INF707 - Projet de session
Volet 1

Léa Arnaud
Jos Landuré
Erwan Sénéchal
Albin Calais

1 Présentation générale

Notre jeu a comme nom provisoire "Celespeed".

> Dans un univers futuriste, sur une autre planète, une route se trouve devant vous. Ce qui vous motive, comme tous les autres ? L'adrénaline de la vitesse... mais cette route ne se laissera pas dompter facilement.

Le but de Celespeed est de réussir à gérer la très grande vitesse de votre véhicule afin de terminer la course le plus rapidement possible. Pour cela, il faudra éviter les obstacles, gérer les sols et leurs différentes propriétés ainsi que gérer le circuit dans son ensemble.

Actuellement, le jeu n'est pas prévu pour avoir des adversaires (IA). Pour l'instant le gameplay sera du "contre la montre".

2 Révision des contraintes et objectifs

Le terrain sera affiché en utilisant Blinn-Phong, en utilisant plusieurs textures à différents endroits.

Les textures auront pour la plupart des niveaux de mipmap générés au chargement.

La course comportera un tunnel passant dans des bâtiments, à l'intérieur la lumière sera calculée dynamiquement avec un certain nombre de point-lights en plus des effets appliqués à toute la scène (bloom...). La lumière du soleil (directional light) ne sera plus considérée dans le tunnel. La transition sera faite de manière agréable visuellement.

La scène sera délimitée par des crevasses, lorsque le joueur tombera hors de la scène il sera ramené sur la course.

Le menu pause sera accessible depuis le menu principal et depuis la course, cela sera abordé plus tard.

Des informations de la course (vitesse du joueur, temps restant...) seront affichées en tout temps soit en 2D sur l'écran soit diégétiquement intégré sur le véhicule directement.

Une skybox sera visible de manière permanente, elle ne sera jamais affectée par les lumières de la scène et sera affichée en dernier dans la render-pipeline pour éviter l'overdraw.

Des billboards seront utilisés pour afficher des panneaux animés sur les bâtiments.

Pour les points de complexité, nous implémenterons :

- (2) VFX Ribbon - pour les fumées du véhicule
- (2) MSAA - utilisé partout (scène de jeu et menus)
- (2) Bloom - utilisé dans la totalité de la scène de jeu (rendu en HDR)
- (1+1) Screen shake et Lens distorsion - pour accentuer les effets de vitesse

3 Modules

Module graphique : le module graphique est responsable de toute la partie affichage, c'est le seul qui interagit avec la librairie graphique (DirectX). Il contient des abstractions des fonctionnalités DirectX (index/vertex/constant buffer, textures, shaders...). Il contient aussi toute la partie UI (affichage du texte, éléments UI interactifs...).

Module physique : le module physique est responsable du "monde physique". Les objets du jeu ont une existence dans la scène (WorldObject) et peuvent avoir une existence dans le monde physique indépendante (PhysicsObjects). La scène est responsable de mettre à jour ses objets en se basant sur le monde physique. Ce module est le seul à interagir directement avec la librairie physique (PhysX).

Module de scène : le module de scène contient toute la partie jeu, ce serait la partie du user-code dans un game-engine. Il est responsable des interactions avec le joueur, de l'utilisation des deux autres modules pour afficher et animer les objets.

Voir les dernières pages pour les structures détaillées des modules (format UML).

4 Liens entre les modules

Le module des scènes utilise les deux autres mais les dépendances sont à sens unique. Et il n'y a pas de lien entre le module physique et celui des graphismes. Dans le module graphique, seules certaines parties de haut niveau sont accessibles à l'extérieur. On ne manipulera donc pas de GenericBuffer mais plutôt des Meshs, et on utilisera rarement un SpriteManager mais on préférera utiliser un UIManager.
De la même manière, dans la scène on ne manipulera pas ou peu d'Agent physique, mais on dialoguera avec le monde physique via des PhysicsBody.

L'objectif général est de minimiser le couplage des modules.

Nous ne sommes pas encore très au point sur le traitement parallèle (double/triple buffering), il y aura certainement des changements à faire pour gérer les problèmes de synchronisation et de multithreading.

