PROJET GROUPE E :

Compte Rendu n°2

Groupe: Anouk OMS, Lysandre TERRISSE, Elora ROGET, Thomas DAILLY

But : Nous avons choisi de coder un jeu de plateforme dans le style de Mario en programmation C. Sa représentation sera en 2D comme les anciennes versions.

Langage de Programmation :

Nous avons décidé de coder en C car d'après nos recherches il s'agit d'un bon langage permettant de coder des jeux et nous enlevant certaines contraintes quant à l'optimisation. On travaillera également avec CSFML, une bibliothèque initialement créée pour C++, et qui nous permettra de faire l'aspect graphique de notre jeu.

Au cours de la dernière séance, celle du 24 Janvier, nous avons accueilli un nouveau membre dans le groupe. Nous avons donc pris un peu de temps afin de lui expliquer notre projet et répartir à nouveau les différentes tâches (qui ne sont pas encore toutes attribuées)

Répartition : Nous avons décidé de nous répartir les tâches en suivant deux catégories : l'aspect graphique du jeu et l'aspect code

- **Anouk** : Code
    - **Monstres**:
        - Déplacements : lorsqu'un monstre apparaît, il choisit une direction unique en fonction de la position du joueur : si celui-ci se tient à sa gauche alors il ira vers lui mais ne changera jamais de direction.
        - Attaques : lorsqu'un monstre touche le joueur, il lui fait perdre une vie.
        - Apparition des monstres : les monstres proviennent soit des poubelles implémentées partout sur la carte, soit ils sont déjà présents sur la carte (et sont figés tant qu'ils ne sont pas visibles

sur l'écran). Lorsqu'une poubelle apparaît sur l'écran, celui-ci génère un ennemi de manière régulière (un toutes les dix secondes) et arrête lorsqu'il a fait apparaître dix ennemis afin de ne pas trop charger la carte. Une poubelle arrête de générer des monstres lorsque le joueur s'éloigne d'elle (*définir la distance nécessaire entre une poubelle et le joueur, hors-champ*)
    - Disparition lorsque le joueur est trop loin d'eux (*définir la distance nécessaire entre un monstre et le joueur, hors-champ*)

- **Lysandre** : Code
    - Création de deux formats PixelArt et ObjectArt décrits ci-dessous pour les images, et création des fonctions calculateImage et useImage, qui permettent de calculer la texture d'une image et de la calquer à un endroit précis et avec une échelle précise.
    - Gestion du son et création des musiques.
    - Implémentation du système de collisions décrit ci-dessous.

- **Elora** : Graphisme
    - Création des images
    - Les fonctions touchant à la vie du personnage (initialisation de la barre de vie, décrémentation d'une lors d'une attaque de monstre, mort du joueur si la barre atteint 0).
    - Aide sur les fonctions qui demandent beaucoup de travail et/ou de réflexion comme la fonction à propos des collisions.

- **Thomas** : Graphisme
    - Aide aux graphismes et à la fonction de collision.

Organisation du jeu :

- **Personnage** (joueur) : placé de manière à pouvoir voir ce qu'il y a derrière et devant le personnage (un tiers de l'écran est derrière lui). Il se déplace principalement de manière horizontale. Le personnage a au début de chaque partie trois vies et meurt lorsqu'il n'en a plus. Le personnage aura la forme d'une petite boule de suie noire.

- **Obstacles** : principalement des poubelles ou des blocs que le personnage doit contourner en sautant par dessus ou en s'aidant des plateformes en hauteur.

- **Monstres** : lors de l'apparition du monstre, celui-ci ira en direction du joueur mais ne changera jamais de vitesse ou de direction, ainsi le joueur peut simplement l'éviter en sautant au-dessus. Mais si le joueur touche le monstre, il perd une vie. Les monstres prendront la forme d'un balai.

Planning :

Date	Elora	Anouk	Lysandre	Thomas
Semaine du 23 Janvier	Apprendre les bases du C Avoir une idée du design du jeu. Créer un github.	Apprendre les bases du C Créer un github	Apprendre les bases du C Créer un github	Apprendre les bases du C Créer un github
Semaine du 30 janvier	Apprendre les bases du C Préciser le détail des fonctions Créer un diagramme de Gantt	Apprendre les bases du C Préciser le détail des fonctions Créer un diagramme de Gantt	Apprendre les bases du C Préciser le détail des fonctions Créer un diagramme de Gantt	Apprendre les bases du C Préciser le détail des fonctions Créer un diagramme de Gantt
Semaine de 06 février	Créer un diagramme de Gantt	Créer un diagramme de Gantt	Créer un diagramme de Gantt	Créer un diagramme de Gantt

