Développement de jeu vidéo

Voici le moment tant attendu du TP final ! Vous avez déjà pu découvrir la programmation de jeu vidéo au cours du TP de fin de semestre 1, il est temps de voir ce qu'il est possible de faire lorsque le basecode est bien plus imposant. Vous allez découvrir un nouveau basecode qui reflète la quantité de travail nécessaire au prototypage d'un jeu complet. Il ne vous reste dès lors que la logique de jeu à implémenter : quels ennemis ? Quels comportements pour ces derniers ? Quels bonus ? Un boss ? La base est présente, faites en sorte que notre correction soit un bon (et long) moment de jeu !

1 Introduction

Un jeu vidéo complet peut prendre plusieurs semaines voir mois à préparer. La plupart du temps, il est inutile de tout recoder et le développeur se base sur un ensemble de fonctionnalités lui simplifiant la vie. Ces fonctionnalités sont regroupées dans les bibliothèques (*library* pour la version anglaise). Il est par exemple inutile de savoir comment allumer un pixel sur un écran puisqu'il existe une ou plusieurs bibliothèques permettant d'afficher une image à un endroit précis sur l'écran. De cette façon, le développeur se concentre plus sur la logique de jeu que la technique.

Il en est de même pour ce TP : plusieurs bibliothèques sont fournies pour vous « simplifier » la vie, en particulier au niveau de l'affichage des éléments et du moteur physique. Vous aurez également à disposition une bibliothèque contenant les primitives du jeu *Super Potoo World* : de quoi charger la carte du monde, des ennemis, des infos de jeu à afficher etc. Ces bibliothèques sont contenues dans des fichiers `.lib` : il s'agit d'un ensemble de fichiers objets, qui sont donc compilés et qui sont ajoutés au binaire final lors de l'édition de liens.

1.1 La programmation orientée objet

La grosse surprise vient du langage de programmation : nous utilisons ici le `C++`. L'intérêt de ce langage vient de son paradigme de programmation orienté objet. Cela signifie, pour simplifier, que tout élément du langage est un objet et que chaque objet possède des propriétés et fonctions propres. Cette façon de penser le code permet d'uniformiser son écriture. Pour un projet complexe, chaque objet est défini dans un fichier qui lui est propre et le développement de nouveaux objets est calqué sur la définition des objets existants.

Parce qu'un peu de vocabulaire ne fait pas de mal, voici quelques notions que vous retrouverez régulièrement dans ce sujet :

- une *classe* est, grossièrement, une structure dans laquelle on peut ajouter des fonctions ;
- une *méthode* désigne une fonction liée à une classe ;
- un *objet* est une *instance de classe*, c'est à dire une variable correctement allouée et initialisée ;
- les *attributs* désignent les champs décrivant la structure interne d'un objet.

Une autre propriété fondamentale de la programmation objet est la notion d'*héritage*. Elle consiste à pouvoir définir une *classe fille / dérivée* à partir d'une *classe mère*. La classe fille *hérite* des propriétés de la classe mère et en profite pour en spécifier de nouvelles. L'avantage de l'héritage est donc, entre autres, d'obtenir tout un ensemble de propriétés en très peu d'instructions. Nous découvrirons tout au long de ce TP d'autres concepts liés à la programmation objet et les définirons en temps voulu.

1.2 Organisation et compilation du projet

Vous disposez de deux bibliothèques permettant de gérer l'affichage (`libRE`) et le moteur physique (`libPE`). Le code source vous est caché mais vous avez à disposition deux annexes listant les différentes fonctions de ces bibliothèques que vous serez amenés à utiliser durant ce TP.

Les sources que vous avez à disposition pour ce projet sont organisées en dossiers virtuels au sein de la solution Visual Studio. On y retrouve le point d'entrée du programme ainsi qu'un ensemble de dossiers :

- le dossier `Assets/` permet de gérer les différentes ressources propres à SPW (images, son, fichiers de niveau ...) ;
- le dossier `GameObject/` contient l'ensemble des classes liées aux objets du jeu ;
- le dossier `Input/` permet de gérer les entrées utilisateur (clavier et souris) ;
- le dossier `Scene/` permet d'initialiser le menu ou un niveau de jeu ;
- le dossier `Utils/` contient un ensemble de primitives utilisables partout.

Le dossier `GameObject/` est celui dans lequel vous allez passer le plus de temps pour développer la logique de votre de jeu. Afin de découvrir le projet plus en détail et de mieux appréhender la programmation objet, étudiez la section suivante déroulant un tutoriel de prise en main du code.

2 Tutoriel de découverte du projet

Le but de ce tutoriel est de vous faire découvrir quelques objets du jeu et d'implémenter vous-mêmes quelques primitives. Comme dit précédemment, la programmation objet permet de définir un ensemble d'objets : ce projet n'y échappe pas. Afin de réaliser au mieux ce tutoriel, ne sautez aucune étape. Lisez attentivement les informations qui vous sont données et modifiez le basecode uniquement lorsque demandé.

