Candide 2.0 ou l'optimisme en jeu vidéo

Nina Ionescu 3mg01
Mentor : Jean-Marc Ledermann
Lycée Denis de Rougemont

Table des matières

1 Introduction

Il y avait en Westphalie... Et si cet incipit mythique se retrouvait un jour pixelisé, cela donnerait quoi ? C'est ce à quoi j'ai essayé de répondre lors de ce travail de maturité, qui consistait à adapter le premier chapitre dans *Candide* de Voltaire en un jeu vidéo de type RPG (jeu de rôle).

1.1 Lexique

Afin d'avoir une meilleure compréhension de ce document, voici un lexique.

- **assets** : Désigne l'ensemble des fichiers visuels et auditifs du jeu.
- **frame** : Image constituant une partie de l'animation d'un sprite.
- **hp** : Abréviation de health points, points de vie.
- **map** : Carte.
- **overworld** : Monde extérieur en français. Désigne l'ensemble de maps constituant le jeu.
- **pixel art** : Style graphique inspiré des jeux rétros où les assets sont dessinés à la main en respectant certains formats.
- **pnj** : Abréviation de personnage non-jouable.
- **sprite** : Image du jeu en partie transparente capable de déplacement.
- **spritesheet** : Image regroupant les frames d'animation d'un sprite.
- **tile** : Carreau en français ; petite image carrée à texture répétée.
- **tilemap** : Carte à carreaux en français. Carte du jeu formée par des tiles.
- **tileset** : Ensemble de tiles sous forme de spritesheet.
- **warp** : Zone qui une fois pénétrée déclenche un changement de map.

2 Jeu

2.1 Déroulement

expliquer les dialogues, conserver le contenu tout en l'adaptant etc...donner des exemples de situations alternatives . et le résultat actuel expliquer la complication lié au temps, le rabotage etc..

2.2 Projet original

Le but du projet original était d'adapter l'entièreté de Candide sous forme de jeu vidéo avec un scénario alternatif et créer tous les assets visuels et auditifs. Le tout en 6 mois, un projet qui aurait été qualifié de totalement réaliste par la pointe d'ironie de Voltaire et celle de mon mentor avisé...

Dans ce projet, le joueur pourrait incarner Candide et vivre une aventure propre aux choix qu'il serait amené à faire durant le jeu. Il dialoguerait avec les personnages qui proposeraient des quêtes à remplir et influenceraient le jeu, il collecterait des objets trouvés le long de son voyage pour les revendre ou les utiliser lors de combats à l'épée ou philosophiques. Il récolterait une collection de graines pour les planter au final dans son jardin... Le tout agrémenté de belles cinématiques.
Le jeu n'aurait pas forcément été agréable avec le joueur, le but étant que ce dernier ressente ce qu'a ressenti Candide lors de son périple. Un exemple de choix qui avait été imaginé était que le joueur ait le choix très tôt dans le jeu de choisir combien de sucre il souhaite dans son thé. Un choix apparemment innocent qui aurait révélé ses conséquences beaucoup plus tard dans le jeu, lorsque Candide rencontre l'esclave qui travaille pour la production de sucre. En fonction de la réponse du début, l'esclave sera plus ou moins maltraité.

2.3 Résultat final

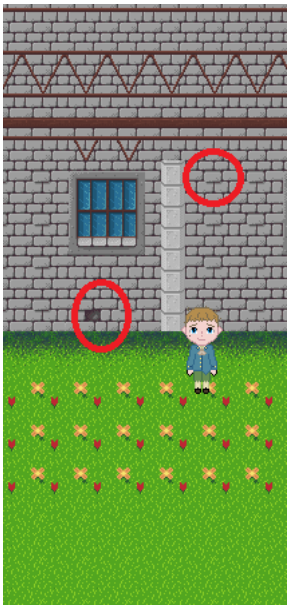
Après avoir réalisé que créer un jeu vidéo n'était pas simple, le projet a été réduit pour adapter uniquement le premier chapitre du livre. Le joueur incarne donc Candide et peut se promener des décors inspirés de l'incipit. Il peut dialoguer avec les pnjs présents dans les décors et faire des combats avec certains d'entre eux.

2.4 Adaptation

L'adaptation est quelque chose de primordial. Comment donner une apparence physique à un personnage ? Le faire parler ? Créer un environnement pour le joueur ? Le meubler ?
Tous les assets sont une interprétation de l'œuvre originale. Par exemple, le design de Pangloss doit faire comprendre immédiatement au joueur que c'est un intellectuel trop sûr de lui. Il porte des lunettes pour accentuer le stéréotype du savant et pour illustrer son discours. Le sprite est déjà marqué de quelques traits de vieillesse, ce qui accentue la "sagesse" de Pangloss. Il plisse des paupières en souriant de coin, ce qui révèle sa confiance. Tout ceci doit transparaître en seulement quelques pixels.

Le personnage de Candide doit faire transparaître de la naïveté, de l'innocence et un peu de maladresse. Ces éléments sont représentés avec ses mèches en bataille et sa chemise à moitié rentrée dans ses chausses, ainsi qu'avec son sourire gentil.

