Max Braudel, Mathéo Palazzolo, Marie Blanchard

Rapport : Sorbet Coco

Voir le projet

1. Présentation générale

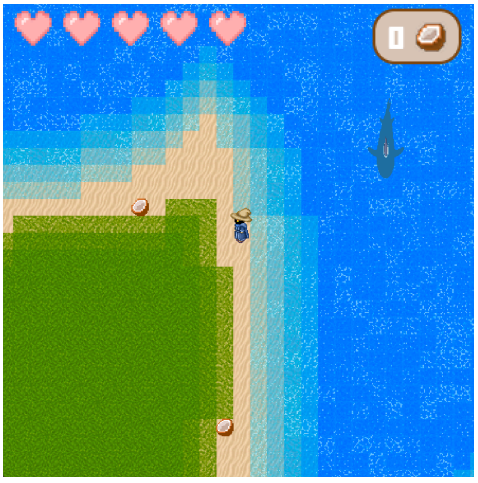
a. Concept du jeu

Dans le cadre de nos cours de programmation C++ et de synthèse d'images, nous avons réalisé un jeu vidéo en 2D.

Nous avons choisi le thème des Caraïbes.🌴
Sur les plages paradisiaques, un mage doit attraper toutes les noix de coco afin de préparer son sorbet coco. Mais, il fait fasse à des obstacles. Des pirates le poursuivent pour le piller.

b. Règles du jeu

Gagner : Récupérer les noix de coco
Perdre : Se faire attrapper par les pirates
Départ : 3 vies
Gagner des vies : Récupérer des coeurs éparpillés sur la carte
Vous pouvez jeter de la glace pour créer des chemins entre les îles.



c. Guide d'utilisation

Déplacement
ZQSD ou flèche directionnelles

Créer de la glace pour faire des chemins
Espace

Sprint
Shift

Réinitialiser la position du joueur
R

Zoomer
P

Dézoomer
,

Mode debug
F7

Quitter
X

2. Description de fonctionnalités

Comment fonctionne la génération de la map ?

La génération du terrain repose sur l'algorithme Cellular Automata.

Bruit procédural : Une **grille de bruit** est générée aléatoirement. La fonction permet une **interpolation bilinéaire** pour obtenir des valeurs lissées à partir des coordonnées.
Génération de la carte : Les blocs sont classifiés (eau, sable, herbe) selon des **seuils** définis appliqués aux valeurs de bruit.
Distances au sable : Un algorithme BFS calcule pour chaque bloc sa distance au plus proche bloc de sable.
Affinement des textures : Les blocs d'eau et d'herbe reçoivent une **texture spécifique** selon leur distance au sable, pour un rendu plus naturel.
Placement d'éléments et d'entités : Des **règles de génération** (probabilités, distances, types de blocs) guident le placement des éléments décoratifs et des entités (PNJ, ennemis…). Le placement peut être **aléatoire ou séquentiel**, avec des offsets pour simuler des regroupements naturels.

Comment gérer les pirates ennemis ?

Les pirates ont trois comportements :

Mode patrouille : Quand ils ne voient pas le joueur, ils se déplacent dans leur zone, au hasard.

Mode poursuite : Dès que le pirate voit le joueur, il se met à le poursuivre, en utilisant un algorithme de pathfinding.

Les ennemis sont générés automatiquement au lancement de la partie. Leur nombre dépend de la taille de la carte, pour un bon équilibre. Ils savent détecter les obstacles (cocotiers).

Comment marche le système de multi-threading ?

Le jeu utilise deux threads pour répartir le travail entre différentes parties du programme, ce qui permet de le rendre plus fluide et plus réactif. En effet, cela permet d'afficher le jeu, malgré les calculs lourds du gameplay.

Ici, on utilise cette technique pour séparer :

La logique de jeu : déplacement des entités, collisions, états du jeu, input utilisateur. Ce thread tourne à un rythme fixe de 60 images par seconde.

Le rendu graphique : sprites, map, interface utilisateur, animation des personnages. Ce thread tourne le plus vite possible.

La gestion des collisions

Détection des obstacles : On vérifie si les deux éléments du jeu se touchent grâce à du bounding box. Les objets sont entourés d'un rectangle invisible. S'ils se chevauchent, alors il y a collision.

Résolution des obstacles :

- Contre un obstacle : le joueur est bloqué et son déplacement est annulé
- Contre un ennemi : perte d'une vie

Pour ne pas ralentir le jeu, on vérifie les collisions avec les objets proches ou on filtre. par exemple, on n'a pas besoin de tester une collision entre deux pirates.

