …
            if (hit) {
                …
                collisions.left = (rayDirectionX == -1);
                collisions.right = (rayDirectionX == 1);
            }
        }
    }

    void VerticalCollisions(ref Vector3 velocity) {
        …
            if (hit) {
                …
                collisions.below = (rayDirectionY == -1);
                collisions.above = (rayDirectionY == 1);
            }
        }
    }
```

Maintenant, nous pouvons régler le problème. Dans Player on annule la vélocité verticale quand on a une collision verticale.

```
    void Update () {
        if(controller.Collisions.above || controller.Collisions.below) {
            velocity.y = 0;
        }
        …
    }
```

8. Sauter

Ajouter une variable jumpVelocity dans Player:

```
    float jumpVelocity = 8f;
```

Dans Update, ajoute le code :

```
        if (Input.GetKeyDown(KeyCode.Space) && controller.Collisions.below) {
            velocity.y += jumpVelocity;
        }
```

On va utiliser des variables plus faciles à gérer :

```
    [SerializeField] float jumpHeight;
    [SerializeField] float jumpTimeToApex;
```

Pour obtenir la gravité. En physique :

*Deplacement sur une unité de temps = vitesse initiale * unité de temps + (accéleration * unité de temps au carré) / 2*

*jumpHeight = 0 + gravity * jumpTimeToApex * jumpTimeToApex / 2*

*gravity = 2 * jumpHeight / (jumpTimeToApex * jumpTimeToApex)*

Pour obtenir la jumpVelocity :

*Vélocité finale – vélocité courante = accélération * unité de temps*

*jumpVelocity = gravity * jumpTimeToApex*

On peut maintenant donner une valeur initiale à gravity et à jumpVelocity :

```
    void Start () {
        controller = GetComponent<Controller>();
        gravity = -2 * jumpHeight / (jumpTimeToApex * jumpTimeToApex);
        jumpVelocity = Mathf.Abs(gravity) * jumpTimeToApex;
```

```
    }
```

9. Déplacement du joueur

On veut créer un moyuvement d'accélération doux. Unity possède quelques fonctions pratiques pour cela. Première chose : on veut une accélération différente en l'air et au sol. Ajouter dans Player :

```
    [SerializeField] float accelerationTimeAirborn = 0.2f;
    [SerializeField] float accelerationTimegrounded = 0.1f;
```

Ajouter aussi une variable qui servira de vitesse de transition :

```
    float velocityXSmooth;
```

Pour ce qui est du mouvement, on vide une vitesse maximum et on utilise une fonction de unity pour atteindre de manière élégante cette vitesse :

```
oid Update () {
        …
        // Jump
        if (Input.GetKeyDown(KeyCode.Space) && controller.Collisions.below) {
            velocity.y += jumpVelocity;
        }
        // Move
        Vector2 input = new Vector2( Input.GetAxisRaw("Horizontal"), Input.GetAxisRaw("Vertical"));
        float targetVelocityX = input.x * moveSpeed;
        float accelerationTime = (controller.Collisions.below? accelerationTimegrounded : accelerationTimeAirborn);
        velocity.x = Mathf.SmoothDamp(velocity.x, targetVelocityX, ref velocityXSmooth, accelerationTime);

        velocity.y += gravity * Time.deltaTime;
        controller.Move(velocity * Time.deltaTime);
    }
```

10. Pentes

On va détecter les collisions sur les pentes avec les ray casts horizontaux. L'angle de la pente est égal à l'angle entre la normale à la pente et la verticale.