De même que le décor : Il y a la cour et le château. Le château *avait une porte et des fenêtres* et il est fait de pierres taillées. Pour illustrer une part d'ironie de Voltaire, quelques pierres sont irrégulières, voire manquantes.



Malheureusement, le décor n'est pas complet à cause du temps que la création prend. Il est également difficile de créer un jeu sans décor dessiné au préalable.

sources : Phaser.js par Richard Davey Phaser-tilemap-plus par Colin Vella Bosca Ceoil par Terry Cavanagh Pyxel Edit par Danik (pseudonyme)

3 Fonctionement du code

Ce projet a été réalisé en javascript, avec les additions de ECMAScript6 et de deux librairies conçues pour la réalisation de jeu-vidéo : Phaser.js ainsi que Phaser-tilemap-plus.js . La majorité du code comprend des objets et des

classes créés afin de pouvoir les utiliser comme outils de création.

Note : Les extraits de code présentés sont des extraits du code source comprenant les passages les plus importants, ils ne sont pas complets.

3.1 Système général

Avant toute chose, il a fallu concevoir un système qui permette d'atteindre diverses variables au travers de tous les outils du jeu. C'est pourquoi l'objet *globals* (présent dans globals.js), une variable globale a été créée. Cet objet stocke toutes les variables afin d'y avoir accès facilement sans devoir se soucier des scopes des différentes classes et de leur méthodes.

(Pour retenir les diverses actions effectuées par le joueur ainsi que des données importantes telles que son positionnement ou le décor actuellement chargé à l'écran, un autre objet global (présent dans gameRef.js) est utilisé. Il sert de référence pour initialiser le personnage incarné par le joueur dans l'état où ce dernier l'avait laissé.) (pas encore sûre, voir le local storage)

Afin d'étendre les possibilités futures de diffusion du projet, une option de traduction a été pensée dans le code de départ. Il s'agit d'un chiffre (0 pour le français, 1 pour l'anglais, ...) stocké dans l'objet mentionné au paragraphe ci-dessus. Grâce à cette variable, la fonction de dialogList.js retourne les textes figurants dans le jeu dans leur bonne version.

```
1  function setDialog(langue){
2     switch(langue){
3          case 0 :
4          globals.dialogs.myChar=["Bonjour"];
5     break;
6          case 1:
7          globals.dialogs.myChar=["Hello"];
8     break;
9   }
10 }
```

Exemple de la fonction traductrice pour le dialogue d'un personnage quelconque

3.2 Commencement

3.3 Gestion des states du jeu

Les states, propres à Phaser, représentent une section du jeu. Chaque objet state a son lot de fonctions Phaser, à savoir *preload()*, *create()*, *update()* et *render()*. L'utilisation de ces fonctions va être décrite dans les paragraphes ci-dessous.

3.3.1 Boot et Preload

La state "Boot" fait démarrer le jeu. Elle déclenche la state "Preload". Preload est une state qui utilise uniquement la fonction *preload()*, qui sert à associer tous les assets du jeu à un nom et les stocker dans le cache pour pouvoir les utiliser dans le code. Une fois tous les éléments stockés, la state "Menu principal" est enclenchée. Le choix d'utiliser une unique state pour ce travail permet une meilleure organisation au sein du code.

3.3.2 Menu Principal

La state "Menu principal" permet au joueur de lancer le jeu ou d'en modifier la langue. Les boutons qui permettent de faire une action sont des objets Phaser. L'ensemble des boutons est initié dans une seule fonction. Cette fonction dépend de la langue que le joueur a choisi. Chaque bouton a sa propre spritesheet, ce qui pourrait poser problème au niveau de la taille et du temps pour les dessiner si d'autres langues sont ajoutées. Une option serait de générer dans le code, les textes dans les boutons à la place de les dessiner à la main.

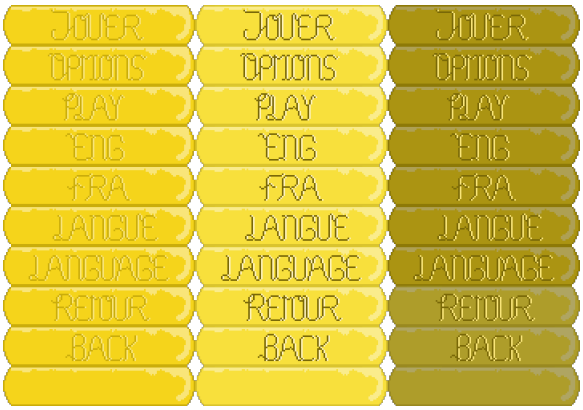


FIGURE 1 – Spritesheet des boutons utilisés dans le menu

3.3.3 Game

La state "Game" utilise les fonctions *create()* et *update()*. Dans la fonction *create()*, toutes les actions de jeu sont initiées : le terrain et le personnage sont initiés et les textes du jeu sont traduits.

```
1  var gameState = {
2      create: function(){
3          setDialog(gameRef.main.langue);
4          createMap1();
5          globals.terrainManager.initMap(globals.maps.chateau,
       true);
6          initPlayer(1182,1152)
7      },
8      update: function(){
9          updatePlayer();
10         globals.terrainManager.update();
11     }
12 };
```

Extrait du code de la state "Game".

