Workshop projet de jeu vidéo n^{o} 1

Théo Sellem (TheneDiel)

Vendredi 09 Novembre 2012

Table des matières

I Introduction

1 Présentation du workshop

Nous arrivons enfin dans la partie plaisante de cette soirée ! Nous espérons que vous prendrez autant de plaisir que nous à réaliser ce jeu dont la renommée n'est plus à faire.

Allez-vous chercher un café ou une autre boison caféinée dans le distributeur le plus proche, ça va bientôt commencer.

Dans ce second workshop, nous vous proposons de vous guider pas-à-pas dans la réalisation d'un jeu, copie de VVVVVV dont vous pourrez voir plus loin des captures d'écrans.

2 Histoire et règles

Pour essayer de reproduire un jeu, il est important de comprendre quelles sont ses règles et dans quel contexte il est sorti. Cependant, si cette partie vous ennuie, que vous connaissez déjà tout ou que vous vous en foutez, rien ne vous empêche de la passer.

2.1 Histoire

VVVVVV est un jeu type plateforme 2D conçu par Terry Cavanagh. Le jeu a été développé avec Adobe Flash et publié le 11 Janvier 2010, pour Microsoft Windows et Mac OS X .
Le joueur contrôle le capitaine Viridian, qui dès le début de VVVVVV doit évacuer le vaisseau spatial avec son équipage, lorsque le navire est affecté par "une interférence dimensionnelle". L'équipage s'échappe par un téléporteur sur le navire, mais le capitaine Viridian se sépare du reste de l'équipage à l'autre bout du téléporteur. De retour à bord du navire, le capitaine apprend que celui-ci est pris au piège dans une autre dimension (appelé Dimension VVVVVV), et que l'équipage du navire a été éparpillé dans celle-ci. Le but du joueur est de secourir les membres d'équipage portés disparus et trouver la cause de l'interférence dimensionnelle.

2.2 Règles

Contrairement à la plupart des jeux plateforme, le joueur n'est pas capable de sauter, mais peut inverser le sens de la gravité en position debout sur une surface, provoquant le capitaine Viridian à tomber vers le haut ou vers le bas. Le joueur utilise cette mécanique pour parcourir les nombreux niveaux en évitant tout les obstacles qu'il rencontre.

Parlons un peu plus en détail du jeu, même si vous n'êtes pas obligé de réaliser une copie conforme de l'original. Il existe différents éléments indispensable au gameplay.

- les pointes/pics : les principaux obstacles du jeu qui provoquent la mort du capitaine lorsqu'il y a collision entre les deux.
- les barres mouvantes/rebondissantes : elles permettent respectivement de se deplacer sur la barre et de rebondir (en changeant la gravité)
- les checkpoints : sauvegarder la position du joueur (histoire de pas recommencer depuis le début...)

Il y a d'autres objets dans la suite des niveaux mais nous ne les aborderons pas dans ce TP.

II Architecture d'un jeu vidéo

Un jeu vidéo est un projet intéressant : vous aurez l'occasion de toucher à beaucoup de choses fondamentalement différentes, et opérant sur des périphériques tout aussi différents comme le réseau, le graphisme, l'interface utilisateur, la physique, l'intelligence artificielle, ... Vous aurez aussi à faire appel à vos talents extra-informatique afin de vous assurer que votre jeu aura une histoire palpitante, une durée de vie acceptable, et un gameplay implaquable.

Devant cette entreprise, vous êtes quatre. Vous êtes pleins d'imagination, mais dès que vous commencez aux points 'technique'... vient cette question : « Par où on commence ? »

1 Par où on commencer ? – la segmentation !

Comme dit précédemment, un jeu est composé de plusieurs petites briques distinctes. Vous l'aurez comprit : il faut segmenter son travail en autant de parties qu'il y a de briques. On aura typiquement une ou deux personnes par briques :

- La brique « UI » (Interface Utillisateur/User Interface),
- la brique « graphique »,
- la brique « intelligence artificielle »,
- la brique « réseau »,
- la brique « physique »,
- la brique « son ».

Seulement, parler de « briques » ne fait pas très professionnel, alors on prend un mot plus sexy : moteur (moteur physique, moteur graphique, etc.). Et pour faire encore plus pro, on cause en anglais (graphics engine, sound engine, physics engine, etc.) !