5 Lien avec le module de physique

L'objectif du module physique est d'abstraire la librairie physique (PhysX) pour que le module de Scène n'ait pas à manipuler les objets PhysX directement. Pour-ce faire, nous utiliserons une classe intermédiaire PhysicsBody, c'est le PhysicsBody qui créé les PxAgent. Pour faire le lien depuis la simulation vers les PhysicsBody on pourra passer soit par un système d'ID, soit par PxShape::userData qui est un pointeur arbitraire pensé précisément pour ce genre d'utilisations.

Il est probable que nous n'ayons pas le temps de construire une abstraction complète de PhysX, dans un premier temps nous nous permettrons d'utiliser les fonctions de PhysX dans certaines classes du module de Scène (celles héritant de WorldObject) pour créer les agents.

La simulation peut théoriquement tourner en parallèle du rendu, nous essayerons d'implémenter le double/tripple-buffering via du multithreading.

PhysX offre plusieurs géométries possibles. Pour le terrain la géométrie tout à fait adaptée est le HeightField. Pour la plupart des objets on utilisera des ConvexMesh et des TriangleMesh. Pour les boîtes de déclenchement on utilisera de simples Box.
Pour la gestion des collisions on créera une abstraction des filterShader et simulationEventCallback/contactModifyCallback. De cette manière, la scène n'aura pas à connaître la librairie physique. On s'arrangera tout de même pour que la scène ait le maximum d'information et de contrôle sur les évènements et sur le monde physique en général.

6 Format du terrain

Pour le "terrain", nous découpons le terme en deux catégories :

- Le circuit suspendu dans les airs, qui sera généré sous forme de splines converties en objets 3D directement depuis le jeu
- Le sol de la planète, avec ses crevasses et son bout du circuit, représenté par une heightmap sous le format bitmap qui sera convertie en objet 3D en jeu

Il sera possible de gérer l'affichage en fonction de la distance, qui sera particulièrement grande dans notre jeu.

7 Objet Caméra

La caméra sera attachée au joueur et possédera plusieurs attributs: sa position, son orientation et des paramètres de projection.

En ce qui concerne l'orientation de la caméra, celle-ci peut être exprimée sous la forme d'angles d'Euler (yaw, pitch, roll) ou de quaternions. Afin de créer une impression de vitesse, nous pouvons modifier le champ de vision (FOV) de la caméra. Augmenter le FOV lorsque le joueur accélère peut donner l'impression de vitesse.
Il est également important de permettre au joueur de passer du point de vue à la première personne à celui à la troisième personne, et vice versa, pour offrir une expérience de jeu polyvalente. Pour cela, nous avons deux options : modifier la position et l'orientation de la caméra en conséquence, ou avoir deux caméras distinctes, une pour la vue à la première personne et une autre pour la vue à la troisième personne. Cette flexibilité donne au joueur le contrôle sur la perspective qui lui convient le mieux.

Enfin, pour que la caméra suive le véhicule de manière fluide, nous pouvons utiliser des techniques de lissage de mouvement. Cela peut être réalisé par interpolation ou damping, ce qui permet d'atténuer les mouvements brusques de la caméra en réponse aux actions du véhicule. Cela garantit une expérience visuelle agréable et immersive pour le joueur tout en suivant le véhicule de manière précise et fluide.

8 Objet Véhicule

Le véhicule est un objet physique un peu particulier puisque c'est le seul qui sera modifié à la fois depuis la scène et depuis la simulation physique. La classe Player sera responsable du contrôle total du véhicule, informée par la classe résponsable de la logique de jeu.

La physique du véhicule sera étudiée de manière à rendre le contrôle agréable. Notre véhicule étant un planneur, nous utiliserons les "spring joint" (ou équivalent PhysX) pour le faire réster à une certaine altitude au dessus de la piste de manière réaliste. La largeur et la longueur de la piste seront directement impactée par la taille du véhicule et sa vitesse de pointe.