La méthode *update()* est une fonction qui utilise *requestAnimationFrame*. Sont présentes dans cette méthode, toutes les fonctions qui servent à interagir avec le joueur, c'est à dire la détection de la collision, les changements de maps au contact d'une zone de warp et l'apparition des bulles de dialogues des personnages non-jouables quand le joueur les approche. La fonction qui sert à faire marcher le joueur est aussi présente dans la fonction *update()*.

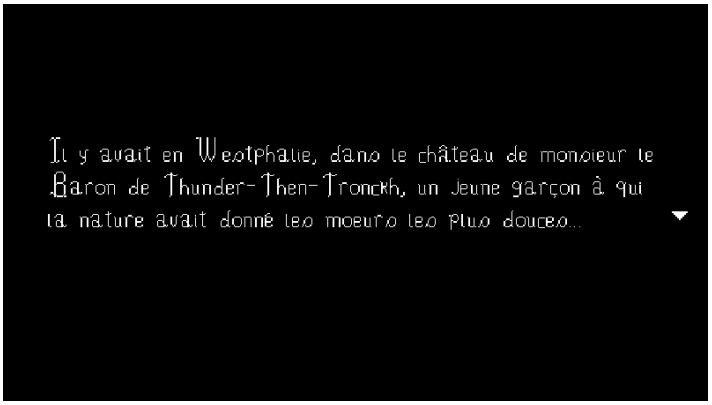
3.3.4 Battle

La state "Battle" est utilisée lors des combats pour initier l'interface de combat. Une nouvelle state est utilisée car il est plus facile de créer un contenu visuel à partir d'une state vide que de modifier une state déjà existante. En effet, la state de combat ne requiert pas une détection de collision ni la création de personnages non-jouables.

Une map de fond est initiée dans la fonction *create()*, ainsi que les personnages à l'écran et leur barres de vie. Le combat est ensuite géré par une classe, textitbattleManager, dont le fonctionnement va être décrit plus loin.

3.3.5 Transition avec texte

Cette state sert uniquement à afficher du texte. Une fonction est passée en argument lorsqu'on fait appel à cette state qui l'exécutera. Cette state est une alternative narrative aux cinématiques, trop difficiles à programmer.



3.4 Joueur

Le joueur est un sprite qui peut se déplacer lorsqu'une touche directionnelle est enfoncée. Ces déplacements sont activés uniquement quand la propriété "canMove" = true. Cette propriété sert à ce que le personnage ne puisse pas se déplacer pendant les dialogues. Le sprite possède 4 animations de marche qui sont jouées quand l'image se déplace.

FIGURE 2 – le sprite du joueur avec les différentes frames des animations de marche

Les déplacements sont détectés dans la fonction *updatePlayer()* appellée dans la state "Game".

Ce sprite possède un corps physique (spécificité de Phaser) qui permet les collisions avec son environnement.

Le sprite possède également de la vie et des attaques, qui vont être utilisées durant les combats.

3.5 Les pnjs

3.5.1 Personnages normaux : La classe *Pnj*

Les pnjs sont une nouvelle classe étendue de la classe sprite de Phaser. Pour créer un nouveau pnj, il faut utiliser :

```
1 var myChar = new Pnj(x,y,key,frame,name,dialogs,faceAnimKey);
```

Voici l'explication de ces arguments :

- **x,y** : Définissent la position du sprite.
- **key** : Référence de la spritesheet sous forme de string.
- **frame** : Numéro qui définit la frame affichée. (habituellement 0 pour avoir le personnage de face)
- **name** : Nom du personnage sous forme de string.
- **dialogs** : Variable contenant les répliques du personnage.
- **faceAnimKey** : Référence à la spritesheet des animations pendant un dialogue sous forme de string.

Chaque pnj a une méthode *update()* qui permet de détecter le joueur et créer une interaction avec lui. Le code regarde si le rectangle formé par le sprite du joueur intersecte un autre rectangle plus grand, qui englobe le pnj. Si c'est le cas et que le phylactère du pnj n'est pas encore à l'écran, le personnage crée une bulle. Lorsque la bulle est affichée, le joueur peut appuyer sur "enter" et faire démarrer le dialogue.

Cette méthode de détection de bulle a été codée avec des "if" et beaucoup de variables booléennes. Une amélioration est peut-être possible avec des signaux, une spécificité de Phaser, qui permettent l'appel d'une fonction lorsque qu'un événement est déclenché.