---

Principe de l'algorithme de vitesses et d'inputs :

Pour que le joueur ait un mouvement fluide, au lieu d'incrémenter sa position lorsqu'une touche est maintenue, nous avons choisi d'**incrémenter une vitesse**. Plus précisément, il y a une vitesse verticale et une vitesse horizontale, qui représentent toutes les deux une distance parcourue (en blocs) par cycle de calcul (tick ou image ou frame). Ces deux vitesses forment un vecteur de vitesse qui nous dit où est censé aller le personnage au prochain tick (dans le cas où il ne rencontre pas d'obstacles). Cette vitesse peut être modifiée par le joueur avec les flèches haut,

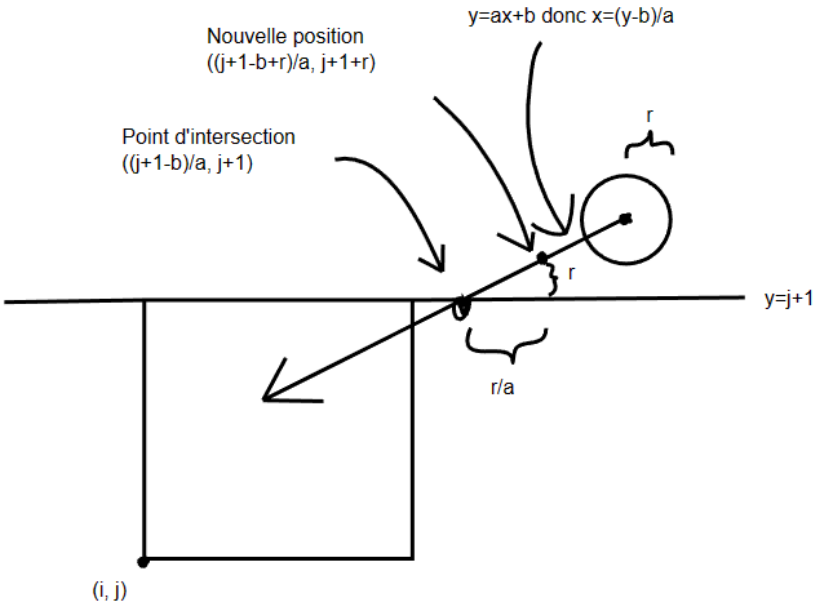
gauche, et droite. Pour le saut, il faut faire une affectation plutôt qu'une incrémentation, pour que bond soit soudain. Voici des procédures potentielles pour les vitesses.

```
Procedure: changer_vitesse_verticale
Début
        si la flèche du haut est maintenue et la face du bas touche un bloc alors
                vitesse_verticale <- constante_de_saut
        sinon
                vitesse_verticale    <-    max(-vitesse_verticale_maximale,    vitesse_verticale    -
constante_de_gravite)
        fin si
Fin


Procedure: changer_vitesse_horizontale
Début
        si la flèche de gauche est maintenue et non(la flèche de droite est maintenue) alors
                vitesse_horizontale   <-   max(-vitesse_horizontale_maximale,   vitesse_horizontale   -
constante_de_marche)

        sinon si la flèche de droite est maintenue et non(la flèche de gauche est maintenue)
                vitesse   horizontale   <-   min(vitesse_horizontale_maximale,   vitesse_horizontale   +
constante_de_marche)

        sinon
                vitesse_horizontale <- vitesse_horizontale * constante_de_conservation_de_vitesse
                //constante entre 0 et 1 inclus (probablement très proche de 1).
        fin si
Fin
```