Si vous réfléchissez bien, celui qui travaille sur le graphisme, n'aura pas grand chose à échanger avec celui quit travaille avec la physique (sauf si vous faites du pixel-perfect, mais c'est une autre histoire) ! On peut donc dire que ces moteurs sont indépendants deux à deux. Et c'est justement cette indépendance qui vous permet de vous répartir les tâches efficacement. De plus, une telle indépendance rend l'utilisation de VCS (SVN, Mercurial, Git, etc.) particulièrement intéressante : une fois qu'un moteur

sera achevé, un push (commit sous SVN) permettra d'envoyer les modifications à tout le monde, en plus de permettre à plusieurs personnes de travailler efficacement sur un même moteur. Typiquement, le graphisme, l'IA, et parfois la physique sont des parties plus lourdes que les autres. Il est donc judicieux que vous formiez une répartition des tâches équilibrées du genre :

- une personne pour l'IA + le son,
- une personne pour le réseau + la physique,
- une personne pour l'UI et le graphisme,
- une personne pour ... euh tous les moteurs sont déjà prit !

Arf... la quatrième personne n'aurait rien à faire ? Eh bien si ! En réalité vous ne mettrez pas une personne pour deux moteurs, mais plutôt deux personnes sur deux moteurs. De plus, certains jeux ne nécessitent pas de réseau, ou encore de physique très évoluée.

2 VVVVVV

Bon, sérieusement, il faut être en ING1 ou être particulièrement doué pour coder un jeu complet, « from scratch », beau, avec une bonne durée de vie, toussa... en une seule nuit.

Alors, si vous avez tout suivi, vous commencerez par vous décider de qui va s'occuper de l'affichage, qui du son, qui de la physique et qui des entrées utilisateurs. Pour simplifier, il n'y a pas de réseau ou d'IA.

III Présentation de la programmation orientée objet

1 Le concept

L'idée directrice de la programmation orientée objet est de regrouper les données au sein d'objets tels qu'on les voit : une voiture, un poison, un animal, ...

Ces données sont les caractéristiques de ces objets ; par exemple, la voirutre a une couleur, un nombre de portes, une vitesse, ... ; un animal a une couleur, une taille, un poids.

Chacune de ces caractéristiques peut être modifiée par une ou plusieurs fonctions. Par exemple, la voiture a une fonction accélérer et ralentir qui modifie sa vitesse. Un animal peut quant-à lui avoir une fonction manger qui augmente sa taille et son poids.

Chaque objet possède donc des caractéristiques et des fonctions. Comme tous les objets que l'on rencontre tous les jours !

Info : les fonctions qui appellent les caractéristiques des objets sont quant-à elles appelées « méthodes ».

2 Accessibilité

Pour chaque élément (caractéristique, fonction ou classe), vous devez définir son niveau de visibilité. Les deux niveaux les plus courants sont `public` et `private`. Lorsqu'un élément est définit comme `public`, il est accessible et modifiable (sauf les méthodes) depuis n'importe quelle autre classe. Lorsque vous définissez un élément comme `private`, il accessible et modifiable uniquement depuis la classe qui les définie.

3 Les différent types de variables

Pour rappeler, une variable permet de stocker une donnée d'un type définie dans la mémoire vive de l'ordinateur, un entier par exemple. Ces données peuvent être simples : un entier, un caractère, un peu plus évoluées : un chaîne de caractères, une position dans le plan ou encore plus

complexe : un moteur graphique, physique ...

Ces trois types de variables servent à stocker et à accéder à ces valeurs.

3.1 Les attributs

Ce sont les « véritables variables ». La quasi-totalité du temps elles sont déclarées privé car on préfère les modifier depuis des méthodes ou des accesseurs. Elles se déclarent de cette façon, juste après le début d'une classe :

```
private int myVar;
```

3.2 Les propriétés

Ce sont des attributs « améliorés ». Il est possible de déclarer des propriétés publiques et de limiter la lecture ou l'écriture depuis les autres classes. Elles se déclarent de cette façon :

```
public int MyProperty
{
  get;
  private set;
}
```

Si on enlève le `private` devant le `set`, il est alors possible de modifier la variable depuis une autre classe.