TODO (lecture seule de code)

Il est temps de plonger dans le basecode : ouvrez le dossier `SPW/` et au fur et à mesure de votre lecture, identifiez dans le basecode les lignes de codes correspondant aux explications données ci-après. **Ne modifiez pas** le basecode pour le moment !

2.1 Tout élément du jeu est un objet

Cette phrase signifie indirectement que tous les éléments du jeu sont issus d'un seul et même objet. Cet objet parent de tous les autres est défini dans la paire de fichiers `GameObject/GameObject.cpp/.h`. Comprenez donc que tous les autres objets sont des

`GameObject` dans lequel des informations et méthodes ont été modifiées et/ou ajoutées. Commençons l'analyse de cet objet et la découverte de la syntaxe `C++` avec le fichier `.h`.

La définition de l'objet débute avec la définition de sa classe ligne 30. Pour faire le lien avec la syntaxe `C`, il s'agit de définir ici les champs d'une structure qui contient les valeurs (attributs) et fonctions (méthodes) propres à l'objet. Une première différence avec le `C` consiste à classer ces éléments selon leur visibilité :

- `public` (ligne 32) : l'équivalent d'une définition classique dans un `.h`, tout élément de code faisant appel à cette partie de la classe a accès à ces champs ;
- `protected` (ligne 67) : les champs définis ici sont invisibles aux autres classes, sauf les classes héritant de celle-ci ;
- `private` (ligne 71) : tout ce qui est défini sous ce mot-clé est interne à la classe.

Les membres `public` contiennent en premier (lignes 33 à 36) la déclaration du *constructeur* et *destructeur* de classe. Pour tenter de faire un parallèle avec le `C`, il s'agit de fonctions qui permettent d'initialiser un objet et d'allouer sa mémoire interne ou de libérer la mémoire interne allouée. Nous verrons d'autres exemples plus abordables lors de l'étude des prochains objets.

La classe définit ensuite un ensemble de *méthodes virtuelles* sur les lignes 38 à 45. Ces fonctions sont les primitives propres à un objet de type `GameObject`. L'attribut `virtual` indique qu'elles ont pour vocation à être redéfinies lors de la création de nouveaux objets héritant de la classe. En effet, tous les objets du jeu ont en commun la particularité de :

- `DrawGizmos()` posséder des indications visuelles des propriétés physiques de l'objet et les dessiner ;
- `FixedUpdate()` pouvoir être mis à jour au niveau physique ;
- `OnDisable()` gérer le comportement de l'objet lors d'une désactivation temporaire ;
- `OnEnable()` gérer le comportement de l'objet lors de son activation ;
- `OnRespawn()` gérer le comportement de l'objet lors de la réapparition (*respawn*) du joueur ;
- `Render()` pouvoir être affichés ;
- `Start()` gérer l'initialisation de leurs paramètres lors de la création de l'objet ;
- `Update()` modifier leurs attributs en fonctions des entrées utilisateur.

Le reste de la définition de classe comporte des attributs et méthodes propres à l'objet pour le modifier.

Comme en `C`, les déclarations des méthodes vues précédemment s'accompagnent de leur définition dans le fichier `.cpp`. Vous pouvez notamment remarquer la définition du constructeur ligne 6, celle du destructeur ligne 64 et celles des méthodes virtuelles lignes 69 à 99. De base, les méthodes virtuelles ne font rien : c'est normal dans notre contexte. Il est difficile d'attribuer un comportement par défaut particulier pour les objets du jeu tant ils sont variés. Un joueur a beaucoup plus d'éléments à initialiser, modifier ou afficher qu'un bloc de sol, et quasiment aucun en commun. En revanche, chaque objet aura besoin de définir ses propres méthodes virtuelles.

Dans le cadre de ce TP, il n'y aura jamais besoin de modifier le comportement de base d'un `GameObject` : vous n'aurez donc pas à modifier ce fichier. En revanche, sa constitution suit un format classique :

- déclaration des membres et attributs de la classe dans le `.h` ;

- attribution du mot clé `public` ou `private` aux membres de la classe ;
- définition des méthodes dans le `.cpp`.

Voyons maintenant la définition de nouveaux objets via l'héritage de la classe `GameObject` pour créer un `Player`, un `Enemy` ou une `Firefly`.

2.2 Notions d'héritage

Voyons la théorie de l'héritage via l'étude d'un objet particulier : le `Player`. Comme précédemment, l'analyse de l'objet `Player` commence par l'étude de son fichier `.h`. On retrouve à partir de la ligne 8 la déclaration de la classe avec les constructeurs et destructeurs sur les lignes 11 et 12. Notons toutefois quelques subtilités par rapport à ce que nous avons vu précédemment. Observons la ligne 8 : le nom de la classe est suivi de l'expression `public GameBody`. Il s'agit de la façon de faire un héritage, la classe `Player` est ainsi une classe fille de la classe `GameBody`. Un `CTRL + clic` sur le nom de la classe nous amène à la définition de la classe `GameBody`.