3.5.2 Ennemis : La classe *Ennemy*

Les ennemis sont une classe étendue de la classe Pnj. Ce sont des pnjs dotés de points de vie et d'attaques pour pouvoir les utiliser lors de combats. Ils ont une propriété importante : *isAlive*, un booléen qui détermine si un combat peut être lancé et permet de mettre fin à un combat lorsque l'ennemi est défait. Chaque ennemi possède une méthode *turn()* utilisée lors d'un tour au combat et une méthode *startCombat()* qui fait basculer le jeu sur la state de combat. Cette méthode sert à définir une cible aléatoire parmi les cibles présentes, choisir une attaque aléatoire et lancer l'attaque sur la cible choisie. Cette méthode renvoie aussi un message qui sera afficher pendant le combat, en fonction de l'attaque choisie.

3.6 Les bulles : La classe *Bubble*

Chaque bulle affichée à l'écran est composée de trois parties, le fond, le texte et un triangle animé. Les bulles jouent plusieurs rôles. Elles peuvent indiquer un dialogue mais aussi permettre au joueur de rentrer dans une nouvelle map (grâce à la méthode *goToHouse()*) lorsqu'il y a un élément tel qu'une porte que le joueur doit ouvrir. La détection des zones de warps à bulle est programmée à l'aide de boucles et de "if". Le code pourrait être simplifié à l'aide de signaux Phaser.

FIGURE 3 – Exemples de phylactères

3.7 Les attaques : La classe *Attaque*

Les attaques des objets de la classe *Attaque*. Elles sont utilisées par le joueur et les ennemis lors des phases de combats. Pour créer une attaque il faut utiliser :

```
1 var myAttck = new Attaque(name,pdg,cEff,minPdg);
```

Les arguments on la fonction suivante :

- **name** : Nom de l'attaque.
- **pdg** : Abréviation de points de dégâts. Le nombre que l'attaque inflige lors de sa première utilisation.
- **cEff** : Abréviation de coefficient d'efficacité, définit de combien les points de dégâts diminuent après chaque utilisation.
- **minPdg** : Nombre de dégâts minimal que l'attaque inflige.

Chaque attaque possède la méthode *nextTurn()*, qui recalcule les points de dégâts en divisant les points de dégâts actuels par le coefficient s'ils ne sont en dessus du minimum fixé.

Elles possèdent aussi les méthodes *normal(cible)*, *rate(cible)* et *coupCritique(cible)* qui infligent différent dégâts (Les dommages sont multipliés par 1.5 quand l'attaque est critique. L'attaque ne fait rien lorsqu'elle rate.)
Ces méthodes sont utilisées dans la méthode *turn(cible)*, qui choisit aléatoirement une de ces méthodes et l'applique sur la cible choisie.

Les attaques possèdent encore deux méthodes qui distribuent de l'information textuelle qui sera utilisée lors des combats. La méthode *info()* indique le nombre de dommages que l'attaque cause. La méthode *desc()* retourne un string en fonction de ce qu'il s'est passé lors du tour.

3.8 Les barres de vie : La classe *Healthbar*

Les barres de vies sont des objets graphiques qui s'adaptent en fonction du nombre de points de vie attribués à un personnages. Elle sont en réalité composées de 3 éléments, un contenant vide, le remplissage rectiligne et le remplissage curviligne près du contenant. Il était plus facile de le dessiner ainsi car l'homothétie posait problème lorsqu'il ne restait que quelques hp.

A chaque tour, en fonction du nombre de points de vie du personnage, la partie colorée se réduit grâce à une homothétie sur la composante x de facteur nouvelle vie / vie maximum. Les sprites sont en teintes de gris, pour permettre une coloration en fonction du nombre de points de vie : vert quand les hp sont au dessus de la moitié du maximum, orange quand ils sont au dessus d'un quart du maximum et rouge quand ils sont en dessous. Quand tous les points de vies sont éliminés, le personnage est défait et l'animation de la partie curviligne de l'animation se déclenche.

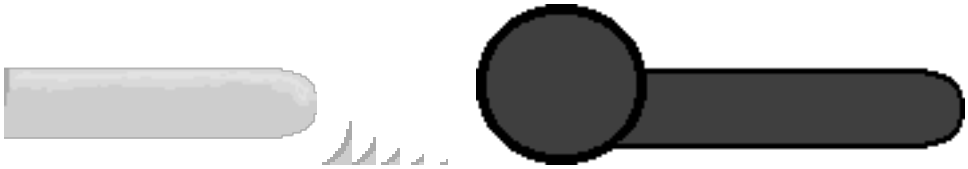


FIGURE 4 – Les trois parties de la barre de vie

3.9 Les objets de terrain : Les classes *CustomMap* et *Warp*

Cette section regroupe deux petits objets traitant du terrains, d'où leur regroupement.

Les customMaps sont des objets qui stockent de l'information de terrain. Ils sont utilisés dans le *terrainManager*. Pour déclarer un objet customMap, il faut utiliser :

```
1 var myCustomMap = new CustomMap(key,tilesets,layerKeys,music);
```

Les arguments remplissent les fonctions suivantes :
- **key** : Référence de la map sous forme de string.
- **tilesets** : Array contenant le nom des tilesets utilisés pour la map.
- **layerKeys** : Le nom des couches de la map.
- **music** : La référence à la musique associée à la map. Il est automatiquement assigné à undefined si l'argument n'est pas rempli.