3.3 Les accesseurs

À cheval entre les méthodes et les propriétés, les accesseurs se déclarent de la même manière que les propriétés mais ont un fonctionnement proche des méthodes. Par exemple, pour accéder à notre attribut `myVar`, déclarée plus haut, depuis une autre classe, on écrira :

```
public int MyAcc
{
  get
  {
    return this.myVar;
    }
}
```

Vous pouvez également définir la partie `set` lorsque vous souhaitez pouvoir définir l'attribut en question. Ce qui donne l'accesseur suivant :

```
public int myAcc
{
  get
  {
    return this.myVar;
    }
    set
    {
      this.myVar = value;
      }
}
```

Le mot clé `value` correspond à la valeur passé à l'accesseur lorsqu'il est appelé :

```
MyClasse instance = new myClass();
instance.MyVar = 42;
```

Dans cet exemple, `value` vaudra donc 42.

Il est tout à fait possible (et c'est même encouragé) de tester la valeur passé à l'accesseur. Vous pouvez en effet faire tous les tests que vous voulez mais aussi modifier d'autres attributs ou propriétés de manière à ce qu'elles restent toutes cohérentes.

Par exemple, remettre à 0 une variable lorsque vous en modifiez une autre :

```
public int myAcc
{
  get
  {
    return this.myVar;
    }
    set
    {
      if (value > 0)
      {
        this.myVar = value;
        this.incr = 0;
        }
        }
}
```

Ici, l'attribut `myVar` n'est modifié que si la valeur passé est supérieure à 0 et dans ce cas, l'attribut `incr` est définit à 0.

4 Instanciation de classes

Commençons par rappeler la déclaration d'une variable de base :

```
int a = 42;
```

On indique ici au compilateur que la variable `a` est de type entier (`int`) et vaut 42 à son initialisation. En effet chaque variable doit avoir un type défini. Les structure et les classes sont des types de données.

Prenons comme exemple la classe Point. Pour créer un nouveau point, on fait ainsi :

```
Vector2 position = new Vector2(2, 4);
```

Ici, le premier `Vector2` indique le type de la variable `position`. Le deuxième initialise la variable en appelant le constructeur de `Vector2` avec les deux arguments qu'il attend.

IV Présentation du squelette

Vous avez a votre disposition un squelette contenant une partie de l'architecture du projet, l'utiliser presente des avantages et des désavantages, mais nous vous laissons le choix de l'utiliser ou non

Les avantages :
- Ne pas se prendre la tête pour la structure
- Arriver plus vite à la partie "fun"
- Avoir plus de chances de finir le TP

Les désavantages
- Ne pas apprendre à démarrer un projet "from scratch" (depuis le début)
- Ne pas apprendre à structurer un projet du début à la fin

Concretement
- Si vous etes débutant et que vous comptez prendre le dernier metro, vous aurez plus l'impression d'avoir fait quelque chose de votre soirée si vous utilisez le squelette
- Si vous etes débutant et que vous comptez prendre le PREMIER metro ! Alors vous etes pas des tapettes et pouvez commencer from scratch !
- si vous etes des kikitoutdur du C#/XNA, que vous voulez pas vous faire chier a structurer du code et que vous preferez vous amuser à faire les bonus, take the skel !

Avant de rentrer dans la réalisation de votre VVVVVV, nous allons vous présenter la surcouche que vous avez à disposition, autrement dis, si vous ne comptez pas l'utiliser, passez au prochain chapitre.

1 Arborescence

- `VVVVVV/` : répertoire de base du projet.
- `VVVVVV/VVVVVV.sln/` : solution du projet (si vous cliquez VS2010 se lance et charge le projet)
- `VVVVVV/Vvvvvvvv/` : répertoire dans lequel se trouve le dossier VvvvvvContent contenant les images/sons etc... et le dossier Vvvvvv ou il y a vos classes et tout le reste...
- `VVVVVV/Vvvvvv/Vvvvvv/bin/x86/Debug` : répertoire contenant l'exécutable et les fichiers tierces qui pourraient-être chargés depuis votre programme (je ne parle pas des fichiers content, mais par exemple de la carte qui sera un format .txt (ou autre))

2 Le squelette

2.1 La classe Program et la classe Game

Lorsqu'on créer une projet XNA on à deux classes déjà existante, Program et Game Il faut juste retenir que Program appelle Game, donc on ne touche pas à Program et on code dans Game et les autres classes.

