Partie BSP

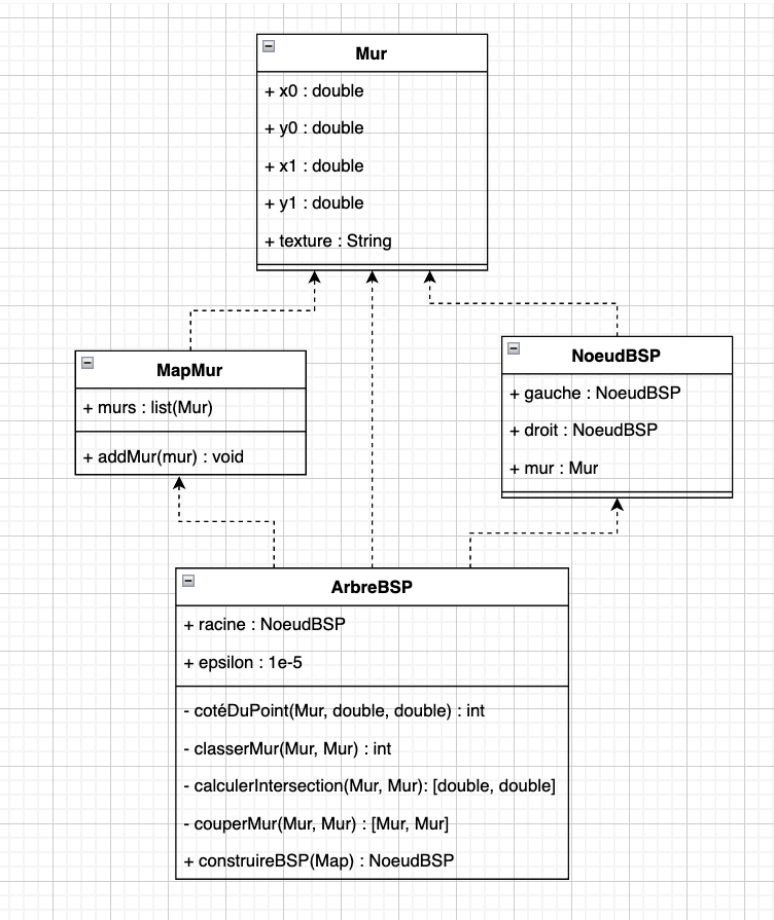
Découpage du BSP en plusieurs étapes:

1) Création de l'arbre Terminé
2) Affichage d'un seul mur en fonction de la position du joueur, etc Terminé
3) Parcour de l'arbre pour afficher tous les murs Pas commencé
4) Ajout de sprites (ennemis/joueurs) dans le rendu Pas commencé

Découpage de l’arbre

Sommaire:

Diagramme de classe:



https://app.diagrams.net/#G1j_Swy6lXRmvUZEFX53HQ2Wg-Ws8gCoy6#%7B%22pageId%22%3A%22IMjAHQZCgAVA_dQMFtUt%22%7D

Fonctions :

cotéDuPoint :

Source pour le calcul de la méthode cotéDuPoint :
https://zestedesavoir.com/forums/sujet/11587/savoir-si-le-point-est-a-gauche-ou-a-droite-dun-vecteur/

> **Vael**, lundi 12 novembre 2018 à 01h55
>
> ✓ Cette réponse a été utile
>
> Il te suffit de calculer le produit vectoriel entre $\overrightarrow{AB}$ et $\overrightarrow{AO_{vni}}$. Le signe du produit te donne la droite ou la gauche.

```
//une fonction pour déterminer de quel côté un point est par rapport à une ligne
fonction cotéDuPoint(murPartition, pointX, pointY):
    //calcul du produit vectoriel pour déterminer le côté
    dx <- murPartition.x1 - murPartition.x0
    dy <- murPartition.y1 - murPartition.y0

    //vecteur du point de départ vers le point testé
    px <- pointX - murPartition.x0
    py <- pointY - murPartition.y0

    produitCroisé <- dx * py - dy * px

    si produitCroisé > Ɛ:
        retourner 1 //gauche
    sinon si produitCroisé < -Ɛ:
        retourner -1 //droite
    sinon:
        retourner 0 //sur ligne
```

classerMur :

```
//une fonction pour dire si un mur est à gauche, droite, ou coupé par le mur
fonction classerMur(murPartition, murAClasser):
    //on teste les deux extrémités du mur
    coté0 <- cotéDuPoint(murPartition, murAClasser.x0, murAClasser.y0)
    coté1 <- cotéDuPoint(murPartition, murAClasser.x1, murAClasser.y1)

    si (coté0 = 0 ET coté1 = 0)
        // On décide arbitrairement de le mettre à GAUCHE (ou droite, peu importe, mais pas
COUPER)
        // Cela empêche le mur de disparaître.
        retourner 1

    si (coté0 = 0)
        retourner coté1
    si (coté1 = 0)
        retourner coté0

   si (coté0 = coté1)
        retourner coté0

    //le mur traverse la partition, il faut le couper
    retourner 0 //coupe
```

