SHOOTER 2D SFML

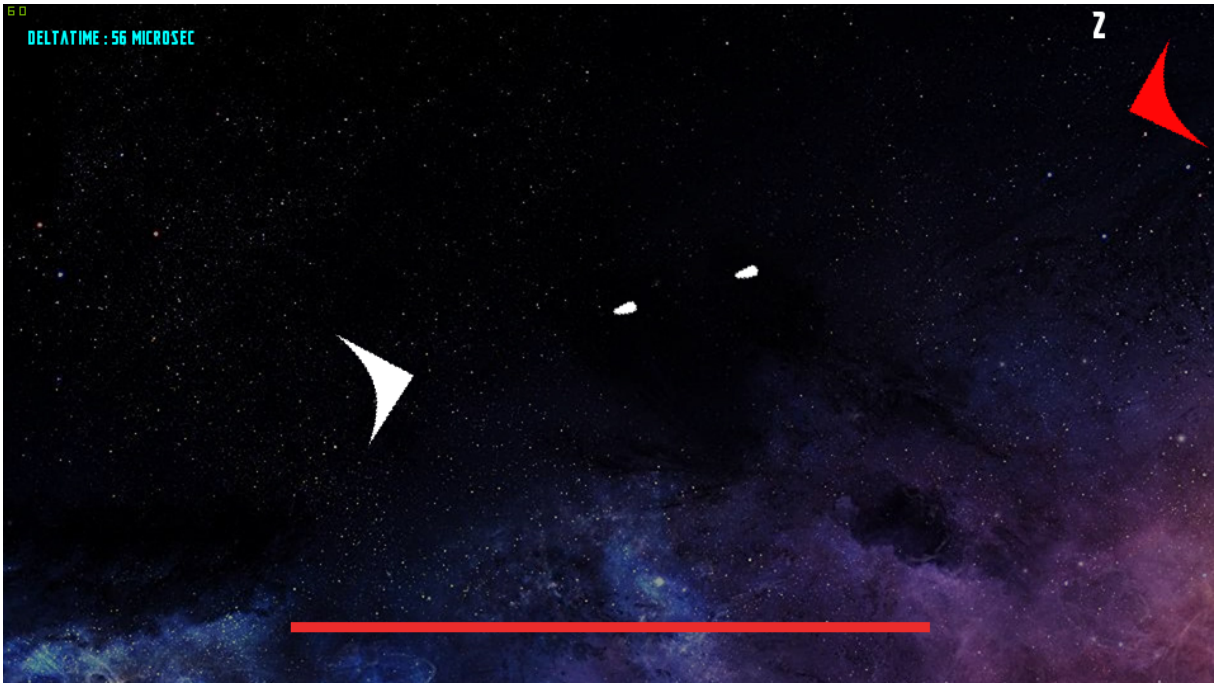
Projet de PROG YNOV M2

Jean-Baptiste Curvat, Gaultier Deshayes et Jason Coltier

1. Objectif du projet et présentation

L'objectif était de créer un jeu en c++ à partir de la librairie SFML. L'important était de concevoir une architecture robuste et suffisamment générique pour permettre des extensions futures.

Ce jeu est un space shooter en 2D où l'objectif est de survivre le plus longtemps possible face à des vagues d'ennemis et de faire le meilleur score.



2. Instructions d'installation et de lancement

Prérequis :

Installation de cmake :

https://cmake.org/download/

Utilisez le Windows x64 Installer .msi

Installation de vcpkg :

Récupérez le contenu du git suivant : https://github.com/microsoft/vcpkg.git

Lancez le fichier bootstrap-vcpkg.bat se trouvant à la racine.

Faire ensuite dans la commande windows à l'emplacement racine du dossier vcpkg :

```
vcpkg install sfml:x64-windows
```

Cela installera la librairie sfml dans le dossier vcpkg.

Il faut ensuite faire l'intégration de vcpkg aux différents environnements de développement grâce à la commande :

```
vcpkg integrate install
```

Lancement du projet sous visual studio :

Récupérez le projet sur git à l'adresse suivante : https://github.com/JasonColtier/Shooter2D_SFML.git

Créez un dossier "build" à la racine du projet.