Un *corps* désigne un objet possédant des propriétés physiques : le joueur, les ennemis, les blocs, les bonus etc. Les seuls éléments ne possédant pas de corps sont les éléments d'information comme le menu et le background. Les lignes 14 à 23 du fichier `GameBody.h` nous montre les méthodes virtuelles qu'il sera possible de redéfinir lors de la création d'une classe fille. Nous retrouvons la redéfinition des méthodes virtuelles `OnDisable()` et `OnEnable` d'un `GameObject`, les méthodes de gestion de collision `OnCollisionXXX()` et les méthodes d'affichage d'informations liée au corps `DrawXXX()`.

Revenons donc à notre `Player` : il s'agit d'un `GameObject` dont on a ajouté des propriétés pour en faire un `GameBody` avant d'en spécifier de nouvelles lors de la définition de la classe `Player`. Aussi, même si ces fonctions n'existent pas dans le fichier `Player.h`, la méthode `SetLayer()` de la classe `GameObject` ou la méthode `SetStartPosition()` d'un `GameBody` sont disponibles pour le `Player`.

Les lignes 14 à 22 du fichier `Player.h` indiquent les méthodes virtuelles qui sont redéfinies pour les adapter au `Player`. Le suffixe `override` est une protection supplémentaire qui lève une alerte au niveau du compilateur si l'on tente de redéfinir une méthode qui ne serait pas virtuelle. Les lignes 24 à 31 permettent de déclarer les méthodes propres à la classe `Player` qui seront accessibles à l'extérieur. Par exemple, le contact avec un ennemi appellera la méthode `Damage()` du `Player`. Enfin les lignes 36 à 52 définissent les attributs de l'objet : il s'agit de variables internes qui n'ont aucune raison d'être modifiées en dehors de la classe. Remarquons enfin que le champ `m_scene` défini en tant que membre `protected` d'un `GameObject` est disponible au niveau du `Player`, même si la classe du `Player` n'indique rien de cela.

2.3 Modification du Player

Il est temps de se plonger dans le fichier source afin de modifier le comportement du Player. Ouvrez donc le fichier `GameObject/GameBody/Player.cpp` et étudions le constructeur (à partir de la ligne 8).

2.3.a Constructeur et création d'animations

Nous observons sur les premières lignes l'initialisation des membres privés sous la forme `m_nomMembre(valeur)`. Il s'agit là du but premier d'un constructeur : l'initialisation des membres de la classe.

Nous y ajoutons à partir de la ligne 17 la gestion de l'animation. Afin de donner l'illusion d'un battement d'ailes ou de la course du joueur, un ensemble d'images (*atlas*) a été dessiné dans le fichier `Assets/Atlas/Player.png`. Le fichier `Player.json` qui accompagne l'atlas permet de définir plus plusieurs animations au sein de cet atlas. Sa structure est relativement simple et nous indique quelles images servent à quelle animation par le biais de noms.

Le chargement des textures et animations d'un atlas est géré par un `AssetManager`. Son initialisation est faite entre les lignes 17 à 19. Vient ensuite le moment d'affecter une animation au `Player`.

Création d'une animation

```
22    part = atlas->GetPart("Idle");
23    AssertNew(part);
24    RE_TexAnim *idleAnim = new RE_TexAnim(
25       m_animator, "Idle", part
26    );
27    idleAnim->SetCycleCount(0);
```

Il faut pour cela :

- récupérer une partie de l'atlas (lignes 22, 23) ;
- créer une nouvelle animation et l'affecter à l'animateur du joueur (lignes 24 à 26) ;
- adapter les paramètres de l'animation, ici, ne pas la faire tourner (ligne 27).

Remarquons que le fait de relier l'animation au `Player` est effectuée en passant le membre privé `m_animator` de la classe `Player` à la méthode `RE_TexAnim`. Le champ `m_animator` appartenant à la classe, il est inutile en `C++` de préciser à l'aide d'une flèche ou d'un point qu'il est issu du `Player`. Remarquez d'ailleurs qu'il n'existe pas dans le constructeur de variable de type `Player` !

Todo 1 (Ajout d'une animation)

En vous inspirant du code présent entre les lignes 22 et 27, ajoutez une animation de type chute libre. Pour cela, dupliquez les lignes 22 à 27 et modifiez-les pour :

- récupérer la partie `"Falling"` du même atlas ;
- créer une nouvelle animation `fallingAnim` à l'aide de la fonction `RE_TexAnim` en 'l'affectant au même animateur que précédemment ;
- paramétrez son affichage en boucle avec la valeur -1.

Terminez en modifiant le temps d'affichage de l'animation en ajoutant l'instruction `fallingAnim->SetCycleTime(0.2f);`.

2.3.b Méthode virtuelle Start() et création du corps

La modification du constructeur vous a permis d'initialiser non seulement l'ensemble des membres de la classe mais aussi les animations du `Player`. Lorsque le constructeur est appelé, un espace mémoire est réservé pour le `Player` et ce dernier est ajouté à la scène. En revanche, l'initialisation de la partie physique du joueur se fait dans la méthode `Start()`.