L'argument music n'est pas encore pris en compte par le code car la musique n'a pas encore été créée. Il est présent pour les utilisations futures.

Chaque objet CustomMap possède une clé *plus* qui permet une définition externe (via Tiled) de certains paramètres comme les coordonnées des pnjs et celles des warps. Chaque objet possède également deux méthodes permettant d'ajouter des pnjs et des warps aux clés correspondantes.

Dans le code conçu initialement, chaque objet CustomMap était dépendant de toutes ses composantes y compris les pnjs et les warps. Le problème étant qu'après avoir intégré la clé *plus*, les pnjs/waprs étaient aussi dépendant de l'objet CustomMap ! Donc impossible de créer l'un ou l'autre, d'où la nécessité d'utiliser des méthodes, une fois que tous les objets ont été créés.

Les Warps sont également des objets de données. Voici comment déclarer un warp :

```
1 var myWarp = new Warp(isHouse,toMap,newX,newY,x,y,tileWidth,
    tileHeight,text);
```

Un Warp est une zone rectangulaire qui permet le changement de map lorsque le sprite du joueur l'intersecte. Les arguments pour déclarer un warp exercent les fonctions suivantes :

- **isHouse** : Booléen servant à définir si le warp s'active directement ou laisse apparaître un phylactère quand le joueur l'intersecte pour l'activer manuellement.
- **toMap** : Nom de la variable contenant un objet CustomMap pour la nouvelle map où le joueur sera amené.
- **newX,newY** : Les coordonnées où le sprite du joueur est affiché sur la nouvelle map.
- **x,y** : Les coordonnées du haut à gauche du rectangle qui compose l'objet Warp.
- **tileWidth,tileHeight** : La largeur et longueur du rectangle en terme de tile de 32x32 pixels. Pour exprimer un rectangle d'une largeur de 64 pixels, il faut utiliser 2 pour tileWidth.
- **text** : Le texte affiché dans la bulle du personnage si le Warp doit être activé manuellement. Exemple : "Entrer ?"

3.10 Gestion des dialogues : La classe *dialogManager*

Tout affichage de texte est géré par la classe *dialogManager* et ses méthodes. Chaque morceau de texte doit avoir une syntaxe précise pour pouvoir être affiché grâce à cette classe. Le texte doit être un array qui contient des strings ou des autres arrays.

```
1   var txtSimple = ["texte simple"];
2   var txtCallback = [
3     ["texte simple avec callback",function(){}]
4   ];
```

Exemple de la syntaxe d'un texte simple et d'un texte suivi d'une callback

```
1   var txtChoix = [
2     [
```

```
      "action avec choix",
      ["choix 1","choix 2"],
      [function(){callback 1},function(){callback 2}]
    ]
  ];
```

Exemple de syntaxe pour un texte qui comporte des choix

FIGURE 5 – Exemple d'un choix onéreux

3.10.1 méthode *start()*

Cette méthode affiche une boîte de dialogue standard à l'écran et crée un objet bitmapText vide. Cet objet est un élément graphique de Phaser qui permet d'afficher du texte avec une police choisie au préalable, en l'occurrence la police créée pour ce projet.

3.10.2 méthode *displayText()*

```
displayText(texts,index,isDialog,faceAnim,battleDesc)
    if(typeof texts[index] == "string"){
        var textArray = texts[index].split(" ");
    }
    else {
        var textArray = texts[index][0].split(" ");
        this.texts = texts;
        this.index = index;
    }

    var compteurMots = 0;
```

```
12
13      this.bmpText.text = "";
14
15      this.wordTimer = game.time.create();
16
17      this.wordTimer.repeat(100,textArray.length,function(){
18          this.bmpText.text += textArray[compteurMots] + " ";
19
20          if(this.bmpText.text.length >= 184){
21              this.wait(isDialog,false,false);
22              this.wordTimer.pause();
23          }
24          compteurMots ++;
25      },this);
26
27          if(typeof texts[index] == "string"){
28              this.wordTimer.onComplete.add(function(){
29                  this.wait(isDialog,true,null,false);
30              },this);
31          }
32          else if(texts[index].length == 2 ){
33              this.wordTimer.onComplete.add(function(){
34                  this.wait(isDialog,true,texts[index][1],false);
35              },this);
36          }
37          else{
38              this.wordTimer.onComplete.add(function(){
39                  this.wait(isDialog,false,null,true);
40              },this);
41          }
42
43      this.wordTimer.start();
44
45  }
```

Extrait de la méthode displayText()
Cette méthode sert à afficher les mots un par un. Voici ce que représente les arguments :

- **texts** : La variable qui contient le texte à afficher.
- **index** : L'index à partir du quel se trouve le texte à afficher.
- **isDialog** : Booléen servant à afficher une animation spéciale si c'est un dialogue.
- **faceAnim** : La référence de la spritesheet à animer lors d'un dialogue.
- **battleDesc** : Objet de donnée qui est utilisé lors des combats. Com-

prend les clés booléennes *is,callback* (qui définissent si le texte doit s'en aller automatiquement et s'il contient une callback) et la clé *time* (qui définit le nombre de milisecondes avant que le texte ne s'efface).