couperMur :

```
//fonction pour couper un mur en deux
fonction couperMur(murPartition, murACouper):
    //calcul du point d'intersection
    pointIntersectionX, pointIntersectionY <- calculerIntersection(murPartition, murACouper)

    //créer deux nouveaux murs
    mur1 <- nouveauMur(murACouper.x0, murACouper.y0, pointIntersectionX,
pointIntersectionY)
    mur2 <- nouveauMur(pointIntersectionX, pointIntersectionY, murACouper.x1, murACouper.y1)

    retourner [mur1, mur2]
```

calculerIntersection :

```
fonction calculerIntersection(murPartition, murACouper):
    // Calculer le dénominateur (D)
    // C'est le produit en croix des vecteurs directeurs
    // D = (x1 - x2) * (y3 - y4) - (y1 - y2) * (x3 - x4)
    denominateur <- (murPartition.x0 - murPartition.x1) * (murACouper.y0 - murACouper.y1) -
(murPartition.y0 - murPartition.y1) * (murACouper.x0 - murACouper.x1)

     //sécurité anti-crash si jamais on essaie de couper deux murs parallèles (ne devrait plus
arriver avec le fix classerMur)
    si (valeur_absolu(denominateur) < ε)
        retourner [murACouper.x0, murACouper.y0]

    // Calculer les numérateurs pour X et Y
    // On pré-calcule les produits croisés des points pour alléger la formule
    produit_12 <- (murPartition.x0 * murPartition.y1 - murPartition.y0 * murPartition.x1)
    produit_34 <- (murACouper.x0 * murACouper.y1 - murACouper.y0  * murACouper.x1)

    pointX <- (produit_12 * (murACouper.x0 - murACouper.x1) - (murPartition.x0 -
murPartition.x1) * produit_34) / denominateur
    pointY <- (produit_12 * (murACouper.y0 - murACouper.y1) - (murPartition.y0 -
murPartition.y1) * produit_34) / denominateur

    Retourner pointX, pointY
```

construireBPS :

```
//fonction récursive pour construire l'arbre BSP
fonction construireBSP(listeMurs):
    //si c'est vide on a fini
    si listeMurs est vide:
        retourner null

    //choisir un mur
    //on choisit le premier, mais on peut imaginer en choisir un autre … c'est un peu aléatoire ici,
//il faut tester plusieur cas et voir si on arrive à un arbre avec moins de noeud (comme doom)
    murPartition <- listeMurs[0]

    //créer le nœud
    noeud <- nouveau NoeudBSP
    noeud.mur <- murPartition
```

```
    //listes pour les murs à gauche et à droite, vides au début
    mursGauche <- []
    mursDroite <- []

    //parcourir les autres murs (on démarre à 1 pour ignorer le mur parent, ATTENTION si on
//utilise un autre mur que le premier comme parent)
    pour chaque mur dans listeMurs[1..fin]:
        classification <- classerMur(murPartition, mur)

        si classification = 1:
            mursGauche.ajouter(mur)
        sinon si classification = -1:
            mursDroite.ajouter(mur)
        sinon:
            //couper le mur en deux
            [mur0, mur1] <- couperMur(murPartition, mur)

            //On vérifie où se trouve le DEBUT du mur d'origine
            coteDepart <- coterDuPoint(murPartition, mur.x0, mur.y0)

            si (coteDepart = 1) // Début à Gauche
                    mursGauche.ajouter(mur0)
                    mursDroite.ajouter(mur1)
            si (coteDepart = -1) // Début à Droite
                    mursGauche.ajouter(mur1)
                    mursDroite.ajouter(mur0)
            sinon:
                    //si le point d'origine est sur la ligne, on doit regarder le point d'arrivée pour savoir
l'orientation
                    coteArrivee = coteDuPoint(murPartition, mur.x1, mur.y1)
                    si (coteArrivee = 1)
                            mursGauche.ajouter(mur1)
                            mursDroite.ajouter(mur0)
                    sinon
                            mursGauche.ajouter(mur0)
                            mursDroite.ajouter(mur1)

    //construire récursivement les sous-arbres
    noeud.noeudGauche <- construireBSP(mursGauche)
    noeud.noeudDroit <- construireBSP(mursDroite)

    retourner noeud
```