Remarque 1. Il est probable que les lignes de code affichées dans les bloc de code ci-dessous ne correspondent pas à ceux que vous voyez dans le basecode. C'est normal, puisque vous avez ajouté des lignes précédemment avec l'animation `Falling`.

La méthode `Start()` débute avec le lancement de l'animation initiale du `Player`. Ce dernier étant par défaut immobile, l'animation `Idle` est privilégiée. S'ensuit la création du corps physique (`PE_BodyDef`) que nous étudierons plus en détails plus tard. Vient ensuite en fin de méthode la création du collider du joueur.

Définition d'un collider

```
57      // Création du collider
58      PE_ColliderDef colliderDef;
59
60      PE_CapsuleShape capsule(
61         PE_Vec2(1.7f, 1.35f), PE_Vec2(-1.0f, 1.85f), 1.35f);
62      colliderDef.friction = 1.0f;
63      colliderDef.filter.categoryBits = CATEGORY_PLAYER;
64      colliderDef.shape = &capsule;
65      PE_Collider *collider = body->CreateCollider(colliderDef);
```

Un corps est doté d'un ou plusieurs *collider*. Ces boites servent de détecteurs de collision pour le moteur physique. Quelques formes sont définies pour les boites, les plus utilisées étant la boite rectangulaire et la circulaire. On y ajoute un ensemble de *filtres* permettant d'identifier les objets qui peuvent entrer en collision avec cet objet. Le champ `categoryBits` renseigne le type du collider (nous l'utiliserons plus tard). Le champ `maskBits` renseigne le type des colliders qui pourront interagir avec celui-ci (absent du Player, mais disponible dans les ennemis ou les `Collectable`). Les constantes disponibles sont définies dans le fichier `GameObject.h`, lignes 7 à 10. L'ajout d'un type se fait par ajout de la valeur numérique associée au filtre. Les colliders pouvant interagir seront alors ceux qui ont le filtre de l'autre activé et inversement : un collider de catégorie A ayant un masque B activé pourra interagir avec un collider de catégorie B ayant un masque A activé.

Todo 2 (Taille de collider cohérente)

En exécutant votre programme et en appuyant sur la touche `P`, vous obtenez des informations de déboguage qui représentent les données du moteur physique. En particulier, observer le collider du joueur : rien ne va ! Jouez avec les paramètres de la définition de la capsule pour comprendre leur fonctionnement et proposez des dimensions raisonnables pour le collider du joueur. Inspirez-vous de la dimension normale du sprite dans les assets, nous modifierons à nouveau ce collider plus tard si besoin.

2.3.c Contrôles utilisateurs et le saut

La méthode virtuelle `Update()` fait le lien entre les actions clavier ou souris de l'utilisateur et l'état des éléments du jeu. Actuellement, seule la direction du joueur (gauche - droite) est implémentée via la récupération du champ `hAxis` du contrôleur. Cette information est récupéré dans un membre du `Player` afin de pouvoir mettre à jour son déplacement dans la méthode `FixedUpdate()`, cette dernière gérant l'ensemble des mises à jour physique d'un objet. Via la variable `controls`, il est donc possible d'accéder à l'état des touches clavier pour mettre à jour les paramètres du joueur.

Todo 3 (Récupération de l'information de saut)

En observant les membres privés du `Player`, nous remarquons qu'il existe un booléen `m_jump`. Mettez-le à jour en fonction de l'état du contrôleur concernant la touche saut (champ `jumpPressed`). Utilisez la complétion automatique pour savoir à quel champ du contrôleur il faut accéder.

Comme indiqué précédemment, la méthode `Update()` ne met pas à jour la physique du joueur. Cette mise à jour est attribuée à la méthode `FixedUpdate()` que nous étudions maintenant. Le principe de cette fonction est proche de ce que l'on peut faire en physique : l'environnement agit sur le `Player` et modifie sa vitesse. En général, seule la vitesse est mise à jour, le moteur physique s'occupe ensuite de calculer sa position en prenant en compte le temps qu'il s'est écoulé depuis la dernière mise à jour. Par exemple, le respawn du joueur est activé lorsqu'il tombe dans un trou.

Chute du joueur

```
113  if (position.y < -2.0f)
114  {
115    m_scene.Respawn();
116    return;
117  }
```

Les lignes suivantes permettent une détection fiable du sol à l'aide de lancers de rayons, nous ne détaillerons pas cette partie. Intéressons plutôt à la mise à jour de la vitesse du joueur. La première étape consiste à récupérer la vitesse actuelle du joueur, de la même façon que l'on a récupéré sa position.

Todo 4 (Récupération de la vitesse du joueur)

En début de fonction, juste après la récupération de la position, ajoutez la ligne de code suivante permettant de récupérer la vitesse du joueur.

```
PE_Vec2 velocity = body->GetLocalVelocity();
```

On privilégie la version `GetLocalVelocity()` plutôt que `GetVelocity()` pour des questions de compatibilité future avec des potentielles plateformes mobiles.