*Note :*L'usage de *isDialog* et *battleDesc* n'est pas présent dans l'extrait ci-dessus.

Si le texte est un dialogue, l'image de la tête du personnage est ajoutée à la boîte de dialogue et l'animation du personnage qui parle est lancée. Cette méthode sépare ensuite le texte d'entrée en array (*textArray*) et initialise un compteur à 0. Elle crée un timer Phaser, qui ressemble dans son fonctionnement à un *setInterval()*. Toutes les 100 milisecondes, un mot est ajouté à l'élément graphique *this.bmpText* et le compteur est incrémenté jusqu'à ce que le timer ait fait l'opération autant de fois que la longueur de *textArray*. Si le nombre de caractères présents dans l'affichage graphique dépasse 184, le programme met en pause le timer et lance la méthode *wait()*, pour éviter de déborder de la boîte de dialogue.

Quand le timer a terminé d'afficher tous les mots, il lance la méthode *wait()* en fonction du type de texte affiché. Si le texte a un argument battleDesc, la méthode *stop()* est appelée directement lorsque le timer est terminé.

Finalement, cette méthode enclenche le timer (il n'a que été créé auparavant).

3.10.3 méthode *wait()*

```
1  wait(isDialog,isLast,callback,isQuestion){
2      input.enter.onDown.addOnce(function(){
3        if(!isQuestion){
4           if(isLast && callback != null){
5                callback();
6            }
7            else if(isLast){
8                this.stop(isDialog,true);
9            }
10           else{
11               this.resume(isDialog);
12           }
13       }
14       else{
15           this.question(this.texts,this.index);
16       }
17   },this);
```

```
18 }
```

Extrait de la méthode wait()
Cette méthode est un relai qui demande l'appui de la touche "enter" pour continuer quoi que se soit. Voici à quoi servent les nouveaux arguments :

- **isLast** : Booléen servant à fermer le dialogue s'il est complètement affiché.
- **callback** : Une fonction callback qui est appelée à la place de la méthode *stop()*
- **isQuestion** : Booléen servant à appeler la méthode *question()* si le texte contient des choix.

Cette méthode dessine et anime un petit triangle d'animation pour signaler au joueur que le script attend une action de sa part. Si le texte affiché est un dialogue, l'animation du personnage est changée : Il cligne simplement des yeux. Cette méthode utilise un signal Phaser. Quand la touche enter est pressée, une méthode est appelée en fonction des besoins du texte qui doit être affiché.

3.10.4 méthode *resume()*

Cette méthode prend l'argument *isDialog* et relance si c'est le cas l'animation du personnage qui parle. Elle supprime le triangle animé, remet à zéro le contenu de bitmapText et relance le timer de la méthode *displayText()* :

3.10.5 méthode *question()*

```
1  question(texts,index){I
2      for(var k=0;k<texts[index][1].length;k++){
3          var answerBox = game.add.image(0,0,"answerBox");
4          answerBox.alignTo(this.dialBox,Phaser.TOP_RIGHT,0,(k
       *50)+k*1);
5
6          var answer = game.add.bitmapText(0,0,"candideFont",
       texts[index][1][k],50);
7          answer.alignIn(answerBox,Phaser.LEFT_CENTER,-15,-10);
8
9          this.answerList.push(answer);
10         this.answerBoxes.push(answerBox);
11     }
```

```
12      this.selectionBox = game.add.sprite(this.answerBoxes[0].x,
        this.answerBoxes[0].y,"selection");
13      this.selection(texts,index);
14  }
```

Cette méthode fonctionne à l'aide des arguments *texts* et *index* précédemment mentionnés. Elle supprime le triangle animé, mets à zéro deux arrays contenant les boîtes de réponses ainsi que ces dernières. Ensuite, grâce à une boucle, elle crée les éléments graphiques des réponses (la boîte et le texte). Une fois cela terminé, elle ajoute un sprite en forme de cadre sur la première réponse. Il permet au joueur de voir la réponse qu'il souhaite sélectionner. La méthode *selection()* est ensuite appelée.