Rendu d’un mur

```
fonction rendreMur(joueur, mur):
    angleRadiant <- radian(joueur.angle)

    cosA <- cos(angleRadiant )
    sinA <- sin(angleRadiant )

    //transformer le mur en espace joueur
    wx0 <- mur.x0 - joueur.x
    wy0 <- mur.y0 - joueur.y
    wx1 <- mur.x1 - joueur.x
    wy1 <- mur.y1 - joueur.y

    //rotation inverse (monde -> caméra)
    cx0 <-  wx0 * cosA + wy0 * sinA
    cz0 <- -wx0 * sinA + wy0 * cosA
    cx1 <-  wx1 * cosA + wy1 * sinA
    cz1 <- -wx1 * sinA + wy1 * cosA

    // Rejet rapide (Si les deux points sont derrière le joueur)
    si cz0 < epsilon et cz1 < epsilon :
        continue
        //on peut ignorer le mur

    // Clipping (Si un seul point est derrière)
    // Si point 0 derrière, on le coupe pour le ramener à z = 0.1
    // epsilon = 0.1
    si cz0 < epsilon:
        t <- (epsilon - cz0) / (cz1 - cz0)
        cz0 <- epsilon
        cx0 <- cx0 + t * (cx1 - cx0)

    si cz1 < 0.1:
        t = (epsilon - cz1) / (cz0 - cz1)
        cz1 = epsilon
        cx1 = cx1 + t * (cx0 - cx1)

    //projection perspective
    scale <- (ecran.largeur / 2) / tan(joueur.fov / 2)

    sx0 <- (cx0 / cz0) * scale + (ecran.largeur / 2)
    sx1 <- (cx1 / cz1) * scale + (ecran.largeur / 2)
```

```
        // Mise en ordre (gauche vers droite)
        // Il faut s'assurer que x_start < x_end pour la boucle de dessin
        si sx0 < sx1:
                x_start <- sx0
                x_end <- sx1
                z_start <- cz0
                z_end <- cz1
        sinon:
                x_start <- sx1
                x_end <- sx0
                z_start <- cz1
                z_end <- cz0

        //profondeur du mur (painter)
        depth = (cz0 + cz1) / 2

                // Stocker pour le painter's algorithm
                renderData <- {
                    x_start: x_start,
                    x_end: x_end,
                    z_start: z_start,
                    z_end: z_end,
                    depth: depth
                }

                rafraichirAffichage()
```

Affichage (dessin)

```
Fonction rafraichirAffichage():
    largeur <- ecran.largeur
    hauteur <- ecran.hauteur
    pixels <- tableau[largeur * hauteur] // Tableau 1D représentant l'écran

    // Remplir le fond (ciel + sol)
    miHauteur <- hauteur / 2

    Pour y de 0 à hauteur - 1:
        // Choisir la couleur selon la moitié de l'écran
        Si y < miHauteur:
            couleur <- COULEUR_CIEL
        Sinon:
            couleur <- COULEUR_SOL
```

```
    // Remplir la ligne
    Pour x de 0 à largeur - 1:
        pixels[y * largeur + x] <- couleur


// Récupération des données calculées dans rendreMur()
x_debut_ecran <- renderData.x_start
x_fin_ecran   <- renderData.x_end
z_debut       <- renderData.z_start
z_fin         <- renderData.z_end

// Paramètres de la scène
// Hauteur définie arbitrairement dans le monde
// Ex: 200px
// Largeur définie en fonction de la largeur du de l'écran / nombre de colonnes
// Ex: 800px / 100cols = 8px par colonne
hauteurMurMonde <- 200.0
largeurColonne  <- largeur / nombreDeColonnes

// Pré-calcul pour l'interpolation perspective (Correction 1/Z)
// Permet d'éviter que le mur se déforme
invZ_debut <- 1.0 / z_debut
invZ_fin   <- 1.0 / z_fin

// Déterminer les colonnes logiques à dessiner
col_min <- max(0, partieEntiere(x_debut_ecran / largeurColonne))
col_max <- min(nombreDeColonnes - 1, partieEntiere(x_fin_ecran / largeurColonne))

// Boucle sur les colonnes verticales
Pour col de col_min à col_max:

    // Calculer le centre X de la colonne actuelle (en pixel)
    x_centre <- (col + 0.5) * largeurColonne

    // Vérification de sécurité (si le mur est très incliné)
    Si x_centre < x_debut_ecran OU x_centre > x_fin_ecran:
        Continuer // Passer à la colonne suivante

    // On calcule 't' (pourcentage d'avancement entre 0 et 1 sur l'axe X écran)
    t <- (x_centre - x_debut_ecran) / (x_fin_ecran - x_debut_ecran)

    // On interpole 1/z, puis on inverse pour trouver le vrai Z
    invZ_actuel <- invZ_debut + t * (invZ_fin - invZ_debut)
    z_actuel <- 1.0 / invZ_actuel
```

```
    // Thales : HauteurEcran = HauteurMonde / Profondeur
    hauteurProjetee <- hauteurMurMonde / z_actuel
    demiHauteur <- hauteurProjetee / 2

    y_haut <- max(0, miHauteur - demiHauteur)
    y_bas  <- min(hauteur - 1, miHauteur + demiHauteur)

    // Plus c'est loin (z grand), plus c'est sombre
    intensite <- borne(255 - (z_actuel * 20), 50, 255)
    couleurMur <- couleurRGB(intensite, intensite / 2, intensite / 3) // Teinte orangée

    // On remplit les pixels correspondant à cette colonne logique
    pixel_x_start <- partieEntiere(col * largeurColonne)
    pixel_x_end   <- partieEntiere(min(largeur - 1, (col + 1) * largeurColonne - 1))

    Pour y de y_haut à y_bas:
        offsetY <- y * largeur
        Pour x de pixel_x_start à pixel_x_end:
            pixels[offsetY + x] <- couleurMur

//rendu final
drawImage(...)
```

Rendu Graphique