Labyrinthe v0.3 - Collisions avec les murs

Dans les jeux vidéo, la gestion des **collisions** permet d'éviter que le personnage passe à travers les murs ou d'autres obstacles. Dans notre projet de labyrinthe, nous devons empêcher le joueur de traverser les murs en vérifiant si une position à laquelle il souhaite se déplacer est libre (un sol) ou bloquée (un mur). Dans cet article, nous allons voir comment gérer les collisions avec les murs en utilisant les coordonnées logiques de notre labyrinthe, c'est-à-dire en vérifiant la position du personnage sur la grille du labyrinthe plutôt qu'en utilisant directement les pixels.

Dans cet article, nous allons voir comment gérer les collisions avec les murs en utilisant les **coordonnées logiques** de notre labyrinthe, c'est-à-dire en vérifiant la position du personnage sur la grille du labyrinthe plutôt qu'en utilisant directement les pixels.

Objectif de cet article :

1. Comprendre les **coordonnées logiques** du labyrinthe (lignes/colonnes) par rapport aux **pixels** sur l'écran.
2. Vérifier les **collisions avec les murs** en fonction des coordonnées logiques.
3. Empêcher le joueur de traverser un mur lors de ses déplacements avec les touches fléchées.

Étape 1 : Comprendre les coordonnées logiques et les pixels

Notre labyrinthe est représenté par une matrice (ou tableau à deux dimensions) dans laquelle :

- `1` représente un mur.
- `0` représente un sol libre.

Chaque élément de la matrice qui représente notre labyrinthe correspond à une **case logique** du labyrinthe. Le joueur se déplace de case en case sur cette grille, et sa position est déterminée par des **coordonnées logiques** (ligne et colonne).

Par contre, sur l'écran, nous dessinons ces cases en **pixels**. Par exemple, si chaque case fait 32x32 pixels, la position du personnage à l'écran est calculée en multipliant ses coordonnées logiques par la taille de la tuile.

Étape 2 : Calculer les coordonnées logiques du personnage

Pour vérifier les collisions avec les murs, nous allons utiliser les **coordonnées logiques** du personnage, c'est-à-dire la ligne et la colonne actuelles dans le labyrinthe.

Si la position du personnage en pixels est `(x_personnage, y_personnage)`, ses coordonnées logiques peuvent être obtenues en divisant ces coordonnées par la taille de la tuile (ici, 32 pixels).

Formule pour convertir les pixels en coordonnées logiques :

```
colonne_logique = x_personnage // TAILLE_TUILE
ligne_logique = y_personnage // TAILLE_TUILE
```

Étape 3 : Gérer les collisions avec les murs

Lorsque le joueur essaie de se déplacer, nous devons d'abord vérifier si la case vers laquelle il se dirige est un mur (`1`) ou un sol (`0`). Si c'est un mur, nous bloquons le déplacement.

Voici comment vérifier si le joueur peut se déplacer vers la gauche :

1. Convertir la position actuelle du personnage en **coordonnées logiques** (ligne/colonne).
2. Vérifier si la case logique vers laquelle il souhaite se déplacer est un mur.
3. Si c'est un mur, ne pas changer la position du personnage.

Exemple complet avec gestion des collisions

```python
# ...

# Position initiale du personnage (en pixels)
x_personnage = 32  # 1ère colonne (1 * taille de tuile)
y_personnage = 32  # 1ère ligne (1 * taille de tuile)
vitesse = 32  # Le personnage se déplace de 32 pixels à la fois (la taille d'une tuile)

# Fonction pour vérifier si la position x,y est un mur ou un sol. Elle doit renvoyer True si l
def est_un_mur(x, y):

    # Créez le contenu de cette fonction.

# Boucle de jeu principale
continuer = True
while continuer:

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            continuer = False

    # Récupérer les touches pressées
    touches = pygame.key.get_pressed()

    # Déplacer le personnage en fonction des touches (seulement si la case suivante n'est pas
    if touches[pygame.K_LEFT]:
        if not est_un_mur(x_personnage - vitesse, y_personnage):
            x_personnage -= vitesse  # Déplacer vers la gauche
    if touches[pygame.K_RIGHT]:
        if not est_un_mur(x_personnage + vitesse, y_personnage):
            x_personnage += vitesse  # Déplacer vers la droite
    if touches[pygame.K_UP]:
        if not est_un_mur(x_personnage, y_personnage - vitesse):
            y_personnage -= vitesse  # Déplacer vers le haut
    if touches[pygame.K_DOWN]:
        if not est_un_mur(x_personnage, y_personnage + vitesse):
            y_personnage += vitesse  # Déplacer vers le bas

    # ...
```

Explication du code :

1. **Calcul des coordonnées logiques** :
    - La fonction `est_un_mur(x, y)` convertit les coordonnées en pixels du personnage en **coordonnées logiques**. En divisant les coordonnées en pixels par la taille de la tuile, on obtient la colonne et la ligne correspondantes.
    - Par exemple, si le personnage est à `x_personnage = 64`, et que chaque tuile fait 32 pixels, la colonne logique serait `64 // 32 = 2`.
2. **Vérification de la collision** :
    - Avant de déplacer le personnage, on appelle `est_un_mur(x, y)` pour vérifier si la prochaine case est un mur ou non. Si c'est un mur (`True`), le déplacement est bloqué. Sinon, le personnage peut se déplacer.
    - Par exemple, si le joueur appuie sur la touche gauche, le programme vérifie d'abord si la case à gauche du personnage est un mur (`x_personnage - vitesse`). Si ce n'est pas un mur, le personnage est déplacé.

3. **Rendu graphique** :
    - Le labyrinthe est dessiné à chaque tour de boucle, puis le personnage est affiché à sa nouvelle position.
    - Le framerate est limité à 60 images par seconde pour un jeu fluide.

Étape 4 : Test des collisions

Grâce à ce système, le personnage ne peut plus traverser les murs dans le labyrinthe. Par exemple, s'il essaie de se déplacer vers un mur en appuyant sur une touche, il reste bloqué tant qu'il essaie d'entrer dans une case occupée par un mur.

Améliorations possibles

1. **Animation du personnage** : Ajoutez des animations lorsque le personnage se déplace pour rendre le mouvement plus fluide.
2. **Déplacements plus subtils** : Vous pourriez modifier la vitesse pour permettre des déplacements plus lents (pas nécessairement d'une tuile entière à chaque fois).
3. **Ajout de bonus ou d'ennemis** : Vous pourriez ajouter un ou des objet·s supplémentaire·s dans le labyrinthe (comme des clés ou des ennemis), qui interagissent avec le joueur.

Conclusion

Nous avons vu dans cet article comment gérer les collisions dans un labyrinthe en PyGame en utilisant les **coordonnées logiques** du labyrinthe pour déterminer si le joueur peut se déplacer vers une nouvelle case. En vérifiant si la case vers laquelle il se déplace est un mur ou non, nous empêchons le personnage de traverser des obstacles, rendant ainsi le jeu plus réaliste et interactif.

En appliquant cette technique, vous avez désormais une base solide pour gérer les mouvements et les collisions dans vos propres jeux en 2D avec PyGame !