20/12/2024

RAPPORT PROJET JEU VIDEO

Documentation technique

Siaka DOSSO
Marius ADJAKOTAN
Joran MONNERON
Nadjidatie ABDALLAH HOUSSOUNY

ENTITIES

I. Classe Avatar

Représente le personnage jouable, avec ses animations, comportements (sauter, glisser, courir) et interactions avec l'environnement.

Méthodes principales :

1. Constructeur Avatar(...)
    - Initialise l'avatar en définissant sa position(position(2 ,188)), ses sons (saut et collision), ses animations, et son état initial (repos).
    - Crée des animations (running et sliding) en découpant une feuille de sprites.
2. StartJump() et CancelJump()
    - Permettent de gérer le saut lorsque le user click direction-haut et d'interrompre celui-ci si certaines conditions sont réunies comme le temps épuisé qui correspond à la durée de passage de l'obstacle ou click direction-bas.
    - Simule un comportement réaliste avec gestion de la vélocité verticale et de la gravité.
3. Slide() et GetUp()
    - Contrôlent la transition entre un état normal et l'état Sliding.
4. Update(GameTime gameTime)
    - Met à jour la position, l'animation, et la vitesse de l'avatar.
    - Applique la gravité si en chute ou saut et gère les transitions d'état (Falling, Running).
5. Draw(...)
    - Affiche l'avatar selon son état actuel.
6. Die()
    - Gère la mort de l'avatar après une collision, arrête sa vitesse, et notifie via un événement.

II. Classe GroundObstacle

Représente les obstacles au sol que l’avatar doit éviter.

Méthodes principales :

1. Constructeur GroundObstacle(...)
    - Crée un obstacle selon son type (Panel, StopPanel, ou Trash), en définissant ses dimensions et sa position.
2. Update(GameTime gameTime)
    - Met à jour la position horizontale de l’obstacle, en fonction de la vitesse de l’avatar.
3. Draw(...)
    - Affiche l'obstacle à l'écran.
4. Propriété CollisionBox
    - Retourne un rectangle de collision pour savoir quand le joueur touche l’objet

Choix :

Les dimensions et positions des textures sont définies de manière statique pour chaque type d'obstacle, simplifiant le rendu.

III. Classe GroundTile

Gère le sol sur lequel l’avatar court, pour créer une impression de mouvement.

Méthodes principales :

1. Constructeur GroundTile(...)
    - Crée un sol en fonction de la position et des dimensions fournies.
2. Update(GameTime gameTime)
    - Déplace la tuile vers la gauche, en fonction de la vitesse de l'avatar.
3. Draw(...)
    - Affiche la tuile.

IV. Classe Obstacle (Abstraite)

Servir de classe de base pour tous les types d'obstacles.

Méthodes principales :

1. Update(GameTime gameTime)
    - Met à jour la position de l'obstacle et vérifie les collisions via un CollisionManager.
2. Draw(...) (Abstraite)
    - Oblige les sous-classes à implémenter leur propre méthode d'affichage.
3. Propriété CollisionBox (Abstraite)
    - Oblige les sous-classes à définir leur propre zone de collision.

Choix :

Favorise la réutilisation et la spécialisation avec des sous-classes comme GroundObstacle.

V. Classe ScoreBoard

Affiche le score actuel du joueur et le score maximal enregistré dans un fichier XML.

Méthodes principales :

1. Constructeur ScoreBoard(...)
    - Charge le score le plus élevé depuis un fichier XML.
    - Initialise les sprites pour l'affichage des scores.
2. Update(GameTime gameTime)
    - Met à jour le score en fonction de la vitesse de l'avatar et du temps écoulé.
3. Draw(...)
    - Affiche le score actuel et le meilleur score, sous forme de chiffres.
4. GetHighScoreFromXmlFile()
    - Extrait le score maximal depuis le fichier XML.
5. DrawScore(...)
    - Découpe les chiffres du score en unités séparées et les dessine.

- **Choix Techniques :**

1. Séparation des responsabilités :
    - Chaque classe gère un élément spécifique (avatar, obstacles, sol, scores).
    - Facilite la maintenance et l'extension du code.
2. Utilisation de constantes :
    - Centralise les valeurs de dimensions et de positions, rendant le code plus lisible.
3. Fichier XML pour les scores :
    - Permet de persister les données entre les sessions, avec un format standard facile à manipuler.