Affichage des éléments UI
Les éléments d'interface (menus, icônes, HUD) sont affichés au-dessus de tous les éléments du jeu.

Chargement : chaque élément est défini par une structure.

Placement : la fonction affiche un élément à une position fixe à l'écran.

Rendu : les éléments sont stockés et dessinés en dernier, pour une priorité d'affichage maximale.

3. Configuration poussée des objets (la force du projet)

CONFIGURATION DES BLOCKS

A. PROPRIÉTÉS DE BASE

- `std::string path` : Chemin vers le fichier texture
- `GLuint textureID` : ID OpenGL de la texture (généré automatiquement)
- `int textureWidth/textureHeight` : Dimensions de la texture

B. SYSTÈME D'ANIMATION

- `TextureAnimationType animType` : Type d'animation (`STATIC` ou `ANIMATED`)
- `float animationSpeed` : Vitesse d'animation en FPS (ex: 20.0f)
- `int frameCount` : Nombre total de frames dans le sprite sheet
- `int frameHeight` : Hauteur d'une frame (ex: 16px)
- `bool animationStartRandomFrame` : Démarrage aléatoire de l'animation
- `float currentFrame` : Frame actuelle (par défaut, remplacée par Block.currentFrame)

C. SYSTÈME DE ROTATION

- `bool randomizedRotation` : Rotation aléatoire activée/désactivée
- Les rotations possibles : 0°, 90°, 180°, 270°

D. SYSTÈME DE TRANSFORMATION TEMPORELLE

- `bool hasTransformation` : Transformation activée/désactivée
- `BlockName transformBlockTo` : Type de bloc cible de la transformation
- `float transformBlockTimeIntervalStart/End` : Intervalle de temps aléatoire pour la transformation
- `bool savePreviousExistingBlock` : Sauvegarde du bloc précédent
- `bool transformBlockToPreviousExistingBlock` : Retour au bloc précédent sauvegardé

2. STRUCTURE `Block` - Instance de Bloc

Chaque bloc placé sur la carte possède ces propriétés :

A. PROPRIÉTÉS SPATIALES

- `BlockName name` : Type de bloc
- `int x, y` : Coordonnées sur la grille

B. ANIMATION INDIVIDUELLE

- `float currentFrame` : Frame actuelle de l'animation (spécifique à chaque instance)
- `int rotationAngle` : Angle de rotation (0, 90, 180, 270)

C. TRANSFORMATION TEMPORELLE

- `float transformationTimer` : Minuteur de transformation
- `float transformationTarget` : Temps cible pour la transformation
- `bool hasBeenInitializedForTransformation` : Flag d'initialisation

CONFIGURATION DES ENTITEES

1. Identification et Rendu

```
EntityName type;                    // Type unique de l'entité (PLAYER, ANTAGONIST, SHARK, etc.)
ElementName elementName;            // Nom de l'élément graphique associé
float scale;                        // Échelle de rendu (1.0 = taille normale)
```

2. Système de Santé et Dégâts

```
int lifePoints = 100;               // Points de vie de l'entité
int damagePoints = 0;               // Points de dégâts infligés par l'entité
std::vector<BlockName> damageBlocks; // Blocs qui infligent des dégâts à l'entité
```

3. Configuration des Sprites et Animations

```
// Configuration par défaut
int defaultSpriteSheetPhase;        // Phase de sprite par défaut
int defaultSpriteSheetFrame;        // Frame de sprite par défaut
float defaultAnimationSpeed;        // Vitesse d'animation par défaut

// Phases d'animation pour les directions de marche
int spritePhaseWalkUp;              // Phase sprite pour marcher vers le haut
int spritePhaseWalkDown;            // Phase sprite pour marcher vers le bas
int spritePhaseWalkLeft;            // Phase sprite pour marcher vers la gauche
int spritePhaseWalkRight;           // Phase sprite pour marcher vers la droite
```

4. Système de Mouvement

```
// Vitesses de déplacement
float normalWalkingSpeed;           // Vitesse de marche normale
float normalWalkingAnimationSpeed;  // Vitesse d'animation en marche normale
float sprintWalkingSpeed;           // Vitesse de course
float sprintWalkingAnimationSpeed;  // Vitesse d'animation en course
```

5. Système de Collision Avancé

Collision de Base

```
bool canCollide;                    // Active/désactive les collisions
std::vector<std::pair<float, float>> collisionShapePoints; // Points définissant la forme de collision
```