3.10.6 méthode *selection()*

```
1   selection(texts,index){
2       input.enter.onDown.removeAll(this);
3       input.up.onDown.removeAll(this);
4       input.down.onDown.removeAll(this);
5
6       var max = this.answerBoxes.length;
7
8       input.up.onDown.addOnce(function(){
9           if(this.selectionBox.y > this.answerBoxes[max -1].y){
10              this.selectionBox.y -=51;
11          }
12          this.selection(texts,index);
13      },this);
14
15      input.down.onDown.addOnce(function(){
16          if(this.selectionBox.y < this.answerBoxes[0].y){
17              this.selectionBox.y +=51;
18          }
19          this.selection(texts,index);
20      },this);
21
22      input.enter.onDown.addOnce(function(){
23          input.up.onDown.removeAll(this);
24          input.down.onDown.removeAll(this);
25          for(let k=0;k<this.answerBoxes.length;k++){
26              if(checkSpriteOverlap(this.selectionBox,this.
        answerBoxes[k])){
27                  texts[index][2][k]();
28              }
29          }
```

```
30          for(let d=0;d<this.answerBoxes.length;d++){
31              this.answerBoxes[d].destroy();
32              this.answerList[d].destroy();
33          }
34          this.selectionBox.destroy();
35      },this);
36  }
```

Cette méthode permet au joueur de choisir une option lorsqu'il est confronté à un choix. Il peut, à l'aide des touches directionnelles verticales, déplacer le sprite cadre. Lorsqu'il appuie sur enter, la callback associée à la réponse se déclenche.

Cette méthode fonctionne de la manière suivante : premièrement, elle supprime toutes les callbacks associées au signal de chaque input pressé. Ensuite, pour chaque input de verticalité, la méthode ajoute une callback qui ne pourra être utilisée qu'une seule fois lorsqu'une touche est appuyée. Cette callback modifie les coordonnées y du sprite cadre et rappelle la méthode *selection()*. Pour la touche enter, une callback à utilisation unique est aussi associée. Elle supprime les callbacks des touches directionnelles, vérifie quelle réponse est entourée par le cadre grâce à une fonction qui renvoie *true* si les rectangles formés à partir du pourtour de deux sprites s'intersectent. Une fois la réponse déterminée, la méthode lance la callback qui lui est associée et détruit les éléments graphiques qui constituaient l'ensemble des réponses.

3.10.7 méthode *stop()*

Cette méthode prend les arguments *isDialog* et *canBulle*. L'argument *canBulle* est un booléen qui détermine si le pnj à qui le joueur parle peut ré-afficher un phylactère à l'écran.

Cette méthode supprime tous les éléments graphiques à l'écran y compris les animations des personnages si *isDialog* = *true*. Lorsque que les deux arguments de la fonction sont égaux à true, un timer phaser est activé (similaire au *setTimeOut()*) et la bulle du personnage qui vient de parler réapparaît à l'écran après 300 milisecondes.

3.10.8 Texte automatique : Les méthodes *startDialog(pnj)*, *startBattleDesc(text,battleDesc)* et *desc(text)*

Ce sont 3 méthodes utilisées dans le jeu qui appellent les différentes méthodes citées précédemment avec les arguments nécessaires.

La méthode *startDialog()* dépend uniquement d'un pnj. Elle affiche le nom du pnj dans un contenant graphique et lance le dialogue en fonction l'index du pnj.

la méthode *startBattleDesc()* est utilisée dans les phases de combats et gère les descriptions automatiquement afin que le joueur n'ait pas à intervenir pour que le texte change.

La méthode *desc()* sert à afficher du texte pur. Elle est utilisée dans la state qui s'occupe des transitions avec texte. Cette méthode sera aussi utilisée lorsque l'interaction avec l'environnement sera implémentée.

3.10.9 méthode *endBattleScreen()*

Cette méthode sert à afficher progressivement une boîte de dialogue ainsi que le texte approprié lorsque le joueur gagne ou perd un combat. Quand le texte est affiché et que le joueur appuie sur enter, le jeu bascule de la state "Battle" à la state "Game". Dans l'état actuel de ce projet, il retourne au même point de départ que lorsque le jeu est lancé mais dans des versions futures, le joueur retrouvera la position qu'il a quittée quand le combat s'est lancé.

3.11 Gestion du terrain : La classe *TerrainManager*

Le *terrainManager* est un objet qui grâce à ses méthodes, gère les changement de terrains.

3.11.1 méthode *initMap()*

```
1
2       initMap(map,collision){
3           this.currentMap = game.add.tilemap(map.key);
4
5           for(let l in map.tilesets){
6               this.currentMap.addTilesetImage(map.tilesets[l],map
    .tilesets[l]);
7           }
8
9           this.currentLayers = [];
10          for(let k of map.layerKeys){
11
12              let layer = this.currentMap.createLayer(k);
13              this.currentLayers.push(layer);
14          }
15
16          if(collision){
17              this.collision = this.currentMap.createLayer("
    Collision");
18              this.currentMap.setCollisionByExclusion([], true,
    this.collision);
19              this.collision.resizeWorld();
20              this.collision.visible = false;
21          }
22          this.currentMap.plus.animation.enable();
```

```
23          this.currentLayers[0].resizeWorld();
24
25          for(let l in map.pnjs){
26              game.add.existing(map.pnjs[l]);
27              this.currentPnjs.push(map.pnjs[l]);
28          }
29          for(let l in map.warps){
30              this.currentWarps.push(map.warps[l]);
31              this.currentWarps[l].overlap=false;
32          }
33      }
```

Cette méthode initialise tous les composants du terrain. Elle prend en compte 2 arguments :

- **map** : L'objet CustomMap qui contient les données du terrain que l'on souhaite afficher.
- **collision** : Booléen qui définit si le terrain possède une couche nommée "Collision".