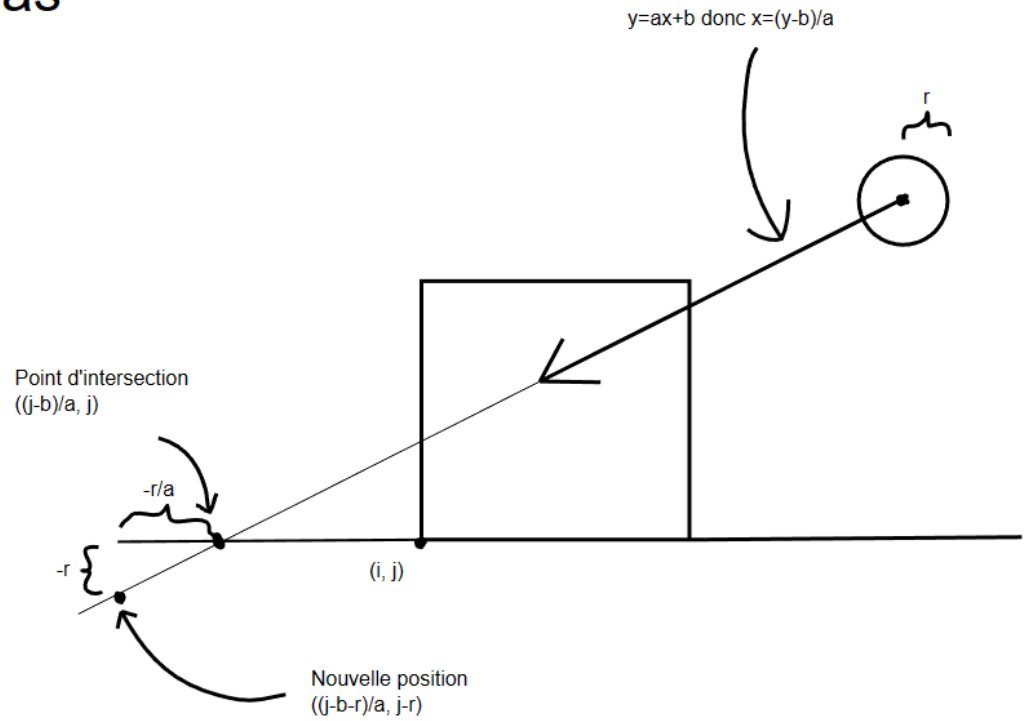
Principe de l'algorithme de collisions :

On a maintenant un vecteur nous disant où devrait aller notre personnage. Mais ce vecteur peut pointer vers une zone invalide, comme à l'intérieur d'un bloc solide. On ne peut pas juste annuler le mouvement. À la place, il faut coller le personnage contre le bord de ce bloc. **Pour chaque bord du bloc**, il faut **déterminer le point d'intersection** entre le vecteur et la droite correspondant à ce bord. Cependant, on ne peut toujours pas placer le centre de notre personnage à ce point, car sinon il serait à moitié dans ce bloc. Il faut donc **faire reculer le personnage** de la moitié de sa propre longueur, que l'on représente ici par **r** pour rayon. Nous avons représenté ci-dessous les calculs nécessaires pour déterminer la position d'un joueur par rapport au coin bas-gauche d'un bloc solide. Après avoir calculé ces quatre points, il faut **choisir celui le plus proche du joueur**, puis **répéter récursivement** avec ce nouveau vecteur de vitesse, jusqu'à ce que la position du joueur soit valide.

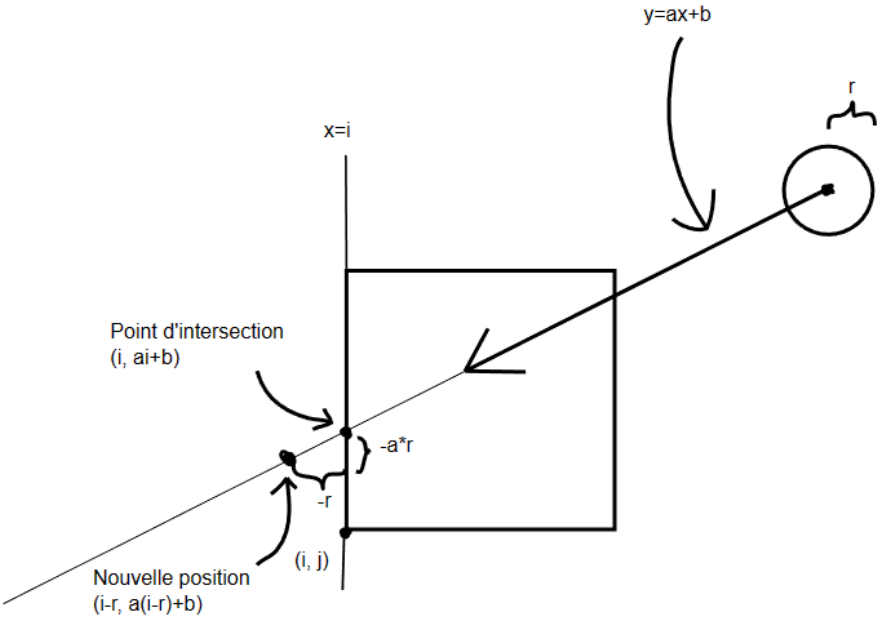
Cas haut


y=ax+b donc x=(y-b)/a
Nouvelle position
((j+1-b+r)/a, j+1+r)
Point d'intersection
((j+1-b)/a, j+1)
r
r
y=j+1
r/a
(i, j)