9 Gestion des zones (niveau de détail)

Nous allons subdiviser le terrain en plusieurs morceaux, appelés "chunks", pour éviter de le rendre intégralement à chaque instant. Cette subdivision sera réalisée en partant de notre heightmap d'origine. Contrairement à l'utilisation de plusieurs niveaux de détail (LOD) pour nos objets, nous avons opté pour une approche basée sur le culling, en utilisant le frustum culling et le backface culling. Ces techniques nous permettront de n'afficher uniquement que les objets nécessaires à l'écran. De plus, pour la structuration de l'espace, nous avons choisi d'employer des quadtrees plutôt que des octrees, en raison du fait que, même si notre monde est en 3D, notre niveau ne présente pas une grande variation en termes de verticalité.

10 Objets importés - Descriptions

Le Paysage Environnant : Notre environnement comprend une heightmap détaillée qui servira de base pour générer un terrain réaliste. Cette heightmap sera associée à des textures pour créer un monde visuellement riche et varié.

Infrastructure Routière Dynamique : La route principale sera générée en temps réel à partir d'une spline, offrant ainsi des sections variées telles que des virages, des lignes droites, et même des portions surélevées avec des barrières de sécurité.

Le Tunnel Souterrain : Un tunnel intrigant, d'une longueur suffisante ainsi que des lumières intégrées à l'intérieur permettant de respecter les contraintes de conception.

L'Architecture Futuriste : Notre paysage urbain sera dominé par des bâtiments au design futuriste. Des panneaux d'affichage ajouterons un élément de modernité et d'interactivité à notre monde.

Le Véhicule : Notre véhicule sera volant mais plutôt de type planeur. C'est-à-dire que nous n'aurons pas de déplacement sur la hauteur, il suivra simplement la route a une faible distance en hauteur de celle-ci. Même s' il ne possède pas de roues, un effet de suspension se fera sentir en cas changement sur le terrain comme la prise d'un tremplin par exemple. Il possédera donc toutes les contraintes physiques d'un véhicule terrestre.

11 Prototypage des données

Les données qui ne relèvent pas des parties graphique et physique sont principalement les données de logique de jeu. Le SceneManager contient une scène (ou plusieurs lorsqu'il y a chevauchement/transition), chaque scène est responsable de ses propres données. Par exemple, la scène de jeu contiendra les informations relatives à la course (temps total, réglages de vitesse max...), les données nécessaires à la manipulation de la caméra etc...

Pour le chargement des ressources, nous utiliserons des formats largement utilisés : .dds pour les textures avec une librairie de chargement et .obj pour les modèles.

12 Prototypage des traitements

On peut différencier les cas d'utilisation selon la scène active, lorsque le jeu se lance on est dans la scène du menu principal.
Menu principal :

- les meilleurs temps sont affichés
- clic sur le bouton "lancer une partie", transition vers la scène de jeu
- clic sur le bouton "quitter", sauvegarde des données et fin du programme
- clic sur le bouton "options", le menu d'options apparaît par dessus la scène en cours

Scène de jeu :

- le temps de la course est affiché
- touches de mouvement, le joueur contrôle le véhicule
- ¡tab?¿, le joueur change de caméra
- ¡esc¿, le menu d'options apparaît par dessus la scène en cours, avec un bouton supplémentaire pour revenir au menu principal

Menu d'options :

- le meilleur temps est affiché, ainsi que le temps de la course si la scène précédente est celle de jeu
- clic sur le bouton "quitter", le menu disparaît
- click/drag sur les réglages, les différents réglages de jeu sont ajustés

13 Options du jeu

Le menu d'options du jeu pourra s'afficher au dessus de la scène en cours, il contiendra des contrôles pour :