Graphics

I. Classe Sprite

La classe Sprite représente une entité graphique de base, une image ou une texture à afficher à l'écran. Elle est essentielle pour gérer les éléments visuels comme les personnages, les objets ou les arrière-plans.

Explication des méthodes et propriétés :

- Propriétés :
    - X, Y : Les coordonnées de la texture dans une grande image source (spritesheet). Cela permet de sélectionner une région spécifique d'une image contenant plusieurs sprites.
    - Width, Height : La taille de la région sélectionnée, utilisée pour découper la bonne portion de l'image.
    - Texture : La texture source contenant l'image ou les images à afficher.
    - TintColor : Permet de modifier la couleur ou la transparence de l'image (blanc par défaut pour ne pas appliquer de changement).
- Constructeur :
    - **Sprite(Texture2D texture, int x, int y, int width, int height)**
      Initialise un sprite avec une texture spécifique, une position dans cette texture (X, Y), et une taille (Width, Height).
      Cela permet de réutiliser une seule grande image pour de nombreux sprites, optimisant l'utilisation des ressources.

- Méthode Draw :
    - **Draw(SpriteBatch spriteBatch, Vector2 position)**
      Affiche le sprite à l'écran à une position donnée. Utilise un rectangle pour découper la bonne portion de la texture et applique une couleur (optionnelle).

II. Classe SpriteAnimationFrame

Cette classe représente une étape d'une animation. Chaque frame est associée à un sprite spécifique et à un moment donné dans le temps.

Explication des méthodes et propriétés :

- **Propriétés :**
    - Sprite : Le sprite affiché pour cette frame. Une exception est levée si la valeur est null, garantissant que chaque frame est valide.
    - TimeStamp : Le temps relatif où cette frame doit être affichée. Cela permet de contrôler précisément la durée de chaque frame dans l'animation.
- **Constructeur :**
    - **SpriteAnimationFrame(Sprite sprite, double timeStamp)**
      Initialise une frame avec un sprite et un timestamp, permettant d'organiser l'animation dans le temps.

Pourquoi ce choix :

Crée une classe dédiée aux frames permet de séparer clairement les données d'animation (temps et sprite) et de faciliter leur gestion dans des animations complexes.

III. Classe SpriteAnimation

Gère l'animation composée de plusieurs frames. Cette classe permet de contrôler l'ordre, le timing et l'affichage des sprites dans une séquence animée.

Explication des méthodes et propriétés :

- **Propriétés :**
    - _frames : Liste des frames de l'animation, contenant les sprites et leurs timestamps associés.

- IsPlaying : Indique si l'animation est en cours d'exécution.
- PlayBackProgess : Temps écoulé depuis le début de l'animation, utilisé pour déterminer la frame actuelle.
- CurrentFrame : Retourne la frame qui doit être affichée en fonction du temps actuel.
- Duration : Temps total de l'animation, calculé à partir des timestamps des frames.

- **Méthodes principales :**
  - **AddFrame(Sprite sprite, double timeStamp)**
    Ajoute une nouvelle frame à l'animation avec un sprite et un timestamp. Cela permet de construire dynamiquement une animation.
  - **Update(GameTime gameTime)**
    Met à jour la progression de l'animation en fonction du temps écoulé (gameTime). Si la progression dépasse la durée totale, elle revient au début (loop).
  - **Draw(SpriteBatch spriteBatch, Vector2 position)**
    Dessine la frame actuelle à la position spécifiée. Cela garantit que seule la bonne image est affichée, même dans des animations complexes.
  - **Start() et Stop()**
    - Start() démarre ou redémarre l'animation.
    - Stop() arrête l'animation et réinitialise le temps écoulé.
  - **GetFrame(int index)**
    Récupère une frame spécifique de la liste. Une exception est levée si l'index est invalide, garantissant la robustesse.

Pourquoi ces choix ?

1. **Séparation des responsabilités :**
   - La classe Sprite est dédiée à la gestion d'une seule image. Cela permet de réutiliser ce concept pour des animations ou d'autres entités.
   - La classe SpriteAnimationFrame simplifie la gestion de chaque étape d'une animation. En la séparant, le code est plus clair et modulaire.