Il est maintenant temps de gérer le saut. Comme vu précédememnt, l'information de saut est récupéré le champ `m_jump`. Il suffit donc de vérifier la valeur de ce champ pour savoir si l'on a besoin d'appliquer un saut au `Player`. Pour cela, il suffit d'affecter une vitesse verticale suffisamment importante au `Player` pour le faire décoller.

Todo 5 (Application du saut)

Vers la fin de la méthode `FixedUpdate()`, ajoutez un test permettant de vérifier si un saut doit être appliqué au joueur. Si c'est le cas, appliquez une vitesse verticale au joueur et pensez à ré-initialiser la valeur `m_jump` afin de ne pas le faire s'envoler vers l'infini. Les vitesses sont données en tuiles par seconde, une tuile faisant 80 pixels de large. Cela doit vous aider à donner une valeur de saut cohérente.

Malgré toutes ces modifications, le joueur refuse de s'envoler. C'est tout à fait normal puisque vous avez modifié une variable locale : la vélocité du joueur a été récupéré en début de fonction et vous travaillez sur cette variable locale. Il est temps d'écraser l'ancienne vitesse avec la nouvelle valeur calculée.

Todo 6 (Mise à jour de la vitesse)

La dernière ligne de la méthode `FixedUpdate()` consiste à donner une nouvelle valeur à la vitesse du joueur. Pour cela, il faut appliquer la nouvelle de vitesse au corps de l'objet à l'aide de l'instruction suivante.

```
body->SetVelocity(velocity);
```

Comme vous avez pu le vérifier, sa vitesse est bien trop importante pour être jouable. Nous allons profiter de sa modification pour comprendre sa provenance. Les dernières lignes de la méthode `FixedUpdate()` appliquent une force au `Player` paramétrée selon son membre `m_hDirection`.

```
PE_Vec2 direction = PE_Vec2::right;
PE_Vec2 force = (200.0f * m_hDirection) * direction;
body->ApplyForce(force);
```

Cette force est en fait donnée au joueur pour augmenter sa vitesse de déplacement selon la direction voulue par l'utilisateur. L'application de cette force au `body` permet ensuite au moteur physique de mettre à jour la vitesse du `Player`. Diminuer la valeur de cette force va limiter l'accélération du joueur. Toutefois, si on laisse la touche appuyée, le joueur poursuit son accélération tant qu'il ne rencontre pas de mur. La logique voudrait qu'il atteigne une vitesse limite.

Todo 7 (Gestion de la vitesse)

Changez la valeur de la force appliquée au joueur pour l'accélération de la direction (initialement 200). Ajouter ensuite les lignes de code suivantes afin de limiter sa vitesse à 9 tuiles par seconde. La fonction `PE_Clamp` permet de limiter la valeur d'un flottant entre des bornes données.

```
float maxHSpeed = 9.0f;
velocity.x = PE_Clamp(velocity.x, -maxHSpeed, maxHSpeed);
```

2.3.d Sprites et animations

Précédemment, vous avez dû modifier le constructeur afin de créer une nouvelle animation pour le `Player`. Il est maintenant nécessaire de gérer les moments lors desquels cette animation est affichée et la façon dont les sprites s'affichent à l'écran. L'affichage à proprement parler est réalisé au travers de la méthode `Render()`. Un calcul est réalisé

pour adapter les dimensions d'affichage en fonction de la dimension du sprite et de la portion de monde dessinée (aux alentours du TODO). La position du sprite est relative à la position du corps : elle peut s'afficher au milieu de celui-ci, en bas etc. De plus, la taille du sprite et paramétrable est basée sur un nombre de tuiles.

Todo 8 (Taille correcte du player)

Modifiez les valeurs dans les lignes de code permettant de déterminer la bonne taille du `Player` sur l'écran. Pour calculer cette taille, vous aurez besoin des dimensions des sprites du player (80×110). Modifiez finalement l'ancrage du sprite par rapport au corps en trouvant la bonne valeur du paramètre `anchor` dans la méthode `RenderCopyExF` de l'animateur. Les lignes de code à modifier sont les suivantes dans la méthode `Render()` du `Player`.

```
rect.h = 1.f * scale;
rect.w = 2.f * scale;
...
m_animator.RenderCopyExF(
&rect, RE_Anchor::CENTER , 0.0f, Vec2(0.5f, 0.5f), flip);
```

Tout est maintenant parfaitement calibré pour afficher des animations proprement. L'ensemble des animations disponibles est visible dans le constructeur. Nous en avons deux à notre disposition : `"Idle"` qui représente l'ibijau au repos et `"Falling"` qui représente l'ibijau en vol. Il est nécessaire de choisir quelle animation jouer en fonction de l'état du joueur. S'il est sur sol, il faut lire l'animation `"Idle"`, en vol, c'est l'animation `"Falling"`. Le choix de l'animation en cours de lecture est effectuée dans la méthode `FixedUpdate()`, au début de la gestion de l'état du joueur. Actuellement, la ligne suivante vous indique que l'animation `"Idle"` est jouée.