Collision Granulaire avec les Éléments

```
std::vector<ElementName> avoidanceElements;  // Éléments évités en pathfinding
std::vector<ElementName> collisionElements;  // Éléments bloquant physiquement
```

Collision Granulaire avec les Blocs

```
std::vector<BlockName> avoidanceBlocks;      // Blocs évités en pathfinding
std::vector<BlockName> collisionBlocks;      // Blocs bloquant physiquement
```

Collision Granulaire avec les Entités

```
std::vector<EntityName> avoidanceEntities;   // Entités évitées en pathfinding
std::vector<EntityName> collisionEntities;   // Entités bloquant physiquement
```

6. Contrôle des Limites de Carte

```
bool offMapAvoidance = true;        // Éviter les bords de carte en pathfinding
bool offMapCollision = true;        // Collisions avec les bords de carte
```

7. Système Comportemental Automatique

Configuration Générale

```
bool automaticBehaviors = false;    // Active/désactive les comportements automatiques
```

État Passif (Exploration Aléatoire)

```
bool passiveState = false;          // Active l'état passif
float passiveStateWalkingRadius = 10.0f;  // Rayon d'exploration
float passiveStateRandomWalkTriggerTimeIntervalMin = 2.0f;  // Temps min entre marches
float passiveStateRandomWalkTriggerTimeIntervalMax = 8.0f;  // Temps max entre marches
```

État d'Alerte

```
bool alertState = false;            // Active l'état d'alerte
float alertStateStartRadius = 3.0f; // Rayon de début d'alerte
float alertStateEndRadius = 8.0f;   // Rayon de fin d'alerte
std::vector<EntityName> alertStateTriggerEntitiesList; // Entités déclenchant l'alerte
```

État de Fuite

```
bool fleeState = false;             // Active l'état de fuite
bool fleeStateRunning = true;       // Courir pendant la fuite
float fleeStateStartRadius = 3.0f;  // Rayon de début de fuite
float fleeStateEndRadius = 6.0f;    // Rayon de fin de fuite
float fleeStateMinDistance = 8.0f;  // Distance minimale à maintenir
float fleeStateMaxDistance = 12.0f; // Distance maximale de fuite
std::vector<EntityName> fleeStateTriggerEntitiesList; // Entités déclenchant la fuite
```

État d'Attaque

```
bool attackState = false;           // Active l'état d'attaque
bool attackStateRunning = true;     // Courir pendant l'attaque
float attackStateStartRadius = 5.0f; // Rayon de début d'attaque
float attackStateEndRadius = 10.0f; // Rayon de fin d'attaque
float attackStateWaitBeforeChargeMin = 1.0f; // Temps min avant charge
float attackStateWaitBeforeChargeMax = 3.0f; // Temps max avant charge
std::vector<EntityName> attackStateTriggerEntitiesList; // Entités déclenchant l'attaque
```

CONFIGURATION DES ELEMENTS

2. Type et Dimensions de Texture

```
ElementTextureType type = ElementTextureType::STATIC; // Type : STATIC ou SPRITESHEET
int spriteWidth = 0;            // Largeur d'un sprite individuel (pour spritesheets)
int spriteHeight = 0;           // Hauteur d'un sprite individuel (pour spritesheets)
int totalWidth = 0;             // Largeur totale de la texture
int totalHeight = 0;            // Hauteur totale de la texture
GLuint textureID = 0;           // Handle OpenGL de la texture
```

3. Système de Point d'Ancrage

```
AnchorPoint anchorPoint = AnchorPoint::CENTER; // Point d'ancrage par défaut
float anchorOffsetX = 0.0f;     // Décalage X depuis le point d'ancrage
float anchorOffsetY = 0.0f;     // Décalage Y depuis le point d'ancrage
```

4. Système de Collision

```
bool hasCollision = false;      // Active/désactive la détection de collision
std::vector<std::pair<float, float>> collisionShapePoints; // Points définissant le polygone de collision
```

Énumération AnchorPoint

Le système propose **7 points d'ancrage différents** :

```
enum class AnchorPoint {
    CENTER,              // Centre de la texture (par défaut)
    TOP_LEFT_CORNER,     // Coin supérieur gauche
    TOP_RIGHT_CORNER,    // Coin supérieur droit
    BOTTOM_LEFT_CORNER,  // Coin inférieur gauche
    BOTTOM_RIGHT_CORNER, // Coin inférieur droit
    BOTTOM_CENTER,       // Centre inférieur (idéal pour personnages)
    USE_TEXTURE_DEFAULT  // Utilise le point d'ancrage par défaut de la texture
};
```

