

Mini Projet

Casse brique

Abdelkader Gouaïch

2023/2024

Introduction

La séance présente un mini-projet qui sera évalué et noté.

Le projet a pour but de consolider votre compréhension des concepts clés que nous avons abordés dans ce module, notamment le rendu graphique avec WebGL et le système Entity-Component-System (ECS).

Votre mission consiste à développer (ou étendre) l'implémentation d'un jeu de type casse-brique, un classique indémodable!

Une logique de jeu de base est proposée avec un rendu graphique en HTML/CSS. Votre tâche principale sera de proposer un système de rendu en utilisant WebGL. Vous pouvez utiliser pour cela les concepts et fonctions présentés lors des séances précédentes.

Une fois cette étape accomplie, nous vous proposons d'améliorer cette première implémentation avec un système de gestion de la défaite et des variations dans la mécanique de jeu.

Mais avant cela, nous allons rappeler les principes d'ECS et présenter par la suite la conception du jeu casse-brique.

Rappels sur Entity-Component-System (ECS)

Le système Entity-Component-System (ECS) est une architecture de programmation largement utilisée dans le développement de jeux vidéo. Elle repose sur les concepts clés suivants : Entity, Component et System.

Entity (Entité)

Une entité est un objet concret ou abstrait dans le jeu. Elle est souvent représentée par un identifiant unique ou un conteneur léger. Dans l'ECS, une entité est essentiellement un container d'identifiants de composants. Elle sert de support

pour associer différents composants et ne contient aucune logique de programmation.

Component (Composant)

Un composant est un conteneur de données qui représente un aspect spécifique d'une entité, comme sa position, sa vitesse etc. Les composants sont des structures de données pures mais sans logique de programmation.

Les composants sont utilisés pour caractériser les entités afin qu'elles puissent exhiber des comportements. Par exemple, une entité avec des composants de position, de vitesse et de rendu pourrait représenter un objet graphique en mouvement.

System (Système)

Un système est dynamique qui traite des entités ayant un ensemble spécifique de composants. Par exemple, un système de rendu pourrait traiter toutes les entités ayant des composants de position et de graphique. Les systèmes sont le cœur de la dynamique du jeu dans l'architecture ECS. Chaque système est responsable d'une fonctionnalité spécifique et opère sur les entités qui ont les composants requis pour cette fonctionnalité.

Conception du jeu Casse-Brique

Les composants

Les composants utilisés pour le jeu sont définis dans le fichiers `components.js`. Voici une brève description de ces composants.

PositionComponent

Ce composant stocke la position d'une entité dans l'espace de jeu. Les paramètres `x`, `y`, `z` représentent les coordonnées de l'entité. Les paramètres `min_x`, `min_y`, `min_z`, `max_x`, `max_y`, `max_z` définissent les limites dans lesquelles l'entité peut se déplacer. Cela est utile pour empêcher les entités de sortir de l'aire de jeu.

GraphicsComponent

Ce composant contient des informations sur l'apparence graphique de l'entité. `shape` est une description de la forme de l'entité (par exemple, "rectangle"), et `shapeInfo` contient des détails spécifiques à cette forme, comme les dimensions et la couleur.

LifeComponent

Ce composant est utilisé pour gérer la vie ou la durabilité d'une entité. **maxLife** est la vie maximale, et **life** est la vie actuelle. Ce composant est particulièrement pertinent pour les briques qui peuvent être détruites après un certain nombre de coups.

GameStateComponent

Ce composant gère l'état global du jeu, comme s'il est en cours d'exécution, en pause, ou terminé. **state** peut avoir des valeurs comme 'running', 'paused', 'gameover'. **hits** compte le nombre de collisions, et **leftControl** et **rightControl** sont utilisés pour gérer les entrées du joueur.

CollisionBoxComponent

Ce composant définit la zone de collision d'une entité. **width** et **height** sont les dimensions de la boîte de collision. **hit** est un indicateur pour savoir si cette entité a récemment été impliquée dans une collision.

VelocityComponent

Ce composant stocke la vitesse de l'entité, avec **dx** et **dy** représentant la vitesse dans les directions x et y, respectivement.

PhysicsTag, CollisionTag, RenderableTag

Ces "tags" sont des marqueurs utilisés pour catégoriser les entités. Ils n'ont pas de données associées mais servent à identifier rapidement les entités qui doivent être traitées par certains systèmes (par exemple, physique, collision, rendu).

BallTag, RaquetteTag, BriqueTag, MurTag

Ces tags sont utilisés pour identifier le type spécifique d'une entité. Par exemple, **BallTag** est utilisé pour marquer l'entité comme étant la balle du jeu, **RaquetteTag** pour la raquette, **BriqueTag** pour les briques, et **MurTag** pour les murs ou les limites du jeu.

les entités

Les entités sont créées par des fonctions dans le fichier **entities.js**

Ball (Balle)

La fonction **Ball** crée une entité représentant la balle dans le jeu. Elle possède les composants suivants :

- **BallTag** : Identifie l'entité comme étant la balle.

- **PhysicsTag** : Indique que la balle est soumise à la physique, comme le mouvement et les collisions.
- **PositionComponent** : Stocke la position actuelle de la balle.
- **VelocityComponent** : Gère la vitesse de la balle, avec des vitesses initiales dx et dy .
- **RenderableTag** : Marque la balle comme étant visible à l'écran.
- **GraphicsComponent** : Définit l'apparence graphique de la balle (forme, taille, couleur).
- **CollisionTag** : Indique que la balle peut entrer en collision avec d'autres objets.
- **CollisionBoxComponent** : Définit la zone de collision de la balle, légèrement plus petite que ses dimensions réelles pour un gameplay plus fluide.