Depuis ce dossier build, lancez la commande

```
cmake ..
```

Cela va créer un fichier .sln à partir des sources du projet que vous pouvez lancer.

Lancement du projet compilé :

Une version compilée du projet se trouve dans le dossier zippé _Export.7z du git.

Il vous suffit de l'extraire et de cliquer sur le fichier Shooter2DSFML.exe dans Release pour lancer le jeu.

3. La gaming loop

3.1. Update

La gaming loop est le cœur du jeu. Elle s'occupe de gérer la boucle **Update** et la boucle **Render**.

Elle est dans le fichier GameLoop.cpp

L'objectif ici est de faire des Update (modification des positions, gestion des collisions, gestion des inputs) plus fréquemment que des rendus. Le temps mis pour fair X update et 1 rendu est calculé et comparé au nombre d'images par secondes voulu (1/60) Le surplus est alors retiré de la boucle suivante pour qu'elle soit légèrement moins longue et que le nombre d'images par secondes soit globalement constant peu importe la durée des Update et des Render.

```
m_updateTime = 0;
//tant qu'on a pas atteint le nombre de fps voulu on continue les updates
while (m_updateTime < (1.0 / static_cast<double>(m_targetFPS)) * 1000000 - m_updateSurplus)
{
  m_deltaTime = Clock.restart().asMicroseconds(); //on utilise les microsec pour éviter de travailler avec des nombres minuscules et garder en précision
  m_updateTime += static_cast<int>(m_deltaTime);

  _Update();
}
//pour sortir du while il faut nécessairement dépasser le temps alloué pour atteindre nos FPS donc on reprend à partir de ce temps dépassé pour la prochaine update
```

```
_Render();
m_updateSurplus = m_updateTime - static_cast<int>(((1.0 /
static_cast<double>(m_targetFPS)) * 1000000));
```

Dans la fonction Update, la fonction Tick(int_64 deltatime) de chaque gameobject actif de notre jeu est appelée.

3.2. Render

Dans la fonction Render, la fonction RenderUpdate du RenderHandler de chaque gameobject qui en a un est appelée. Cette fonction fait un Draw de chaque sf::Drawable contenu dans ce RenderHandler.

Pour le rendu nous trions nos GameObject par un zIndex afin de savoir dans quel ordre les rendre et permettre à certain de s'afficher aux dessus d'autres.

4. Activation et désactivation des GameObjects

Notre jeu repose sur un relativement grand nombre de créations d'objets, notamment pour les balles et les ennemis.
Nous avons donc mis en place un système d'activation / de désactivation sur tous les GameObject afin de limiter les instanciations et les destructions.

Lors de spawn d'un nouveau GameObject, nous vérifions si un objet de même classe est déjà dans la liste des objets désactivés, si oui on le réactive avec de nouveaux paramètres. Sinon on le crée et on appelle sa fonction Activate().
Ce n'est pas dans le constructeur mais dans le Activate() que se fait l'initialisation de nos objets.

```
//Spawn d'un nouveau GameObject et passage des arguments à la
fonction Activate
template <class T = GameObject, typename ...Args>
```

```
T* SpawnActor(Args ...args)
{
   for (auto* object : m_lObjectsDeactivate)
   {
       auto* tmp = dynamic_cast<T*>(object);
       if (tmp)
       {
           tmp->Activate(args...);
           ActivateObject(*tmp, false);
           return tmp;
       }
   }
   T* newObject = new T();
   newObject->Activate(args...);
   return newObject;
}
```

Il faut noter que chaque fonction Activate() doit appeler le Activate() de sa classe parente, tout comme pour le Deactivate()

Un objet désactivé reste en mémoire mais n'est plus rendu ni updaté et ses components sont détruits.

5. La gestion des inputs via SignalSlot

Les inputs sont gérés dans la classe statique InputHandler grâce au système du **SignalSlot**. L'objectif était de centraliser cette gestion des inputs et de la détacher du reste du jeu tout en permettant à tous les gameObjects qui le veulent de récupérer des évents.