- La classe SpriteAnimation encapsule toute la logique d'une séquence animée (ajout de frames, gestion du temps, affichage).

2. **Facilité d'extension :**
   - Ce design est facilement extensible. Par exemple, il serait simple d'ajouter des transitions, des effets ou des événements entre frames.
3. **Performance :**
   - En pré-calculant et en organisant les frames, l'animation peut être affichée rapidement à chaque mise à jour, ce qui est crucial pour des performances en temps réel.
4. **Flexibilité :**
   - L'utilisation des timestamps permet de créer des animations avec des rythmes variés (par exemple, certaines frames peuvent durer plus longtemps que d'autres).
5. **Réutilisabilité :**
   - Ce système peut être utilisé pour divers types d'animations : personnages, objets, effets visuels, etc.

MANAGERS

I. Classe EntityManager

Le gestionnaire d'entités est responsable de la gestion centralisée de tous les objets dans le jeu, comme les avatars, obstacles, ou autres entités interactives. Il simplifie les opérations globales comme l'ajout, la suppression, ou l'actualisation d'entités.

Méthodes et propriétés :

- **Propriétés principales :**
  - _entities : Liste principale des entités actives dans le jeu.
  - _entitiesToAdd : Liste temporaire pour les nouvelles entités à ajouter.
  - _entitiesToRemove : Liste temporaire pour les entités à supprimer.
- **Méthodes principales :**
  - **AddEntity et RemoveEntity :**
    Ces méthodes permettent de gérer dynamiquement les entités. L'utilisation de

listes temporaires évite les conflits ou erreurs liés à la modification de la liste principale lors de l'itération.

- **UpdateEntities :**
  Met à jour toutes les entités actives en appelant leur méthode Update. Après l'itération, ajoute les entités en attente et supprime celles marquées pour suppression. Cela garantit une gestion cohérente et évite les modifications simultanées.
- **DrawEntities :**
  Dessine toutes les entités à l'écran, ordonnées par leur priorité d'affichage (DrawOrder).
- **ClearEntities :**
  Vide toutes les entités de la liste principale en les ajoutant directement à _entitiesToRemove.

Pourquoi ce choix :

- Une gestion centralisée des entités permet une meilleure organisation du code et facilite l'extensibilité.
- La séparation entre les entités en cours d'ajout/suppression et celles déjà actives garantit la stabilité du programme.

II. Classe GroundManager

Gère les tuiles de sol pour le jeu, notamment leur création, positionnement, mise à jour et réapparition lorsqu'elles quittent l'écran. Cela donne une illusion de déplacement continu.

Méthodes et propriétés :

- **Propriétés principales :**
  - _groundTiles : Liste de toutes les tuiles de sol actives.
  - _random : Générateur de nombres aléatoires utilisé pour mélanger ou positionner les tuiles.
  - _avatar : Référence à l'avatar du joueur pour gérer les interactions.

- **Méthodes principales :**
  - **CreateGroundTiles :**
    Crée un ensemble de tuiles de sol avec des positions et dimensions initiales.

- **Initialize :**
  Mélange les tuiles aléatoirement et les positionne de manière séquentielle, mais dans un ordre mélangé pour plus de variété visuelle.
- **Update :**
  Actualise la position des tuiles. Si une tuile sort de l'écran, elle est repositionnée à la droite de l'écran via SpawnTile.
- **SpawnTile :**
  Repositionne une tuile à droite, en choisissant aléatoirement son apparence parmi les tuiles disponibles.
- **Draw :**
  Dessine toutes les tuiles à l'écran.

Pourquoi ce choix :

- La réutilisation et le repositionnement des tuiles améliorent les performances en évitant la création/destruction fréquente d'objets.
- L'utilisation aléatoire des tuiles ajoute de la variété et maintient l'intérêt visuel.

III. Classe ObstacleManager

Gère les obstacles dans le jeu, notamment leur création, positionnement, apparition aléatoire, et suppression lorsqu'ils sortent de l'écran.

Méthodes et propriétés :

- **Propriétés principales :**
  - _obstacles : Liste des obstacles actifs.
  - _removedObstacles : Liste des obstacles supprimés.
  - _scoreBoard : Référence au tableau de score pour déclencher la création des obstacles en fonction du score.
  - _random : Générateur aléatoire pour choisir le type et la position des obstacles.
  - _currentTargetDistance : Distance cible pour la prochaine apparition d'obstacles.