Sélection de l'animation à jouer

```
m_animator.PlayAnimation("Idle");
```

Todo 9 (Animation du joueur)

Adaptez l'animation du `Player` en fonction de son membre `m_onGround` et des information qui vous sont données précédemment. Si vous obtenez directement le résultat escompté, bravo ! Passez à la section suivante. Si non, et si votre code compile normalement, c'est normal. Lisez la partie suivante.

Les modifications que vous avez apportées sont basées sur un contrôle de l'état du joueur : sur le sol ou dans les airs. Or, la fonction `PlayAnimation` lance l'animation à partir du début. La méthode `FixedUpdate()` étant appelée à chaque frame, vous ne voyez que le premier sprite de l'animation, jamais l'animation complète. Il faut donc réaliser un contrôle sur le *changement d'état*.

Todo 10 (Animation correcte du joueur)

La logique est la suivante : si le joueur touche le sol et que le joueur n'est encore pas dans l'état `IDLE`, c'est qu'un changement d'état vient d'avoir lieu. Il faut alors lancer l'animation `"Idle"` et changer l'état du joueur. De la même façon pour le vol, si le joueur st dans les airs mais que son état n'est pas à `FALLING`, il faut changer son état et adapter l'animation. L'état du joueur est accessible grâce au membre `m_state` et les états nécessaires pour le moment sont `State::IDLE` et `State::FALLING`.

Le joueur est maintenant (enfin) fonctionnel ! Il est maintenant temps de découvrir d'autres objets du jeu pour voir comment ils peuvent interagir avec le `Player`

2.4 Modification de la noisette

Comme pour le joueur, la noisette possède un comportement très basique qu'il est possible d'améliorer assez simplement. L'intérêt d'étudier en détail cet objet est de mieux comprendre le principe de *callback* survenant au moment d'une collision. Pour cela, ouvrez le fichier `GameObject/GameBody/Enemy/Nut.cpp` et son header et étudions son basecode en commençant, comme d'habitude, par le header.

Nous observons ligne 6 que la classe `Nut` hérite de la classe `Enemy`. Cette dernière est minimaliste puisqu'elle n'apporte qu'une méthode virtuelle supplémentaire : `Damage()`. Cette méthode permet d'attribuer des dégâts à l'`Enemy` et potentiellement de mettre à jour quelques éléments chez le `damager`, dans notre cas, le `Player`. De base, un `Enemy` se contente de désactiver la présence de son corps dans la scène, mais il faudra écraser cette méthode virtuelle pour chaque nouveau type d'ennemi.

Revenons sur le header de la `Nut`. Nous observons que six méthodes virtuelles et deux membres privés sont définis. Beaucoup d'éléments sont connus, nous les avons rencontrés avec le `Player`. Nous allons donc détailler en particulier les éléments nouveaux, à savoir la gestion des collisions.

2.4.a Filtres de collision

Intéressons-nous plus particulièrement à la méthode virtuelle `Start()`. Comme pour le `Player`, elle crée le corps physique et lance l'animation. Nous souhaitons cependant spécifier un peu son comportement vis-à-vis des autres objets du jeu. Entre autres, nous ne souhaitons pas qu'une noisette n'interagisse avec des bonus ou les lucioles. Comme indiqué précédemment, les masques de filtre (`colliderDef.filter.maskBits`) permettent de préciser qui a le droit d'interagir avec la `Nut`. Par défaut, les masques sont tous activés : l'objet peut entrer en collision avec tous les autres types d'objets. Puisqu'elle ne doit pas traverser le sol et rebondir sur les murs, le filtre `CATEGORY_TERRAIN` est actif. Elle doit également faire demi-tour contre une de ses collègues, et non passer au travers : le filtre `CATEGORY_ENEMY` est actif. Enfin, elle peut naturellement attaquer et être détruite par le joueur, d'où la présence du filtre `CATEGORY_PLAYER`.

Todo 11 (Filtres de collision)

Ajoutez l'instruction suivante après la ligne 47 pour spécifier les objets qui peuvent entrer en collision avec la noisette. `colliderDef.filter.maskBits = CATEGORY_TERRAIN | CATEGORY_PLAYER | CATEGORY_ENEMY;`

2.4.b Ennemis et dégâts

Le propre de l'ennemi est d'infliger des dégâts au joueur et réciproquement. En règle générale, chaque objet est muni d'un méthode virtuelle `OnCollisionXxx` qui est appelée selon le moment de la collision :

- `Xxx = Enter` : première frame de la collision ;
- `Xxx = Stay` : durant la collision (y compris sur la première frame) ;
- `Xxx = Exit` : quand la collision est terminée.

Par exemple le `Player` possède les méthodes `Enter` et `Stay` lui permettant respectivement de détecter le premier contact avec un objet autre que le sol et de gérer correctement le contact au sol. La gestion d'une collision suit toujours la même logique initiale :

- récupération du `manifold` qui contient les informations physique de collision ;
- récupération des informations en provenant de l'autre collider.

A partir de là, tout est possible : résoudre la collision pour séparer les deux objets, appeler des méthodes liées à chacun des objets pour leur infliger des dégâts, leur donner une vie etc.