Le **InputManager** associe un InputsEnum (le type d'action effectuée : shoot, forward…) avec un booléen pour savoir si la touche est appuyée ou non.

Le SignalSlot est appelé pour chaque changement d'état d'intput et tous les objets qui écoutent ces changements en sont informés.

```
//Envoie un signal lors d'un changement d'état d'input
void InputManager::SendSignalIfNewInput(InputMapping& input, bool
pressed)
{
   if (pressed && !input.second)
   {
       input.second = true;
       m_inputSignal(input);
   }
   if (!pressed && input.second)
   {
       input.second = false;
       m_inputSignal(input);
   }
}
```

Les components comme le PlayerMovementComponent peuvent donc réagir aux signaux grâce à un binding dans le constructeur. Cette implémentation du signal slot se fait en donnant en paramètre un pointeur de fonction membre d'une classe.

```
//Binding d'une fonction à la réception d'un signal
InputManager::GetSignal().Connect<PlayerMovementComponent>(this,
&PlayerMovementComponent::OnInputChanged);
```

```
void PlayerMovementComponent::OnInputChanged(const InputMapping
input)
{
  //si on a appuyé ou relâché la touche pour bouger
  if (input.first == InputsEnum::Forward)
  {
     m_moveTowardMouse = input.second;
  }
}
```

6. La gestion des collisions

Le **CollisionManager** a pour rôle de trier les objets par abscisse afin de vérifier rapidement si une collision est possible.

Nous faisons un premier check de collision large grâce à un radius afin de voir si deux objets sont suffisamment proches pour entrer en collision.

Si c'est le cas, nous faisons un check plus précis en testant une intersection de segments entre nos objets.

Si un collision a lieu, le CollisionManager appelle la classe **DispatchOnCollision** en donnant en paramètre les deux objets qui entrent en collision.

Cette classe utilise une Typelist et cherche à appeler la bonne surcharge de la classe **TOnCollision** en fonction des classes des deux objets.

DispatchOnCollision permet grâce à un parcours d'une type liste et à une surcharge de template qui prend un argument variadic de créer Compile-Time les chemins pour accéder à la bonne surcharge de TOnCollision pour chaque combinaison d'objet qui peuvent collisionner.
Il y a plusieurs avantages à utiliser cette méthode, les principaux sont:

- Quand on veut rajouter un comportement pour un duo de classe GameObject, il n'y a pas besoin d'aller modifier un switch ou un enchaînement de if/else pour qu'il puisse trouver cette combinaison.Il suffit de rajouter les types des objets dans la type listes s'ils n'y sont pas, et de créer une surcharge de TOnCollision.
- Si on essaye de faire collisionner deux objets et que le type d'un des deux objets n'est pas dans la typelist, on le saura Compile-Time. Alors que si nous avions utilisé des switch ou if/else, il aurait fallu attendre que le cas arrive au hasard pendant le Run-Time et qu'il fasse crasher l'application pour savoir qu'il n'était pas géré.
-

```
template<template<typename, typename> class Functor, typename T1,
typename FirstT2Type, typename ...OtherT2Types>
struct DispatchOnT2<Functor, T1, TypeList<FirstT2Type,
OtherT2Types...> >
{
  static FnType CheckType(TypeId t2)
  {
     if (t2 == FirstT2Type::GetClassTypeId())
     {
        return [](GameObject& go1, GameObject& go2)
        {
           Functor<T1,
FirstT2Type>::Reaction(static_cast<T1&>(go1),
static_cast<FirstT2Type&>(go2));
        };
     }
     return DispatchOnT2<Functor, T1, TypeList<OtherT2Types...>
>::CheckType(t2);
  }
};
```

La classe TOnCollision regroupe toutes les spécialisations de collisions entre nos classes. Voici un exemple de spécialisation de collision entre un joueur et un bonus pour se soigner.

```
template <>
struct OnCollision<Player, BonusHeal>
{
   static void Reaction(Player& player, BonusHeal& bonusHeal)
   {
       auto life = player.GetComponentOfClass<LifeComponent>();

       life->ModifyHealth(bonusHeal.m_pdtVie);
       Print::PrintLog("take heal ! ");
       bonusHeal.Deactivate();
   }
};
```