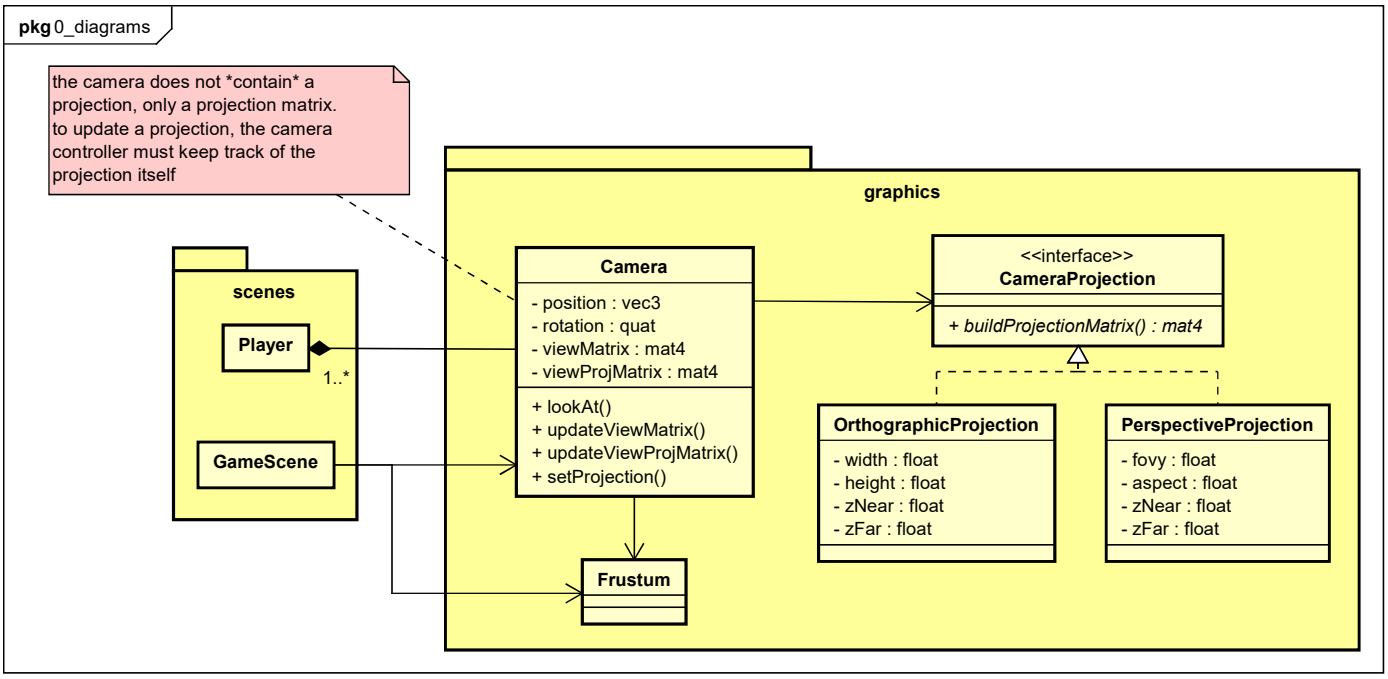
- Activer/Désactiver le multisampling et le bloom
- Modifier la résolution d'affichage et mettre en plein-écran
- Quitter le jeu
- Modifier les contrôles du véhicule
- Le système d'UI du jeu ne sera composé que de boutons, checkboxes et textes. Lorsque le menu sera ouvert, le contrôle sera récupéré par la souris.