Raquette

La fonction **Raquette** crée l'entité de la raquette contrôlée par le joueur. Ses composants sont :

- **RaquetteTag** : Identifie l'entité comme étant la raquette.
- **PhysicsTag** : Applique la physique à la raquette.
- **PositionComponent** : Définit la position de la raquette.
- **RenderableTag** : Permet le rendu de la raquette à l'écran.
- **GraphicsComponent** : Spécifie l'apparence graphique de la raquette.
- **CollisionTag** : Permet à la raquette d'entrer en collision avec la balle.
- **CollisionBoxComponent** : Définit la zone de collision de la raquette.

Brique

La fonction **Brique** crée une entité représentant une brique dans le jeu. Elle comprend :

- **BriqueTag** : Identifie l'entité comme une brique.
- **PositionComponent** : Stocke la position de la brique.
- **RenderableTag** : Rend la brique visible à l'écran.
- **GraphicsComponent** : Définit l'apparence de la brique.
- **CollisionTag** : Permet à la brique d'entrer en collision avec la balle.
- **CollisionBoxComponent** : Définit la zone de collision de la brique.

Mur

La fonction **Mur** crée une entité représentant un mur ou une limite dans le jeu. Ses composants sont :

- **MurTag** : Identifie l'entité comme un mur.
- **PositionComponent** : Définit la position du mur.
- **RenderableTag** : Assure que le mur est rendu visuellement.
- **GraphicsComponent** : Spécifie l'apparence du mur.
- **CollisionTag** : Permet au mur d'entrer en collision avec d'autres objets.

- **CollisionBoxComponent** : Définit la zone de collision du mur.

GameState

La fonction **GameState** crée une entité pour gérer l'état global du jeu. Elle contient :

- **GameStateComponent** : Stocke des informations sur l'état du jeu, comme l'état actuel (en cours, pause, game over), le score, etc.

Les systèmes

Les systèmes utilisés se trouvent dans les fichiers: `collisionSystem.js`, `gameoverSystem.js`, `hudSystem.js`, `inputSystem.js`, `physicsSystem.js`, `renderSystemCSS.js`.

collisionSystem

Le **collisionSystem** est responsable de la détection et de la gestion des collisions entre les entités. Ce système examine généralement les entités qui ont des composants de collision (comme **CollisionBoxComponent**) et vérifie si leurs zones de collision se chevauchent. Lorsqu'une collision est détectée, ce système :

- Mettre à jour l'état des entités impliquées (par exemple, réduire la vie d'une brique ou la retirer).
- Déclencher des événements ou des actions (comme rebondir la balle).
- Mettre à jour le score ou d'autres éléments de jeu en fonction de la collision.

gameoverSystem

Le **gameoverSystem** surveille les conditions de fin de jeu. Ce système vérifie les conditions de "game over", comme la perte de toutes les vies ou la fin du temps imparti. Lorsque les conditions de "game over" sont remplies, ce système va:

- Emettre un événement de "gameover".
- Arrêter ou modifier la logique du jeu.

hudSystem

Le **hudSystem** (Head-Up Display System) est chargé de gérer l'interface utilisateur qui s'affiche à côté du jeu, comme les scores, les vies restantes, et d'autres informations en temps réel. Ce système peut :

- Mettre à jour le score affiché à l'écran.
- Afficher le nombre de vies restantes.
- Montrer des messages ou des alertes en jeu.

physicsSystem

Le `physicsSystem` gère la physique du jeu, comme le mouvement, la vitesse, et l'accélération des entités. Ce système utilise souvent des composants comme `VelocityComponent` et `PositionComponent` pour :

- Mettre à jour la position des entités en fonction de leur vitesse.

renderSystemCSS

Le `renderSystemCSS` est responsable du rendu visuel des entités en utilisant CSS.


Notre système est très spécifique avec un rendu effectué via le DOM et les styles CSS.

Il peut :

- Créer et mettre à jour des éléments HTML pour représenter les entités.
- Appliquer ou modifier des styles CSS pour ces éléments (comme la position, la taille, la couleur).
- Gérer les animations ou les transitions visuelles.

Votre objectif pour ce projet est de remplacer ce système de rendu avec `renderSystem.js` qui utilisera WebGL.

Travail à réaliser

 **Note importante :** Le répertoire de travail pour cette section est `step0`.

Préparation

 **A faire :**

- Lancez le jeu en ouvrant la page `index.html` avec `liveServer`.
- Consultez le code des fichiers `myGame.js` et `engine.js`
- Consultez le code des fichiers des systèmes
- Consultez le code des fichiers des entités
- Consultez le code des fichiers des composants

Le module de rendu

 **A faire :**

- Consultez en détail le code du système de rendu en CSS `renderSystemCSS.js`
- Proposez votre implémentation du système de rendu en WebGL dans `renderSystem.js`
- Modifiez la fonction `function init(htmlCanvasID)` dans `engine.js` pour remplacer l'ancien système de rendu avec le votre
- Testez votre application

La gestion du gameover

A faire :

- Vous remarquez que le joueur ne peut pas perdre pour l'instant
- Proposez des entités pour gérer la perte de balle (si la raquette rate la balle)
- Proposer une gestion de vies pour le joueur
- Proposer une gestion du gameover si le joueur n'a plus de vie.

Variations créatives

A faire :

- Proposez des variations de la mécanique de ce jeu élémentaire (plusieurs balles, des briques mouvantes etc...)
- Proposez des effets visuels pour améliorer l'expérience de jeu