Pour l'attribution de dégâts, en règle générale, c'est l'attaquant qui inflige les dégâts à l'autre objet. Dans le cas de la noisette, un test doit être réalisé afin de connaitre l'angle de collision et le type de l'autre objet. Si l'autre objet est le `Player` et que l'angle formé par la normale de contact entre les deux objets et le vecteur vers le haut est suffisamment faible, le `Player` tue l'ennemi (à observer dans la méthode `Player::OnCollisionEnter()`). Sinon, c'est la noisette qui appelle la fonction de dégâts du joueur.

```
if (otherCollider->CheckCategory(CATEGORY_PLAYER))
  {
    Player *player = dynamic_cast<Player *>(collision.gameBody);
    if (player == nullptr)
    {
      assert(false);
      return;
    }
    float angle = PE_SignedAngleDeg(manifold.normal, PE_Vec2::down);
    if (fabsf(angle) > PLAYER_DAMAGE_ANGLE)
    {
      player->Damage();
    }
    return;
}
```

Todo 12 (Collision Player - Nut)

Copiez le code ci-dessus et collez-le à la suite des instructions présentes dans la méthode `Nut::OnCollisionStay`. Notez l'usage du `dynamic_cast()` qui permet de récupérer un pointeur sur le `Player` afin d'appeler sa méthode `Damage` dans le cas où c'est la noisette qui lui inflige des dégâts.

Maintenant que les callbacks de collision ont été appelés correctement, et que la collision est résolue (ou non), il faut éviter de rentrer à nouveau en collision avec le même objet. Il alors possible d'indiquer au moteur physique d'ignorer la collision.

```
if (m_state == State::DYING)
{
  collision.SetEnabled(false);
  return;
}
```

Todo 13 (Éviter trop de collisions)

Copiez le code ci-dessus et collez-le juste avant la portion de code précédemment ajoutée. `Nut::OnCollisionStay`. En effet, ce test doit être fait en premier pour éviter de réaliser les mêmes actions que précédemment si la noisette a été tuée par le joueur.

La dernière partie consiste à compléter la méthode `Nut::Damage()` afin de coder le comportement de cette dernière lorsqu'elle subit des dégâts. A la façon d'un Mario, nous souhaitons que le joueur rebondisse dessus avant que la noisette ne disparaisse simplement du jeu. La fonction `Damage()` prend en paramètres un pointeur sur le corps de l'attaquant de la noisette. Il suffit donc de le caster en `Player` afin d'accéder à la méthode de rebond de ce dernier.

Todo 14 (Rebond du joueur)

Complétez la méthode `Nut::Damage()` afin d'appliquer un rebond au joueur. Pour cela :

- à l'aide d'un `dynamic_cast`, caster le pointeur `damager` en `Player` ;
- si l'action s'est bien passée, le pointeur de retour est valide, vérifiez-bien cela ;
- si c'est le cas, appelez la méthode `Bounce` du `Player` ;
- finalement, et dans tous les cas, désactivez le corps physique de la noisette.

2.4.c Attaque de la noisette

Les dernières lignes de la méthode `FixedUpdate()` implémentent une optimisation du moteur physique. Si a noisette est invisible à l'écran, le corps est endormi. Lorsque le joueur s'approche, le cors est réveillé et peut donc avoir des interactions avec les objets avoisinants. Afin de ne pas laisser la noisette statique, ce qui serait bien triste pour un ennemi de ce calibre, nous souhaitons la voir se jeter sur le joueur lorsque ce dernier s'approche.

L'idée consiste donc à surveiller un changement d'état de la noisette pour lui donner une impulsion et laisser le moteur physique faire sa mise à jour au fil du temps par la suite. Le changement d'état survient lorsque la noisette entre dans une zone d'activation par rapport au joueur (par exemple 5 tuiles) et qu'elle n'était pas déjà en mouvement (autrement dit qu'elle était statique).

Todo 15 (Attaque de la noisette)

Ajoutez à la fin de la méthode `FixedUpdate` des instruction permettant de faire sauter la noisette en direction du joueur. Il faut pour cela :

- vérifier si la noisette est dans la zone de détection et qu'elle n'est pas dans l'état `State::IDLE` ;
- si c'est le cas, changer son état et lui donner une vitesse initiale (vers la gauche suffira pour le moment).

2.5 Ajout de collectables

Maintenant que vous avez découvert deux objets différents, c'est à vous d'en créer un à partir de zéro. Nous souhaitons créer dans cette section des lucioles qui peuvent être collectées par le joueur. Une classe `Collectable` est déjà définie et il faut donc créer une classe fille `Firefly`. Pour cela, créez une paire de fichier `Firefly.cpp/.h` et ajoutez-là à votre solution Visual Studio dans le dossier virtuel `Collectable`.