Méthodes principales :

- **SpawnRandomObstacle :**
  Crée un obstacle aléatoire avec un type, une position et des propriétés spécifiques. Par exemple, les panneaux ou poubelles ont des hauteurs différentes.
- **Update :**
  Actualise les positions des obstacles et vérifie s'ils sont hors de l'écran pour les supprimer. Gère aussi la création d'obstacles en fonction de la progression du score.
- **RemoveAllObstacles :**
  Ajoute tous les obstacles existants à la liste des obstacles supprimés.
- **Initialize :**
  Réinitialise les variables internes pour permettre un nouveau départ.
- **Draw :**
  Dessine tous les obstacles sur l'écran.

Pourquoi ce choix :

- La gestion centralisée des obstacles facilite l'ajout ou la suppression d'obstacles sans interférer avec d'autres éléments du jeu.
- Le calcul dynamique de la distance entre les obstacles permet d'ajuster la difficulté à mesure que le joueur progresse.

IV. Classe CollisionManager

Gère les collisions entre l'avatar et les obstacles. Cette classe est essentielle pour détecter les interactions et déclencher des conséquences comme la fin de partie.

Méthodes et propriétés :

- **Propriétés principales :**
  - _avatar : Référence à l'avatar pour accéder à sa boîte de collision.
  - _obstacle : Référence à un obstacle spécifique pour vérifier les collisions.
- **Méthode principale :**
  - **CheckCollision :**
    Vérifie si la boîte de collision de l'avatar intersecte celle d'un obstacle. Si une collision est détectée, la méthode Die de l'avatar est appelée.

Pourquoi ce choix :

- En déléguant la gestion des collisions à une classe distincte, le code reste modulaire et plus facile à maintenir.
- Cela permet d'ajouter d'autres types de collisions ou d'interactions à l'avenir sans modifier l'avatar ou les obstacles directement.

Technique :

- **Modularité :** Chaque classe a une responsabilité bien définie.
- **Réutilisabilité :** Les gestionnaires comme EntityManager, GroundManager, et ObstacleManager peuvent être réutilisés dans différents contextes.

ScoreBoard

La classe **ScoreBoard** est utilisée pour gérer le score du joueur dans le jeu Human Dash. Elle permet de suivre le score en temps réel, de sauvegarder les records dans un fichier XML via la sérialisation, et d'afficher ces informations à l'écran après récupération via la désérialisation.

Elle assure trois fonctions principales dans le jeu : le calcul des scores, leur sauvegarde, et leur affichage graphique. Chaque méthode contribue à rendre ces opérations fluides et autonomes, garantissant une expérience utilisateur cohérente et agréable.

Fonctionnalités Principales et Utilité

1. **Update(GameTime gameTime)**
   - Met à jour le score en fonction de la vitesse du joueur et du temps écoulé.
   - Le score augmente progressivement selon un multiplicateur utilisant la formule : Score+=_avatar.Speed***SCORE_INCREMENT_MULTIPLIER***gameTime.ElapsedGaTime.To -talSeconds;
   - ce qui reflète la performance du joueur pendant la partie.
2. **Draw(GameTime gameTime, SpriteBatch spriteBatch)**
   - Affiche le score actuel et, si disponible, le record ("HI Score") à l'écran.
   - Utilise des sprites graphiques pour dessiner les chiffres et le label "HI" dans une position définie.

3. **GetHighScoreFromXmlFile()**
   - Charge le score le plus élevé (record) depuis un fichier XML au démarrage du jeu.
   - Lit et analyse le fichier XML pour récupérer la valeur du record, en tenant compte de l'espace de noms XML.
4. **SetHighScore()**
   - Met à jour le fichier XML si le score actuel dépasse le record existant.
   - Compare le score actuel au record et, si nécessaire, sauvegarde le nouveau record dans le fichier XML.
5. **SplitDigits(int input)**
   - Transforme un score en une liste de chiffres pour un affichage individuel.
   - Prépare chaque chiffre pour être dessiné à sa position respective à l'écran.
6. **DrawScore(SpriteBatch spriteBatch, int score, float startPosX)**
   - Affiche un score donné, chiffre par chiffre, en utilisant des coordonnées de texture spécifiques.

Merci pour la lecture !