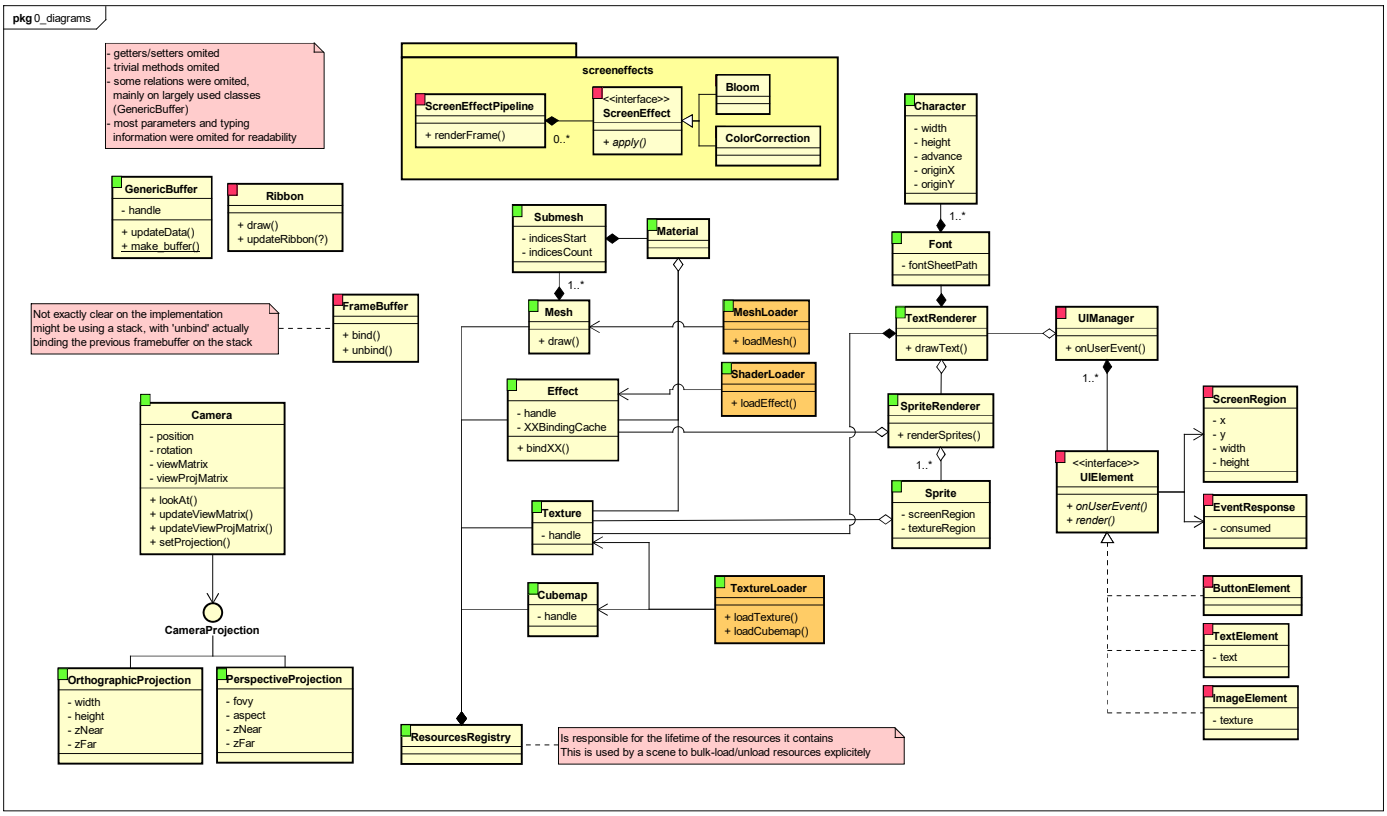
14 Logique du jeu

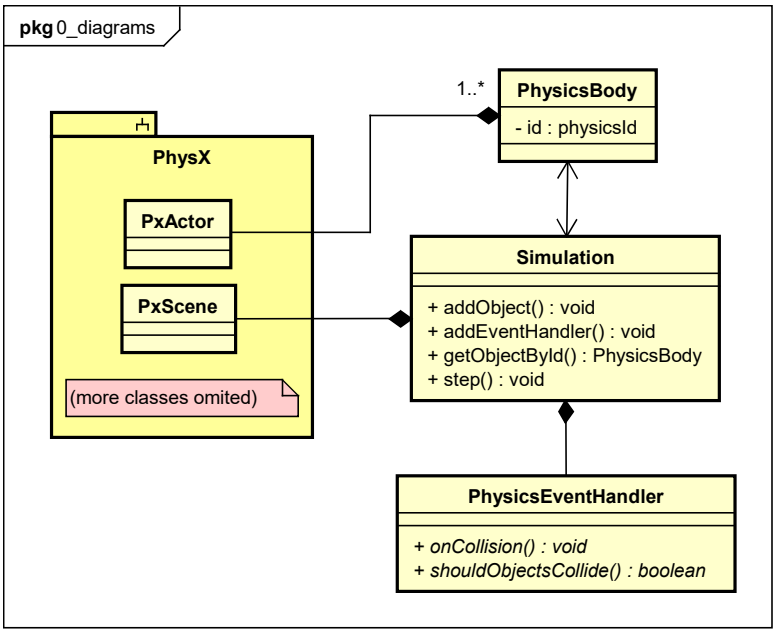
Comme dit dans la présentation générale, le jeu aura un accent très prononcé sur la vitesse.
Le jeu se veut par conséquent très nerveux et rapide, et c'est pourquoi la maniabilité du véhicule sera très grande : les joueurs aux bons réflexes et/ou avec une bonne connaissance du jeu seront récompensés.