Nous avons créé TOnCollision pour regrouper dans une seule fonction un comportement de collision entre deux objets. Au début nous étions partis sur un système qui appelait une fonction dans l'objet ReactionCollision(GameObject* otherObject) mais cela posait beaucoup de problèmes tels que:

- Il faut toujours faire des switch sur le type du otherObject pour savoir avec quoi on a eu une collision et donc quel comportement utiliser.
- Pour une collision on se retrouve avec deux appels de fonction différents, ce qui fait se poser la question de qu'est-ce qui réagit à quoi. Par exemple une collision Bullet/Enemy, on pourrait mettre dans la fonction ReactionCollision de Bullet le comportement qui réduit la vie de Enemy, mais parfois quelqu'un décidera peut-être de faire l'inverse ce qui rend le code dur à maintenir.
- Si dans le premier appel de fonction l'objet détruit l'objet avec lequel il a eu une collision ou s'il se détruit lui-même, on va avoir un problème pour appeler la fonction du deuxième objet.

7. Utilisation des Components

Nous utilisons une architecture permettant aux GameObjects de posséder des Components.
Ces Components ajoutent des fonctionnalités à nos GameObjects tel que le mouvement ou la possibilité de tirer.
Chaque Component reçoit un Tick d'Update qui est appelé après l'update du owner (le GameObject parent).

Nous avons privilégié cette architecture pour la flexibilité qu'elle offre en termes d'ajout de fonctionnalités et car elle s'adapte bien au domaine du jeu.

8. La gestion des médias

Les médias extérieurs comme les images ou les sons sont gérés grâce aux TextureManager et AudioManager.

Ces classes permettent de ne faire qu'une seule instanciation en mémoire de la ressource demandée et de renvoyer un pointeur vers cette ressource à la classe qui l'a demandé.

```
sf::Texture* TextureManager::GetTexturePtr(ETextures t)
{
  const auto Iterator(mapTextures.find(t));

  //si on a notre texture de chargé
  if (Iterator != mapTextures.end())
  {
     return Iterator->second;
  }

  //création d'une nouvelle texture
  sf::Texture* texture = new sf::Texture();
  texture->loadFromFile(_GetPath(t));//on charge l'image voulue
  mapTextures[t] = texture;//on conserve la donnée dans la map
  Print::PrintLog("create new texture ptr for ", _GetPath(t));

  return texture;
}
```

Cette fonction cherche si la texture demandée a déjà été instanciée et ajoutée dans notre map. Elle prend en paramètre un enum qui identifie la texture demandée.
Elle retourne un pointeur vers cette texture et l'instancie si besoin.

9. Gestion des ennemis

Les ennemis sont créés par **EnemySpawner**. Ce GameObject vérifie dans sa boucle d'update s'il y a encore des ennemis actifs ou s'il est possible de créer de nouveaux ennemis.

Lors de la création d'un ennemi, il choisit aléatoirement un IMovementComponent entre

- KamikazeMovementComponent
- RunAwayMovementComponent

Il associe à cet ennemi un ShootComponent parmi :

- ClassicPistol
- ShotGun
- Sniper

Les components nous permettent ainsi de créer des combinaisons différentes et d'apporter de la variété à nos ennemis.

10. Gestion de la difficulté

La classe **EnemySpawner,** en plus de gérer l'apparition des ennemis, gère également l'évolution de la difficulté en jeu. Elle possède un compteur qui lui permet de connaître le nombre d'ennemis éliminés par le joueur ainsi qu'une valeur requise pour passer à la difficulté supérieure.

Une fois que ce compteur atteint la valeur requise, l'**EnemySpawner** va dans un premier temps augmenter le nombre d'ennemis maximum à l'écran puis va augmenter le nombre de points de vie maximum des prochains ennemis qu'il va créer. Pour finir, le compteur d'élimination se réinitialise et la valeur requise pour passer à la difficulté supérieure augmentera.