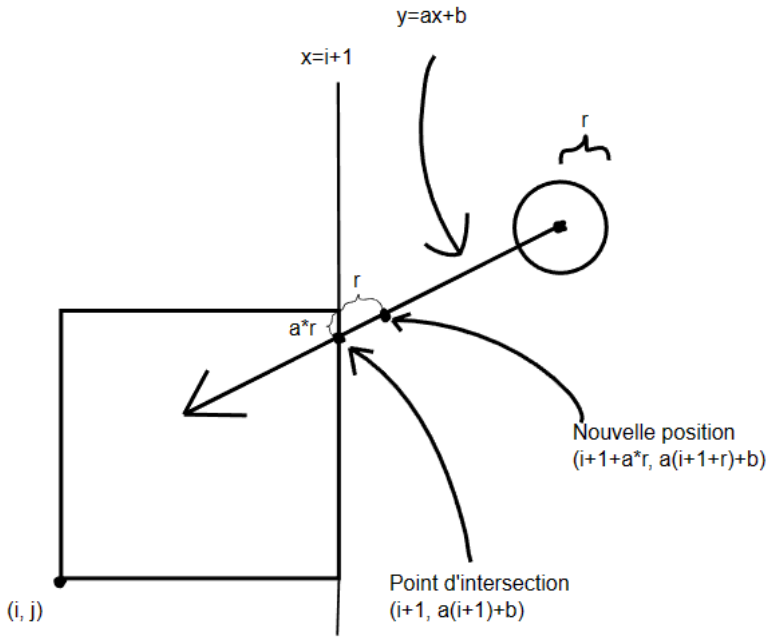
Cas bas



Cas gauche



Cas droite



Principe des formats PixelArt et ObjectArt :

D'après ce que nous avons compris, en C, l'affichage sur l'écran se fait par des textures. Au lieu de dessiner sur l'écran, nous devons dessiner sur des textures, puis les appliquer à une certaine position et avec une certaine échelle. Cela permet de calculer une seule fois chaque texture, plutôt que d'en calculer une à chaque bloc sur l'écran, et à chaque image du jeu. Pour être organisé avec ces textures, nous pensons faire un fichier txt pour chaque texture, avec à l'intérieur une série d'inscriptions permettant de dessiner. Il faut que ces instructions soient simples à exporter du fichier txt, et utiles pour dessiner. Nous avons décidé de faire deux formats : un format pour pixel art, et un format contenant des instructions. Voici des exemples potentiels. Il est cependant très improbable qu'ils restent sous cette forme d'ici la fin du projet.

Pour PixelArt :

```
700 500
(255,0,0)*3,(255,255,255)*2,(0,0,255),None*6;(173,251,23)*19,…
(213, 254, 82)*17,…
…
```

La première ligne signifie que l'image fait 700 pixels de longueur et 500 pixels de largeur. Chaque ligne suivante correspond à une et une seule ligne de l'image. Les instructions de la deuxième ligne veulent dire que l'on fait trois pixels bleus, puis deux pixels blancs, puis un pixel rouge, puis six pixels transparents, puis 19 pixels de couleur (173,251,23), etc.

Pour ObjectArt :

```
700 500
disk(200,300,50,255,255,255)
rectfull(320,200,650,700,0,0,0)
trianglefull(0,0,0,200,200,0,255,0,172)
line(0,0,500,500,255,255,255)
text("Hello world",500,500,"Consolas",12,255,255,255)
texture("grass_block.txt",0,0,100,200)
pixel(300,300,255,255,255)
```

La deuxième ligne demande de faire un disque à la position (200,300), de rayon 50, et de couleur blanche. rectfull prend deux positions (x,y) et une couleur (r,g,b). trianglefull prend trois points et une couleur rgb. line prend deux points et une couleur rgb. text prend une chaîne de caractères, une position, une police d'écriture, une taille de police, et une couleur. La texture prend un fichier, une position, une longueur, et une largeur. pixel prend une position et une couleur. Heureusement pour nous, CSFML propose déjà des fonctions pour le texte, les textures, et les rectangles. Cependant, il faut que l'on programme nous mêmes les fonctions disk, trianglefull, line, et pixel. Contrairement à ce que l'on pourrait penser, disk est assez simple, puisqu'il suffit d'utiliser le théorème de Pythagore (même s'il y a toujours quelques ambiguïtés sur quelle méthode utiliser). La fonction la plus complexe semble de loin être line, puisqu'il n'y a pas de "bonne" manière de faire des lignes obliques.