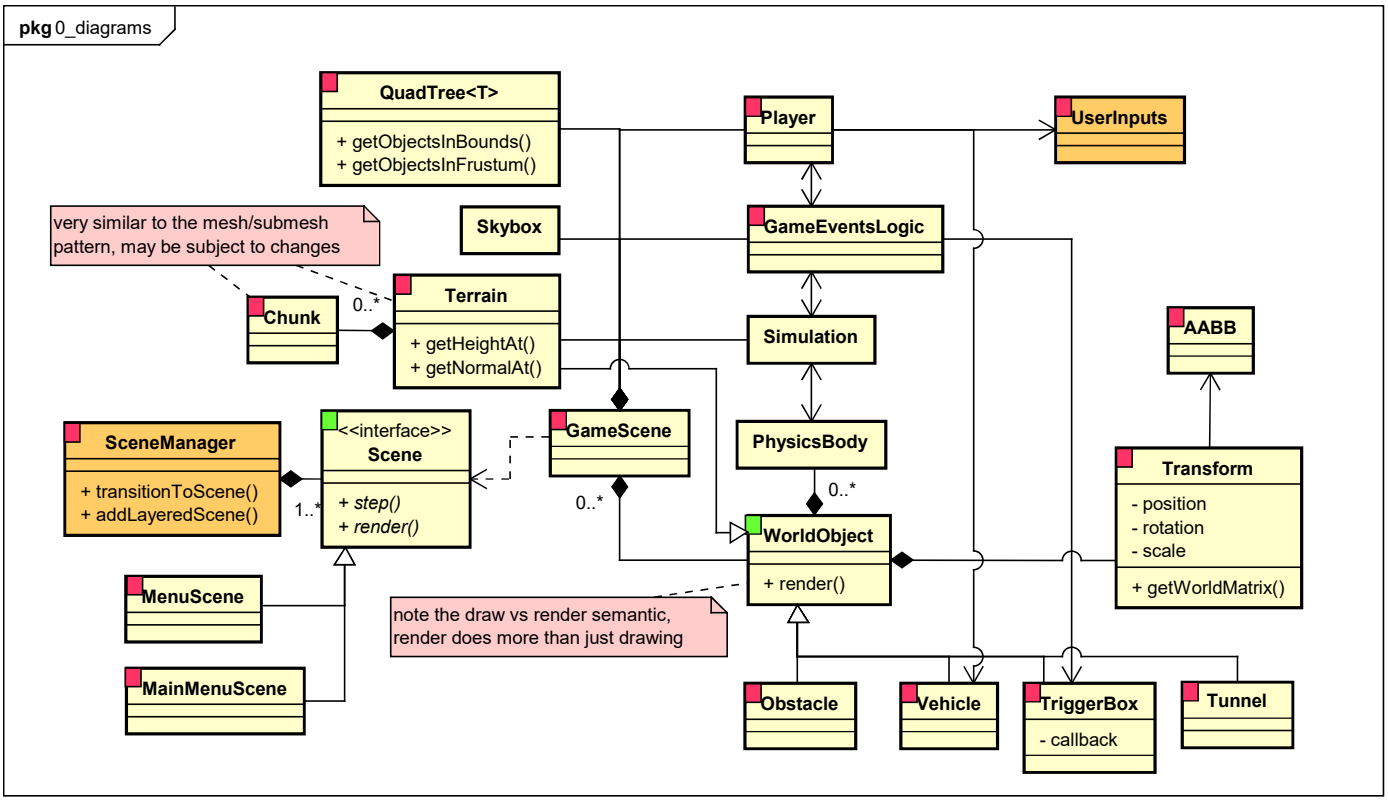
Le véhicule étant un semi-planeur, il ne sera pas soumis directement aux frottements au sol. Par contre, les frottements de l'air s'appliquent toujours et permetteront de décélérer.
De plus, il est possible de faire "sauter" le véhicule par effet ressort du planeur. Cela sera utile en terme de contrôles, mais cette physique influera aussi sur l'atterissage du véhicule.