2.2 La classe Divers

C'est un peu la classe dechet ou on définit des variables qui sont nécessaire à l'initialisation du jeu, et qu'on appelle souvent. Ici on fixe la largeur et la hauteur de la fenetre affichée, ainsi que la position de départ du héros (qui sera par la suite modifiée par les checkpoints).

2.3 La classe Sprite et l'héritage

Créer une classe Sprites permet de définir les différentes caractéristques que peuvent avoir nos Sprites (comme le héros, les pics etc..). Son but est d'éviter de redéfinir N fois des caractéristiques communes à N types de sprites en les faisant hériter de cette classe Sprites.
Par exemple notre héros et un checkpoint vont tout deux avoir comme caractéristiques :

- une largeur
- une hauteur
- une position
- une texture
- un rectangle définit autour de la texture pour gérer ensuite les collisions (que nous verrons après).
- un sourceRectangle qui est en fait un rectangle qui définit seulement une partie de la texture à afficher.

On peut ensuiter surcharger notre classe, c'est à dire créer des propriétées qui ne seront pas forcement utilisées par toute les classes héritées de Sprites mais seulement certaines. Par exemple si on veut que les barres mouvantes et les barres rebondissantes puissent être dessinées en leurs appliquant une rotation alors que les pics, les checkpoints et le héros non, on créera deux méthodes "Draw" différentes dans la classe Sprites, l'une avec rotation et l'autre sans. Ainsi on fera attention à appeler la bonne méthode dans chacunes des classes héritées de Sprites.

2.4 La classe héros héritée de Sprite

Elle contient toutes les propriétées de classe Sprite (logique puisqu'elle en hérite) ainsi que tout les attributs et méthode propre au héros, bien évidemment, ce sera à vous de les fixés/codées, le squelette n'est la que pour la structure du jeu :

attributs

- Une vitesse
- Une position de départ (et son accesseur)
- Une gravitée pour l'axe X et une autre pour l'axe Y
- Un index/maxindex et une vitesse d'animation pour gérer l'animation du héros
- Un booléen movingUpDown qui permettra de savoir si le héros peut monter ou descendre
- Deux booléens inCollisionUpDown et inCollisionRightLeft qui permettrons de savoir si le héros est en collision avec un autre objet

Méthodes

- *UpdatePosRect()* : mettra à jour la position du rectangle définit autour du héros (utile pour gérer les collisions par rectangles)
- *KeyboardChangeGravity()* : changera la gravitée lorsqu'on appuiera sur la touche V
- *AnimationHeros()* : changera la position du sourceRectangle correspondant à la texture representant les différents états du héros en fonction de sa gravitée.
- *KeyboardMoveHeros()* : fera bouger le héros à gauche ou à droite.
- *UpdatePhysicEngine(...)* : on y mettra toute les fonctions relative au Moteur Physique
- *Update(...)* : C'est la fonction qui sera appelé dans la méthode Update appartenant à la classe Game, on y mettra les fonctions que l'on voudra mettre à jour en temps réel.
- *Draw_Hero* : Il me semble que ca sert a dessiner le héros mais je suis pas sûr.... comme la méthode update elle sera appelé dans la méthode Draw de la classe Game

2.5 La classe PhysicsEngine

La classe PhysicsEngine est une classe statique, ce qui veut dire que les fonctions (que l'on définira statique dans cette classe) appartiennent au type lui-même plutot qu'à un objet spécifique, lorsque l'on veut utiliser les fonctions d'un objet, on doit avant tout l'instancier (l'initialiser) : MonObjet objet = new MonObjet() et ensuite appeler sa fonction en faisant monObjet.MaFonction().
Avec une classe statique, si je veux appeler une fonction de ma classe, il me suffit de faire MaClasse.Mafonction().
Dans cette classe il y a pour l'instant deux squelettes de fonctions qui renverront true or false selon si il y a collision vers le Haut/Bas ou vers la Gauche/droite :

- *IsHerosCollidingWall_UpDown(...)*
- *IsHerosCollidingWall_RightLeft(...)*

Ce sera bien évidemment à vous de créer les autres fonctions pour les collisions héros/pics - héros/checkpoints etc...

