Documentation des fonctions mathématique d'Angry Birds en C

Arthur BRU

April 1, 2025

Description des fonctions

1. `SpeedInitial(float angle, float l1)`

But : Cette fonction calcule la vitesse initiale d'éjection d'un objet en prenant en compte son angle, en utilisant la longueur du ressort et les caractéristiques physiques du système (masse, gravité).

Paramètres :

- `angle` : L'angle de lancement du projectile en radians.
- `l1` : La longueur du ressort qui permet de déterminer la force de lancement.

Retour : La vitesse initiale d'éjection, `v_eject`, calculée en fonction des paramètres et des propriétés physiques du système.

Formule : La vitesse initiale est donnée par la formule :

$$v_{\text{eject}} = l1 \times \sqrt{\frac{\text{spring}}{\text{mass}}} \times \sqrt{1 - \left(\frac{\text{mass} \times |g| \times \sin(\text{angle})}{\text{spring} \times l1}\right)^2}$$

où spring est la longueur du ressort, mass est la masse de l'objet, et g est la gravité, ici représenté par celle d'unity soit 9.81f.

Implémentation :

```
private float SpeedInitial(float angle, float l1)
{
    var v_eject = l1 * Mathf.Sqrt(spring / mass) *
                  Mathf.Sqrt(1 - Mathf.Pow(mass * Mathf.Abs(Physics2D.gravity.y) *
    return v_eject;
}
```

2. `ComputeTrajectoryWithoutFriction(float angle, float velocity, bool isForCapacities = false)`

But : Cette fonction calcule la trajectoire d'un projectile sans prendre en compte le frottement de l'air.

Paramètres :

- `angle` : L'angle de lancement en radians.
- `velocity` : La vitesse initiale du projectile.
- `isForCapacities` : Un booléen optionnel pour ajuster la position de départ de la trajectoire.

Retour : Une liste de points représentant la trajectoire du projectile dans le plan 2D.

Formules : Les équations qui régissent la trajectoire sont :

$$x(t) = v_0 \cos(\theta) t$$

$$y(t) = v_0 \sin(\theta) t - \frac{1}{2} g t^2$$

où v_0 est la vitesse initiale, θ est l'angle de lancement, et g est la gravité.

Implémentation :

```
private List<Vector3> ComputeTrajectoryWithoutFriction(float angle, float velocity
{
    List<Vector3> points = new List<Vector3>();
    Vector3 startPosition = transform.position;

    float timeMax = (velocity * Mathf.Sin(angle) +
                     Mathf.Sqrt(Mathf.Pow(velocity * Mathf.Sin(angle), 2) + 2 * gr
    float timeStep = timeMax / numPoints;

    for (int i = 0; i < numPoints; i++)
    {
        float t = i * timeStep;
        float x = velocity * Mathf.Cos(angle) * t;
        float y = velocity * Mathf.Sin(angle) * t - 0.5f * gravity * t * t;

        if (float.IsNaN(x) || float.IsNaN(y)) continue;
        points.Add(new Vector3(isForCapacities ? startPosition.x + x : x, isForCap
    }
```

```
    return points;
}
```

3. `ComputeTrajectoryWithFriction(float angle, float velocity, bool isForCapacities = false)`

But : Cette fonction calcule la trajectoire d'un projectile en prenant en compte le frottement de l'air, ici modéliser par le coefficient de friction.

Paramètres :

- `angle` : L'angle de lancement en radians.
- `velocity` : La vitesse initiale du projectile.
- `isForCapacities` : Un booléen optionnel pour ajuster la position de départ de la trajectoire.

Retour : Une liste de points représentant la trajectoire du projectile en tenant compte du frottement.

Formules : La trajectoire avec frottement suit ces équations :

$$x(t) = \frac{\lambda_x}{k}(1 - e^{-kt})$$

$$y(t) = \frac{\lambda_y}{k}(1 - e^{-kt}) - \frac{g}{k}t$$

où $\lambda_x = v_0 \cos(\theta)$ et $\lambda_y = v_0 \sin(\theta) + \frac{g}{k}$.

Implémentation :

```
private List<Vector3> ComputeTrajectoryWithFriction(float angle, float velocity, b
{
    List<Vector3> points = new List<Vector3>();
    Vector3 startPosition = transform.position;

    float lambdaX = velocity * Mathf.Cos(angle);
    float lambdaY = velocity * Mathf.Sin(angle) + gravity / slingshot.frictionShot
    float timeMax = (velocity * Mathf.Sin(angle) +
                     Mathf.Sqrt(Mathf.Pow(velocity * Mathf.Sin(angle), 2) + 2 * gr
    float timeStep = timeMax / numPoints;

    for (int i = 0; i < numPoints; i++)
    {
```

```
        float t = i * timeStep;
        float x = (lambdaX / slingshot.frictionShot) * (1 - Mathf.Exp(-slingshot.fr
        float y = (lambdaY / slingshot.frictionShot) * (1 - Mathf.Exp(-slingshot.fr

        if (float.IsNaN(x) || float.IsNaN(y)) continue;
        points.Add(new Vector3(isForCapacities ? startPosition.x + x : x, isForCap
    }

    return points;
}
```

4. ComputeJumpTrajectoryWithoutFriction(float angle, float velocity)

But : Cette fonction calcule la récurrence de la trajectoire d'un saut sans frottement avec l'air.

Paramètres :

- `angle` : L'angle de lancement en radians.
- `velocity` : La vitesse initiale du joueur (déjà en l'air).

Retour : Une liste de points représentant la trajectoire du saut dans le plan 2D.

Formules : Les mêmes que pour `ComputeTrajectoryWithoutFriction` mais avec un temps plus petit entre chaque calcul pour obtenir une trajectoire continue.