```
if (hit) {
        float slopeAngle = Vector2.Angle(hit.normal, Vector3.up);
        ...
      }
```

On ajoute une variable pour déterminer l'angle maximum escaladable :

```
  [SerializeField] int maxClimbAngle = 80;
```

On crée une fonction pour escalader la pente :

```
      if (hit) {
        float slopeAngle = Vector2.Angle(hit.normal, Vector3.up);
        if(slopeAngle <= maxClimbAngle && i == 0) {
          ClimbSlope(ref velocity, slopeAngle);
        }
        ...
      }
```

```
  void ClimbSlope(ref Vector3 velocity, float slopeAngle) {
    float moveDistance = Mathf.Abs(velocity.x);
    velocity.x = Mathf.Cos(Mathf.Deg2Rad * slopeAngle) * moveDistance;
    velocity.y = Mathf.Sin(Mathf.Deg2Rad * slopeAngle) * moveDistance * Mathf.Sign(velocity.x);
  }
```

Sauf qu'on ne peut pas sauter quand on monte la pente.

11. Sauter sur une pente

En l'état on ne peut sauter quand on se tient sur une pente.

Première chose : dire que quand on est sur un pente, on touche le sol :

```
void ClimbSlope(ref Vector3 velocity, float slopeAngle) {
  ...
  collisions.below = true;
}
```

Il faut aussi empêcher la velocity.y de s'annuler quand on est sur une pente.

```
void ClimbSlope(ref Vector3 velocity, float slopeAngle) {
  float moveDistance = Mathf.Abs(velocity.x);
  float climbVelocityY = Mathf.Sin(Mathf.Deg2Rad * slopeAngle) * moveDistance;
  if(velocity.y <= climbVelocityY) {
    velocity.x = Mathf.Cos(Mathf.Deg2Rad * slopeAngle) * moveDistance * Mathf.Sign(velocity.x);
    velocity.y = climbVelocityY;
    collisions.below = true;
  }
}
```

On ajoute une petite optimisation : on ne teste la collision horizontale que si on n'est pas en train d'escalader une pente. Ajoutons donc quelques variables dans nos collisionInfo :

```
public struct CollisionInfo {
  public bool above, below, left, right, climbingSlope;
  public float slopeAngle, slopeAngleOld;

  public void Reset() {
```

```
      above = below = left = right = climbingSlope = false;
      slopeAngleOld = slopeAngle;
      slopeAngle = 0;
    }
  }

  void ClimbSlope(ref Vector3 velocity, float slopeAngle) {
    ...
    if(velocity.y <= climbVelocityY) {
      …
      collisions.climbingSlope = true;
      collisions.slopeAngle = slopeAngle;
    }
  }
```

On ne veut tester le reste des collisions horizontales sur si on est pas en train d'escalader une pente.

```
void HorizontalCollisions(ref Vector3 velocity) {
    .
      if (hit) {
        float slopeAngle = Vector2.Angle(hit.normal, Vector3.up);
        if(slopeAngle <= maxClimbAngle && i == 0) {
          ClimbSlope(ref velocity, slopeAngle);
        }

        if(!collisions.climbingSlope || collisions.slopeAngle > maxClimbAngle) {
          velocity.x = (hit.distance - skinWidth) * rayDirectionX;
          ...
          collisions.right = (rayDirectionX == 1);
        }
      }
    }
  }
```

12. Résolution de bugs

Supprimer l'écart entre la pente et le joueur. Dans HoizontalCollisions()

```
      if(slopeAngle <= maxClimbAngle && i == 0) {

         float distanceToSlopeStartX = 0;
         if(slopeAngle != collisions.slopeAngleOld) {
           distanceToSlopeStartX = hit.distance - skinWidth;
           velocity.x -= distanceToSlopeStartX * rayDirectionX;
         }

         ClimbSlope(ref velocity, slopeAngle);
         velocity.x += distanceToSlopeStartX * rayDirectionX;
       }
```

We have a strange behaviour if there is an obstacle above. It is because the x speed has been modified by the slope but not the y speed

```
void VerticalCollisions(ref Vector3 velocity) {
  ...
  for(int i = 0; i < verticalRayCount; i++) {
    ...
    if (hit) {
      ...
      if(collisions.climbingSlope) {
        velocity.x = velocity.y / Mathf.Tan(collisions.slopeAngle * Mathf.Deg2Rad) * Mathf.Sign(velocity.x);
      }
      ...
    }
  }
}
```

It still have a strange behaviour when we hit an obstacle on the side being on a slope, but for now I did not find the solution.

There is a stop frame if we go from a slope to a slope with an higher angle. To solve it, add at the end of VerticalCollisions :