Nous allons nous inspirer de la classe `Nut` pour développer notre luciole. Comme tout objet du jeu, la `Firefly` possède un constructeur et un destructeur, ainsi que les méthodes virtuelles `Start()`, `Render()` et `OnRespawn()`. Un coup d'œil sur le header des `Collectable` nous montre qu'il existe une méthode virtuelle `Collect` spécifique pour ce type d'objets. Étant donné qu'une luciole est statique et attend simplement d'être collectée, elle n'écrase pas la méthode virtuelle `FixedUpdate()`. Il est donc inutile de la redéfinir dans le `.cpp`.

Todo 16 (Architecture du code)

Après avoir créé les fichiers, complétez-les à l'aide des informations données ci-dessus. Les membres `public` doivent être au nombre de six : constructeur, destructeur et quatre méthodes virtuelles. Pensez que la classe `Firefly` doit hériter de la classe `Collectable` dans le `.h`

Énumérons maintenant les grandes différences avec la noisette ou le `Player` afin de compléter correctement le fichier `cpp`.

Constructeur et destructeur. La seule nouveauté réside dans le fait que le constructeur d'un objet de type `Collectable` prend un second argument qui définit sa position d'avant ou arrière plan. Vous disposez pour cela d'une constante `Layer::COLLECTABLE`. L'animation est composée de plusieurs sprites et tourne l'infini, rien de plus par rapport à la noisette. Le destructeur reste vide, comme d'habitude.

Méthode `Start()`. En général, les lucioles ne réapparaissent pas à la mort du joueur : pas besoin de respawn ici. La luciole restera statique sur le niveau, aussi, le type de corps sera `STATIC` à la différence des deux objets précédents. Enfin, comme la noisette est statique et que son comporte est légèrement différent par rapport à la noisette, inutile d'appeler la méthode `SetToRespawn()`.

Méthode `Render()`. L'animation est tout à fait classique et peut être calquée en totalité sur celle de la noisette. Toutefois, l'ancrage du sprite par rapport au corps peut être modifié.

Méthode `OnRespawn()`. Le `Respawn` de la luciole est très basique puisqu'elle est immobile. Il suffit de réactiver le corps physique et de réinitialiser son animation.

Méthode `Collect()`. Voici enfin la seule nouveauté de cette classe. Il s'agit ici de décrire le comportement de la luciole lorsque le `Player` collecte l'objet. L'idée est donc similaire à la méthode `Damage()` du la `Nut`. On caste le `collector` en `Player` et on vérifie si ça a fonctionné. Si c'est le cas, on désactive le corps physique et on appelle la méthode `AddFireFly()` du `Player`.

Todo 17 (Complétion de la classe)

Utilisez toutes les informations ci-dessus pour compléter les méthodes de la classe `Firefly`. Inspirez-vous fortement de ce que vous avez fait sur la noisette. Attention toutefois à bien lire les instructions précédentes pour adapter le code ! Un simple copier-coller ne donnera pas les bons résultats.

Avec toutes ces modifications, vous vous attendez à un résultat spectaculaire, et... Déception ! Vous n'avez finalement rien fait ! En réalité, il existe encore une dernière modification qui consiste à lire les informations de niveau pour prendre en compte les lucioles lors de la lecture du fichier de niveau.

Todo 18 (Parsing du niveau)

Ouvrez le fichier `Assets/LevelParser.cpp` et ajoutez les lignes suivantes après la ligne 194. Nous ne détaillons pas le contenu de ce fichier pour le moment vous devriez commencer à comprendre par vous même quelle est son utilité.

```
Firefly *firefly = new Firefly(scene);
firefly->SetStartPosition(position);
```

Fin du tutoriel

Le tutoriel est maintenant terminé et votre compréhension du basecode est suffisante pour réaliser les modifications de votre choix sur le jeu. Pour vous aider dans votre tâche, un fichier supplémentaire vous détaille un grand nombre de modifications qu'il est possible d'apporter au jeu. Choisissez les améliorations à apporter en fonction de leur niveau de difficulté d'implémentation.

3 Instructions de rendu et notation

Le dépôt de votre TP doit se faire sur Moodle, dans la zone prévue à cet effet, avant **vendredi 9 juin 2023 23h59**. Avant de déposer votre projet, supprimez le dossier caché `.vs` à la racine de votre projet. Renommez le dossier contenant le projet au format `NOM1_NOM2`, par ordre alphabétique. Compressez votre dossier au format zip et déposez-le sur Moodle. Vous déposerez également sur Moodle les fichiers demandés dans la zone de vérification de plagiat. Ce fichier doit être strictement identique à celui de votre projet.

En bonus, vous avez jusqu'au lundi 26 juin, 8h45 pour déposer une version finalisée de votre jeu sur Moodle. Vous pourrez alors gagner jusqu'à deux points supplémentaires sur votre TP.

Des évaluations orales de code auront lieue au cours de la semaine ou au tout début de la semaine suivante. Elle permettront de vérifier que vous avez compris l'ensemble du code que vous avez produit. En cas de réponse non satisfaisante ou de suspicion de triche, une évaluation orale plus longue pourra avoir lieu. Votre note sera finalement coefficientée selon le résultat de votre présentation : un facteur multiplicatif compris entre 0 et 1 sera attribué à la note de votre projet à l'issue de l'oral.