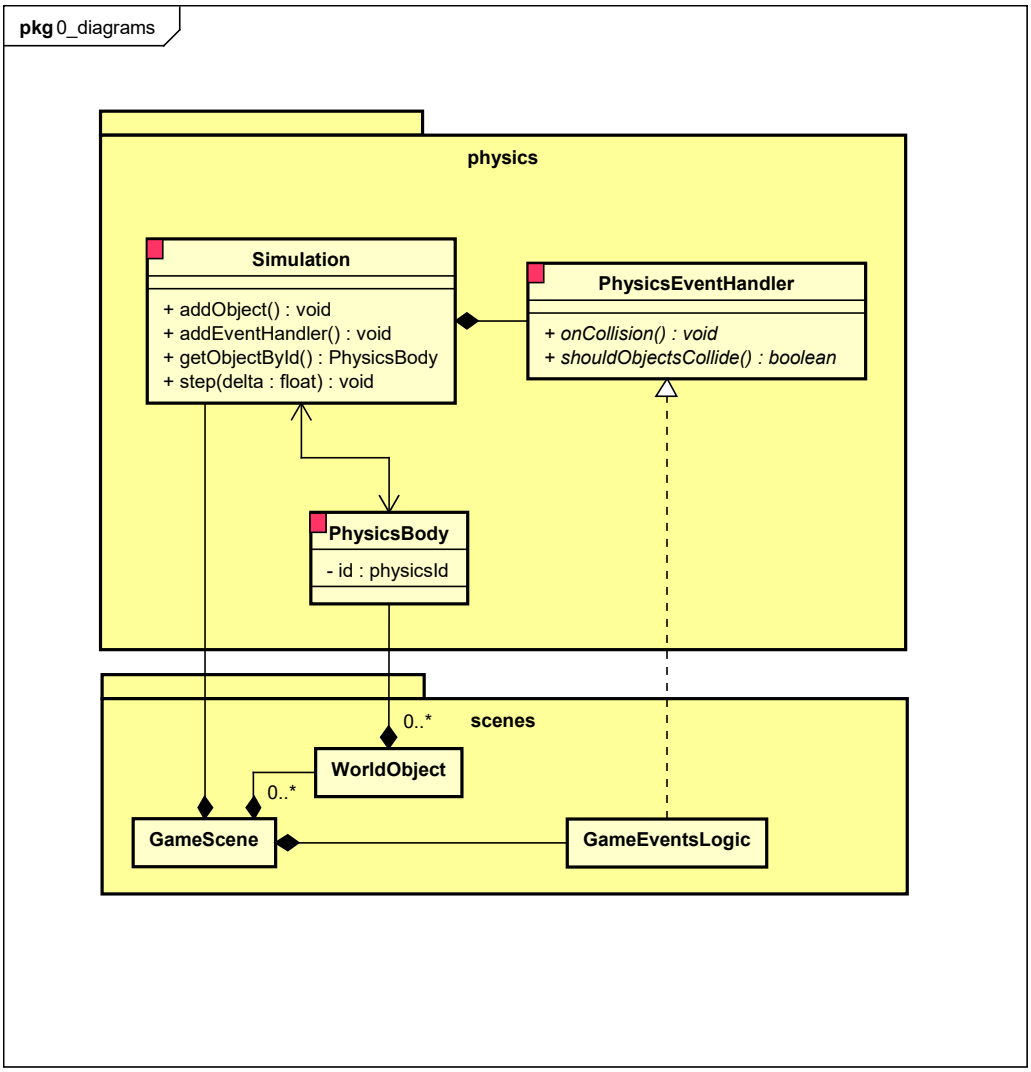
Sur le circuit, il y aura parfois des zones ouvertes et le joueur tombera potentiellement dans le vide. Si cela arrive, il sera ramené sur le circuit. La détection d'un tel accident se fera principalement par un check sur la hauteur. Cela permettra, pour les plus ingénieux, de créer leurs raccourcis en se propulsant à toute vitesse !