```
    void VerticalCollisions(ref Vector3 velocity) {
    ...
    // Avoid one frame stop when changing slope
    if (collisions.climbingSlope) {
      float rayDirectionX = Mathf.Sign(velocity.x);
      rayMagnitude = Mathf.Abs(velocity.x) + skinWidth;
      Vector2 rayOrigin = (rayDirectionX == -1)? raycastOrigins.bottomLeft : raycastOrigins.bottomRight + Vector2.up *
velocity.y;
      RaycastHit2D hit = Physics2D.Raycast(rayOrigin, Vector2.right * rayDirectionX, rayMagnitude, collisionMask);
      if (hit) {
        float slopeAngle = Vector2.Angle(hit.normal, Vector3.up);
        if (slopeAngle != collisions.slopeAngleOld) {
          velocity.x = (hit.distance - skinWidth) * rayDirectionX;
          collisions.slopeAngle = slopeAngle;
        }
      }
    }
}
```

13. Descendre les pentes

Ajouter un angle de descente maximum :

```
  [SerializeField] int maxDescendAngle = 75;
```

Modifier Move pour appeler la fonction de descente :

```
public void Move(Vector3 velocity)
{
    UpdateRaycastOrigins();
    collisions.Reset();

    if(velocity.y < 0) {
      DescendSlope(ref velocity);
    }
    if(velocity.x != 0) {
      HorizontalCollisions(ref velocity);
    }
    if(velocity.y != 0) {
      VerticalCollisions(ref velocity);
    }
    transform.Translate(velocity);
}
```

Créer la fonction :

```
void DescendSlope(ref Vector3 velocity) {
    float rayDirectionX = Mathf.Sign(velocity.x);
    Vector2 rayOrigin = (rayDirectionX == -1)? raycastOrigins.bottomLeft : raycastOrigins.bottomRight;
    RaycastHit2D hit = Physics2D.Raycast(rayOrigin, -Vector2.up, Mathf.Infinity, collisionMask);
    if (hit) {
      float slopeAngle = Vector2.Angle(hit.normal, Vector3.up);
      if(slopeAngle != 0 && slopeAngle < maxDescendAngle) {
        // Moving down the slop
        if (Mathf.Sign(hit.normal.x) == rayDirectionX) {
          float yMove = Mathf.Tan(slopeAngle * Mathf.Deg2Rad) * Mathf.Abs(velocity.x);
          if(hit.distance - skinWidth <= yMove) {
            float moveDistance = Mathf.Abs(velocity.x);
            float descendVelocityY = Mathf.Sin(Mathf.Deg2Rad * slopeAngle) * moveDistance;
            velocity.x = Mathf.Cos(Mathf.Deg2Rad * slopeAngle) * moveDistance * Mathf.Sign(velocity.x);
            velocity.y -= descendVelocityY;
            collisions.below = true;
            collisions.descendingSlope = true;
            collisions.slopeAngle = slopeAngle;
          }
        }
      }
    }
  }
```

Il faut mettre à jour les collisions info.

```
  public struct CollisionInfo {
    public bool above, below, left, right, climbingSlope, descendingSlope;
    ...
    public void Reset() {
      above = below = left = right = climbingSlope = descendingSlope = false;
      ...
```

```
    }
  }
```

Bug quand deux pentes inversées sont proches et que le joueur touche les deux :

Ajouter à CollisionInfo :

```
    public Vector3 velocityOld;
```

Ajouter au début de Move() :

```
    collisions.velocityOld = velocity;
```

Ajouter dans la partie Climb d'HorizontalCollisions() :

```
if (hit) {
        ...
        if(slopeAngle <= maxClimbAngle && i == 0) {

          if(collisions.descendingSlope) {
            collisions.descendingSlope = false;
            velocity = collisions.velocityOld;
          }
```