2.6 La classe Map

La classe Map permettra de définir toutes les propriétées que notre carte aura.
Elle utilise la classe StreamReader (déjà implémenter dans C#) qui permet de lire le contenu d'un fichier.
En effet, la carte sera stockée dans un fichier.txt (on peu mettre ce que l'on veut à la place du ".txt" c'est just pour donner un exemple) qui contiendra un tableau de caractère ou chaque caractère sera associer à une texture.
attributs

- file de type StreamReader qui representera le fichier contenant la map
- map représentant un tableau bidimensionnelle de caractère où on y stockera la carte, car il est plus astucieux de lire le fichier une fois et de stocker la représentation de la carte dans un tableau, plutot que de lire le fichier à chaque fois que l'on veut faire des opérations dessus.
- width/height respectivement la largeur et la hauteur de la carte
- textures représentant un tableau de texture et size un entier representant la taille de ce tableau
- listWall representant une liste de Rectangle, on y stockera tout les murs (associés à des rectangles) de la carte pour gerer la gestion des collisions héros/mur.

Méthodes

- *GetWalls()* : Procédure qui recupère la liste des murs depuis le tableau map.
- *OpenFile()* : c'est la "grosse" fonction de la classe map, elle permettra de lire le fichier contenant la carte et de stocker sa representation dans le tableau map
- *LoadTextures(...)* : On y chargera les différentes textures qui seront stockées dans le tableau textures
- *DrawMap* : Elle dessinera la carte en parcourant le tableau bidimensionnelle map

V Au boulot !

Nous allons maintenant entrer dans le vif du sujet : la réflexion autour de la structure du projet et son implantation.

Note : ne consultez cette partie que si vous ne savez plus quoi faire. Avant de la consulter, appelez un assistant à la rescousse, il pourra vous donner des astuces.
Chacune des sections suivantes vous donneront des astuces des plus générales aux plus fines. Lisez les en fonction de votre besoin.

NB : Les solutions proposées ici ne sont que des propositions. Il y a bien évidemment des solutions (dont certaines pourraient être meilleures), n'oubliez pas que vous êtes libres !

1 Création de la classe Heros

Nous allons créer la première classe qu'on appelera Heros.cs, mais avant tout, qu'elle est l'interet de créer une classe héros ?
En faite si on voulais faire du code de porc, on pourrait coder presque un jeu entier dans la classe Game.cs fournie de base mais lorsqu'il il vous faudra debuger votre programme ou relire votre code, vous serez vite essouflé !

Le but d'utiliser une classe héros est de créer un objet héros disposant de toutes les propriétés dont il aura besoin, cette classe permettra d'autant plus de structurer votre code en y mettant uniquement les interactions de votre héros. Une classe representant un objet à une structure spécifique mais lorsque vous créer une classe c'est presque vide

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace MonProjet
{
    class Heros
    {
    }
}
```

Par exemple vous pouvez structurer votre classe héros de cette manière :

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace MonProjet
{
    class Heros
    {
  //les attributs du heros

   int vitesse;
  Texture2D texture; //Si la declaration Texture2D souligne une
     erreur il faut ajouter la ligne suivante en haut de la
     classe :
  // using Microsoft.Xna.Framework.Graphics; (il est plus facile
      de faire un clic droit sur le texture2D et cliquer sur
     resoudre.

  //Son constructeur (c'est ce qui est appele lorsque vous
     instanciez un objet en faisant
  //MaClasseObjet MonObjet = new MaClasseObjet(parametre(s)))
  //Le constructeur se declare comme ceci
  public NomDeMaClasse(parametre(s))
  {
    //ce que je veux qui soit execute lorsque j'instancie mon
       objet
    vitesse = 4;
  }

  // tout le code qui suit representera les fonctions privee ou
     publique (il suffit de mettre private ou public devant la
     fonction)

  private void MoveHeros(...)
  {
  //Une fonction qui pourrait gerer le deplacement du heros...
  //Cette fonction est privee car elle ne sera jamais appelee en
      dehors de la classe Heros, nous l'utiliserons dans la
     fonction Update(...) qui elle sera appelee dans la classe
     Game
  }
```

```
  public void LoadContent(ContentManager content)
  {
  //La fonction qui associera l'image heros.png au heros (
     utilisez la classe Texture2D)
  }

  public void Update(...)
  {
  //La fonction qui mettra a jour les interactions avec votre
     heros.
  }

  public void DrawHeros(SpriteBatch spriteBatch)
  {
  //La fonction qui dessinera votre heros.
  }
    }
}
```

Pour commencer vous allez devoir définir ses attributs : comme sa vitesse, une texture, et autres que vous devez deviner :)

Pour dessiner votre héros il vous faut charger la texture dans la fonction LoadContent puis la dessiner dans la fonction DrawHeros.