Implémentation :

```
private List<Vector3> ComputeJumpTrajectoryWithoutFriction(float angle, float velo
{
    List<Vector3> points = new List<Vector3>();
    float dt = 0.01f;

    float x = transform.position.x;
    float y = transform.position.y;

    float velocityX = velocity * Mathf.Cos(angle);
    float velocityY = velocity * Mathf.Sin(angle);

    while (y >= 0)
    {
        x += velocityX * dt;
```

```
        y += velocityY * dt;

        if (y < 0) break;

        points.Add(new Vector3(x, y, 0));

        velocityY -= gravity * dt;
    }

    return points;
}
```

5. ComputeJumpTrajectoryWithFriction(float angle, float velocity, float friction)

But : Cette fonction calcule la récurrence d'une trajectoire d'un saut en tenant compte du frottement de l'air.

Paramètres :

- `angle` : L'angle de lancement en radians.
- `velocity` : La vitesse initiale du projectile.
- `friction` : Le coefficient de friction qui modifie la vitesse en fonction du temps.

Retour : Une liste de points représentant la trajectoire du saut avec friction.

Implémentation :

```
private List<Vector3> ComputeJumpTrajectoryWithFriction(float angle, float velocit
{
    List<Vector3> points = new List<Vector3>();
    float dt = 0.01f;

    float x = transform.position.x;
    float y = transform.position.y;

    float velocityX = velocity * Mathf.Cos(angle);
    float velocityY = velocity * Mathf.Sin(angle);

    points.Add(new Vector3(x, y, 0));
```

```
    while (y > 0)
    {
        x += velocityX * dt;
        y += velocityY * dt;

        points.Add(new Vector3(x, y, 0));

        velocityX -= friction * velocityX * dt;
        velocityY -= (gravity + friction * velocityY) * dt;

        if (points.Count > 500) break;
    }

    return points;
}
```

1. DrawMultipleTrajectories(BirdTrajectory bird)

But : Cette fonction permet de dessiner plusieurs trajectoires de projectiles (oiseaux) en simulant des angles de lancement différents. Elle génère plusieurs trajectoires avec un angle variant de 0 à $\frac{\pi}{2}$ et duplique un objet de type `BirdTrajectory` pour chaque angle, en fonction de la puissance du tir (définie par `slingshot.powerShot`).

Paramètre :

- `bird` : Un objet de type `BirdTrajectory` qui représente l'oiseau pour lequel les trajectoires sont générées. Cet objet est utilisé pour obtenir la position initiale de l'oiseau et pour dessiner la trajectoire à l'aide d'un `LineRenderer`.

Retour : Cette fonction ne retourne aucune valeur. Elle effectue une série de calculs et instantiations pour afficher les trajectoires.

Description du processus : La fonction crée 10 trajectoires avec des angles allant de 0 à $\frac{\pi}{2}$ radians. Pour chaque angle, elle instancie un nouvel objet de type `BirdTrajectory`, calcule l'angle correspondant, puis appelle la méthode `DrawTrajectoryWithLineRenderer` pour dessiner la trajectoire et propulser l'oiseau sur sa trajectoire.

Description des étapes :

1. `birdsDuplication.Clear()` : Vide la liste de duplications d'oiseaux.
2. La longueur du ressort (`l1`) est définie par la puissance de tir (`slingshot.powerShot`).
3. L'angle maximal ($\alpha_{\max}$) est fixé à $\frac{\pi}{2}$.

4. Un boucle `for` est utilisée pour générer 10 trajectoires avec des angles allant de 0 à $\frac{\pi}{2}$.

5. À chaque itération :

   - Un nouvel objet `birdTrajectory` est instancié.
   - La position de `birdTrajectory` est mise à celle de l'objet pour que chaque objet crée soit au même niveau que notre joueur `bird`.
   - Un angle α est calculé par interpolation linéaire entre 0 et $\alpha_{\max}$ (soit de 0 à $\frac{\pi}{2}$).
   - La méthode `DrawTrajectoryWithLineRenderer` est appelée pour dessiner la trajectoire avec l'angle calculé et la puissance du tir.
   - Si l'objet `birdTrajectoryScript` existe, il est ajouté à la liste `birdsDuplication`.

Implémentation :

```
public void DrawMultipleTrajectories(BirdTrajectory bird)
{
    birdsDuplication.Clear();

    float l1 = slingshot.powerShot;
    float alphaMax = Mathf.PI / 2;

    for (int i = 0; i <= 10; i++)
    {
        if (i == 0) continue;
        GameObject birdTrajectory = Instantiate(birdDuplication);
        if (bird) birdTrajectory.transform.position = bird.transform.position;
        float alpha = Mathf.Lerp(0, alphaMax, i / 10f); // Génère les angles de 0 

        BirdTrajectory birdTrajectoryScript = birdTrajectory.GetComponent<BirdTraj
        birdTrajectoryScript.DrawTrajectoryWithLineRenderer(alpha * Mathf.Rad2Deg,
        if (birdTrajectoryScript) birdsDuplication.Add(birdTrajectoryScript);
    }
}
```

Formules :

- **Calcul de l'angle :** L'angle α utilisé pour chaque trajectoire est calculé de manière linéaire entre 0 et $\frac{\pi}{2}$, ce qui permet de générer une série d'angles

régulièrement espacés. Ce calcul est effectué avec la fonction `Mathf.Lerp` :

$$\alpha = \texttt{Mathf.Lerp}(0, \alpha_{\max}, \frac{i}{10})$$

où i varie de 1 à 10 et $\alpha_{\max} = \frac{\pi}{2}$.