Énumération ElementTextureType

```
enum class ElementTextureType {
    STATIC,     // Texture statique simple
    SPRITESHEET // Feuille de sprites avec animations
};
```

CONFIGURATION DES UIELEMENTS

2. Type et Dimensions de Texture

```
UIElementTextureType type = UIElementTextureType::STATIC; // Type : STATIC ou SPRITESHEET
int spriteWidth = 0;            // Largeur d'un sprite individuel (pour spritesheets)
int spriteHeight = 0;           // Hauteur d'un sprite individuel (pour spritesheets)
```

3. Configuration des Sprites et Animations

```
int defaultSpriteSheetPhase = 0;  // Phase de sprite par défaut (ligne)
int defaultSpriteSheetFrame = 0;  // Frame de sprite par défaut (colonne)
bool isAnimated = false;          // Animation automatique activée
float animationSpeed = 10.0f;     // Vitesse d'animation en FPS
```

4. Système de Marges pour Positionnement

```
float marginTop = 0.0f;         // Marge supérieure en pixels
float marginBottom = 0.0f;      // Marge inférieure en pixels
float marginLeft = 0.0f;        // Marge gauche en pixels
float marginRight = 0.0f;       // Marge droite en pixels
```

Énumérations du Système UI

UIElementName

```
enum class UIElementName {
    START_MENU,     // Menu de démarrage
    PAUSE_MENU,     // Menu de pause
    GAME_OVER,      // Écran de game over
    WIN_MENU,       // Menu de victoire
    HEALTH_BAR,     // Barre de vie
    SCORE_DISPLAY,  // Affichage du score
    BUTTON_START,   // Bouton démarrer
    BUTTON_QUIT,    // Bouton quitter
    COCONUTS,       // Compteur de noix de coco
    LOGO,           // Logo du jeu
    LOADER          // Écran de chargement
};
```

UIElementTextureType

```
enum class UIElementTextureType {
    STATIC,     // Texture statique simple
    SPRITESHEET // Feuille de sprites avec animations
};
```

UIElementPosition

```
enum class UIElementPosition {
    TOP_LEFT_CORNER,     // Coin supérieur gauche
    TOP_RIGHT_CORNER,    // Coin supérieur droit
    BOTTOM_LEFT_CORNER,  // Coin inférieur gauche
    BOTTOM_RIGHT_CORNER, // Coin inférieur droit
    CENTER               // Centre de l'écran
};
```

CONFIGURATION DES REGLES DE GENERATION

1. Configuration de Base du Spawn

- `SpawnType spawnType` - Type de spawn (ELEMENT, ENTITY, BLOCK)
- `std::vector<ElementName> spawnElements` - Éléments à spawner (équiprobables si multiples)
- `std::vector<EntityName> spawnEntities` - Entités à spawner
- `std::vector<BlockName> spawnBlocks` - Types de blocs sur lesquels spawner

2. Probabilité et Contraintes de Spawn

- `int spawnChance` - Probabilité 1/spawnChance (ex: 50 = 1/50 chance)
- `int maxSpawns` - Nombre maximum total de spawns pour cette règle

3. Contraintes de Distance

- `float minDistanceFromSameRule` - Distance minimale entre spawns de la même règle
- `float maxDistanceFromBlocks` - Distance maximale des blocs spécifiés (0 = pas de contrainte)
- `std::vector<BlockName> proximityBlocks` - Types de blocs dont il faut être proche

4. Spawn en Groupe

- `bool spawnInGroup` - Activer le spawn en groupe
- `float groupRadius` - Rayon pour le spawn en groupe
- `int groupNumberMin/Max` - Nombre min/max d'éléments par groupe

5. Propriétés des Éléments

- `float scaleMin/Max` - Multiplicateurs d'échelle min/max
- `float baseScale` - Échelle de base avant variation aléatoire
- `float rotation` - Rotation en degrés (0 = pas de rotation, -1 = aléatoire)

6. Propriétés des Spritesheets

- `int defaultSpriteSheetPhase/Frame` - Phase et frame par défaut
- `bool randomDefaultSpriteSheetPhase` - Randomiser la phase pour les entités
- `bool isAnimated` - Élément animé
- `float animationSpeed` - Vitesse d'animation

7. Ancrage et Positionnement

- `AnchorPoint anchorPoint` - Point d'ancrage
- `float additionalXAnchorOffset/YAnchorOffset` - Offsets d'ancrage additionnels

8. Stratégie de Placement

- `bool randomPlacement` - Utiliser une sélection aléatoire au lieu de séquentielle
- `std::string ruleName` - Nom optionnel pour le débogage