Une fois votre héros sur l'écran, toutes ses intéractions comme son deplacement ses collisions etc... se dérouleront dans la fonction Update.

Evidemment, il vous faudra créer votre objet héros dans la classe Game et appeler les fonctions qui permettent de charger et dessiner le héros là où il faut :)

2 La gestion du clavier

Pour effectuer une action lorsque l'on appuie sur une touche, on utilise la classe Keyboard.

```
private void MoveHeros(...)
  {
  if(Keyboard.GetState().IsKeyDown(Keys.Right))
    // j'augmente la position en X de mon heros
  }

public void Update(...)
  {
  MoveHeros(...)
  }
```

3 Le Tile-Mapping

Le Tile-Mapping consiste à représenter une carte (Map en anglais) sous forme de cases (Tile en anglais) et de la stocker dans un fichier hors du programme. Dans le dossier du squelette vous est fourni 2 fichiers carte.txt et carteForBonus.txt (ils se trouvent dans le dossier debug (revoir la partie arborescence)).
L'un represente une carte contenant la :

- Hauteur
- Largeur
- Position de départ du héros en X et en Y
- Le tableau de charactères associant chaque lettres à une texture ('m' pour mur et 'b' pour un carré noir)
- On detectera la fin de la lecture du fichier lorsqu'on lira une ligne contenant "end"

Pour lire dans un fichier nous pouvons utiliser la classe StreamReader qui dispose de la méthode ReadLine() et d'autres.
Nous allons donc créer une classe Map.cs qui permettra de gerer tout ce qui est en rapport avec la carte, comme son chargement, la fonction qui la dessine, une fonction qui récupère la liste des murs pour gérer les collisions etc...
Maintenant que vous avez appris à créer une classe, vous allez refaire ce que vous avez fait avec la classe Heros, mais en l'adaptant pour cette classe Map.

Il vous faudra créer un tableau à deux dimensions pour stocker la carte afin d'éviter de lire le fichier texte en continue.

```
char[,] _map; // declaration d'un tableau a deux dimensions.
 _map = new char[haueur, largeur]; // initialisation du meme
    tableau

for (int y = 0; y < _height; y++) // remplissage du tableau
  {
  //TODO
    for (int x = 0; x < _width; x++)
    {
      _map[x, y] = //TODO;
    }
  }
```

4 Le Moteur Physique

La partie physique de ce jeu n'est pas compliqué. En effet on ne gère que des collisions par rectangles. Par exemple si l'on veut gerer la collision entre le héros et un mur potentiel lorsque le héros se déplace vers la droite, on pourrait procéder de la manière suivante :

- On regarde si le rectangle définit autours du héros, en anticipant la position suivante de celui-ci, est en intersection avec l'un des rectangles appartenant à la liste de mur.

- Si il y a intersection, le héros n'avance pas, ou plus précisement, il pourra avancer si il n'y a pas intersection :)

```
for (int i = 0; i < longeur de la listMur ; i++)
{
  if (new Rectangle((int)heros.Position.X, (int)heros.
    Position.Y + heros.HerosSpeed, heros.Width, heros.Height
    ).Intersects(listMur[i]))
  return true;
}
return false;
```

Vous pouvez utiliser ce principe et l'adapter pour toute les directions !

5 Animation des sprites

Pour faire des animations en 2D, il existe différentes méthodes. La première et la plus naïve consiste à charger toutes les images composants une animation dans des textures séparées, puis on affiche successivement ces images afin de réaliser une animation.
Le stockage de ces images ainsi que leur changement sont très couteux pour la carte graphique qui est généralement optimisée pour travailler sur des images carrées et dont la taille est une puissance de deux. Or la plupart du temps les sprites animés ne respectent pas ces conditions.