pkg 0_diagrams
the camera does not *contain* a projection, only a projection matrix. to update a projection, the camera controller must keep track of the projection itself
graphics
scenes
Player
1..*
GameScene
Camera
- position : vec3
- rotation : quat
- viewMatrix : mat4
- viewProjMatrix : mat4
+ lookAt()
+ updateViewMatrix()
+ updateViewProjMatrix()
+ setProjection()
Frustum
<<interface>>
CameraProjection
+ buildProjectionMatrix() : mat4
OrthographicProjection
- width : float
- height : float
- zNear : float
- zFar : float
PerspectiveProjection
- fovy : float
- aspect : float
- zNear : float
- zFar : float


pkg 0_diagrams
- getters/setters omited
- trivial methods omited
- some relations were omited,
mainly on largely used classes
(GenericBuffer)
- most parameters and typing
information were omited for readability
screeneffects
ScreenEffectPipeline
+ renderFrame()
0..*
<<interface>>
ScreenEffect
+ apply()
Bloom
ColorCorrection
Character
- width
- height
- advance
- originX
- originY
1..*
Font
- fontSheetPath
GenericBuffer
- handle
+ updateData()
+ make_buffer()
Ribbon
+ draw()
+ updateRibbon(?)
Submesh
- indicesStart
- indicesCount
Material
1..*
Mesh
+ draw()
MeshLoader
+ loadMesh()
Not exactly clear on the implementation
might be using a stack, with 'unbind' actually
binding the previous framebuffer on the stack
FrameBuffer
+ bind()
+ unbind()
ShaderLoader
+ loadEffect()
Effect
- handle
- XXBindingCache
+ bindXX()
TextRenderer
+ drawText()
UIManager
+ onUserEvent()
1..*
SpriteRenderer
+ renderSprites()
1..*
Sprite
- screenRegion
- textureRegion
ScreenRegion
- x
- y
- width
- height
Camera
- position
- rotation
- viewMatrix
- viewProjMatrix
+ lookAt()
+ updateViewMatrix()
+ updateViewProjMatrix()
+ setProjection()
Texture
- handle
<<interface>>
UIElement
+ onUserEvent()
+ render()
EventResponse
- consumed
CameraProjection
Cubemap
- handle
TextureLoader
+ loadTexture()
+ loadCubemap()
ButtonElement
TextElement
- text
ImageElement
- texture
OrthographicProjection
- width
- height
- zNear
- zFar
PerspectiveProjection
- fovy
- aspect
- zNear
- zFar
ResourcesRegistry
Is responsible for the lifetime of the resources it contains
This is used by a scene to bulk-load/unload resources explicitely


pkg 0_diagrams
1..*
PhysicsBody
- id : physicsId
PhysX
PxActor
PxScene
(more classes omited)
Simulation
+ addObject() : void
+ addEventHandler() : void
+ getObjectById() : PhysicsBody
+ step() : void
PhysicsEventHandler
+ onCollision() : void
+ shouldObjectsCollide() : boolean


pkg 0_diagrams
QuadTree<T>
+ getObjectsInBounds()
+ getObjectsInFrustum()
Player
UserInputs
very similar to the mesh/submesh pattern, may be subject to changes
Skybox
GameEventsLogic
Chunk
0..*
Terrain
+ getHeightAt()
+ getNormalAt()
Simulation
AABB
SceneManager
+ transitionToScene()
+ addLayeredScene()
1..*
<<interface>>
Scene
+ step()
+ render()
GameScene
0..*
PhysicsBody
0..*
Transform
- position
- rotation
- scale
+ getWorldMatrix()
WorldObject
+ render()
MenuScene
note the draw vs render semantic, render does more than just drawing
MainMenuScene
Obstacle
Vehicle
TriggerBox
- callback
Tunnel


pkg 0_diagrams
physics
Simulation
+ addObject() : void
+ addEventHandler() : void
+ getObjectById() : PhysicsBody
+ step(delta : float) : void
PhysicsEventHandler
+ onCollision() : void
+ shouldObjectsCollide() : boolean
PhysicsBody
- id : physicsId
0..*
scenes
WorldObject
0..*
GameScene
GameEventsLogic