Une deuxième méthode majoritairement utilisée consiste à stocker toutes les images composant l'animation sur une seule et même image.
Toutes les sous images sont décalées les unes par rapport aux autres avec une distance constante. Ainsi pour passer d'une image à une autre il suffit de décaler la zone de texture utilisée pour afficher le sprite.
Cette méthode résout les deux problèmes que nous rencontrions avec la première méthode.
Pour ce TP nous vous recommandons de choisir entre l'une ou l'autre des méthodes pour animer vos sprites, mais il existe bien sûr d'autres méthodes que vous êtes libre d'utilisées pour ce TP.

6 Animation de l'arrière-plan

L'arrière-plan est lui aussi animé dans ce jeu pour faire des arrière-plans défilant il y a deux principales techniques.

- La première consiste à utiliser deux spirites qui se déplacent et échange tour à tour leur position.
- La seconde consiste à décaler la fenêtre d'affichage de la texture d'arrière-plan. Ceci se fait au moyen de matrice de texture.

VI Bonus

Une carte spéciale pour les bonus vous est fournie dans le repertoire bin du squelette.
Elle contient :

- Une carte de 4 carrés adjacents
- Les coordonées des pics
- Les coordonées des teleporteurs Les coordonées des barres mouvantes et des barres rebondissantes

A vous d'adapter la carte en fonction de l'avancement dans les bonus.

1 Le Scrolling

Le principe du scrolling n'est pas compliqué, mais pas évident à implémenter.
Il faut définir un Rectangle représentant la camera qui va définir la zone que l'on affiche.
Il faut ensuite adapter la position des textures affichées en fonction de cette camera qui se déplacera elle même en fonction du héros.
Par exemple lorsque le héros atteint l'extremmité de la taille de la fenêtre...

2 les pics

Les pics reprénsentent les seuls objets qui pourront faire mourir le héros.
Pensez à utiliser la surcharge de la méthode Draw pour avoir la possibilité d'effectuer une rotation en fonction de l'orientation voulue du pics.

3 les checkpoints

Lorsque le héros passe sur un checkpoint, celui ci enregistre la nouvelle position de départ du héros, ainsi lorsqu'il rentre en collision avec un pic, il revient à sa position de départ.
Même chose que les pics pour la rotation.

4 Les barres mouvantes

Les barres mouvantes sont obligatoirement horizontales, si le héros rentre en collision avec, alors sa position évolue en fonction de la barre.

5 les barres rebondissantes

Les barres rebondissantes peuvent-être horizontales ou verticales. Si le héros rentre en collision avec, alors sa gravité s'inverse.

6 Un peu de son !

C'est vrai que trouver un rack avec du son, c'est comme trouver la copine de Theory, mais pour ceux qui ont des écouteurs, regardez du coté de la classe Song et MediaPlayer !

7 Coopérons

Pouvoir jouer à un jeu c'est bien, y jouer à plusieurs c'est mieux. Implémenter un mode réseau pourrait être fun non ?

8 Le postprocessing

Le postprocessing est un nom qui désigne l'ensemble des techniques de rendu qui sont faites après le rendu des spirites est des polygones en eux même. Ces techniques travaillent donc uniquement sur l'image qui va être dessinée à l'écran et non sur les différentes parties de la scène. Vous pouvez par exemple implémenter du flou de mouvement, de l'anti-aliasing ou encore des effets de déformation : soyez créatif ! La 2.5 D

Le jeu que nous réalisons ne se prête pas très bien à une version 3D mais à la 2.5D si. Les 2.5 D c'est le fait d'utiliser dans un univers pensé pour la 2D des éléments 3D (ou vis vers ça). ATTENTION : Ce bonus prend énormément de temps et doit avoir été pensé dès le début du projet. Ne vous lancez pas là-dedans sans être sûr de ce que vous faites.

VII Conclusion

Voilà, tout est dit, nous espérons que ce TP vous permettra de vous familiariser avec les bases de la gestion de projet. N'oubliez pas il n'y à pas UNE bonne réponse en particulier à ce TP toutes les approches sont valables. L'objectif principal outre le fait de coder un jeu est avant tout de créer une vraie cohésion et de bonnes habitudes au sein de votre groupe de projet. Have fun !