La couche de collision possède toutes les tiles avec lesquelles le joueur doit rentrer en collision.

Cette méthode crée un objet Phaser Map, puis construit visuellement chaque couche avec les tilesets. Les personnages sont ensuite ajoutés et enfin les warps. Chaque carte affichée est enregistrée sous une clé inaccessible en dehors de la méthode une fois que le décor est posé, d'où la nécessité d'utiliser les CustomMaps.

3.11.2 méthode *clearMap()*

Cette méthode supprime tous les éléments graphiques de l'écran. Elle réinitialise également les clés *currentLayers*, *currentPnjs* et *currentWarps*.

3.11.3 méthode *changeMap()*

Cette méthode s'active lorsque le joueur pose le pied sur un warp. Elle crée un fondu enchaîné sur du noir, détruit tout ce qu'il y a a l'écran, reconstruit le nouveau décor puis l'écran noir se fond à nouveau sur le jeu.

Lors de la réalisation, le fondu a été programmé en plus des autres méthodes mais il ne marchait qu'à moitié : Même si le personnage bougeait, le rectangle noir qui obstruait la fenêtre de vue restait à la même place !

C'est en s'aventurant dans la documentation que des méthodes permettant ces effets ont été découvertes...

3.12 méthode *update()*

Cette méthode établit la collision entre le joueur et l'environnement ainsi que les personnages composants la scène. Elle vérifie aussi les détections d'intersections avec les warps pour afficher si besoin une bulle.

Il y a eu des problèmes avec les bulles, car dans une première version du code, elles étaient constamment supprimées. C'est expliqué par le fait que le warp pénétré n'était pas retenu et que le code vérifiait pour chaque warp présent dans la map si le joueur était dedans. Comme les conditions étaient vérifiées, la bulle était supprimée aussitôt qu'elle était créée. Le problème a été géré en associant la zone de warp à une variable et de vérifier le contenu de cette variable avant de supprimer la bulle.

3.13 Gestion des combats : La classe *BattleManager*

Cette state est responsable des combats du jeu. Elle gère les tours des combats et orchestre les affichages.

3.13.1 méthode *init()*

3.13.2 méthode *clearMap()*

3.13.3 méthode *clearMap()*

3.13.4 méthode *clearMap()*

3.13.5 méthode *clearMap()*

3.13.6 méthode *clearMap()*

3.13.7 méthode *clearMap()*

3.13.8 méthode *clearMap()*

3.13.9 Gestions des signaux

3.14 Interaction avec les objets

à compléter idée : un objet possède ou non un callback

3.15 En résumé

Beaucoup de code pour au final, un petit rendu. En effet, programmer un jeu de type RPG est beaucoup plus compliqué que prévu. Chaque fonction pour la progression du jeu est unique et il est très difficile d'avoir ne serait-ce qu'un minimum de scénario. Le niveau de complexité est déjà assez élevé rien qu'en programmant un long dialogue qui ne sera affiché qu'une fois au cours du jeu, comme par exemple les quelques lignes de Pangloss. Ce à quoi mon projet s'approche le plus est peut-être une situation de base inspirée du premier chapitre de Candide, mais il est loin d'être proche d'un

jeu. Pourtant tout ce qui a été programmé est indispensable pour chaque jeu vidéo de type RPG car ce sont les éléments de base pour le construire. Le code est quelque chose d'invisible pour le joueur et il peut être très frustrant de constater qu'après tant de code écrit, seul un petit résultat est affiché à l'écran.

Certains éléments qui ont été programmés mériteraient d'être réécrits avec les connaissances acquises lors de ce projet. En effet, les premiers programmes ont été créés en découvrant encore Phaser, ils n'exploitent donc pas forcément toutes les fonctionnalités Phaser qui permettent parfois d'avoir un code plus simple, propre et efficace.

4 Scénario alternatif

Cette section va brièvement présenter les idées de bases pour le jeu ainsi que le travail effectué sur le scénario. Ce travail a été réalisé tôt dans la durée du projet, lorsque l'espoir de réaliser un jeu complet était encore à l'horizon. Malheureusement, il ne fait pas encore l'objet de la réalisation en ce moment, mais il servira de canevas pour le futur.

5 Conception des assets

5.1 Visuel

5.1.1 Police personnalisée

La police du jeu est une police conçue pour le projet. En effet, Phaser permet d'afficher du texte de manière assez simple avec des polices standards, le problème est que les polices décadrent avec l'ambiance du jeu. Phaser permet aussi d'afficher du texte avec des polices bitmap. Il était également possible d'en choisir une déjà existante et de créditer son auteur mais aucune police ne correspondait aux attentes. La meilleure solution était donc de créer une police personnalisée, qui intègrerait sans problème le style graphique du jeu.

Pour créer la police, deux sites ont étés utilisés, calligraphr.com ainsi que kvazars.com/littera.

Calligraphr permet de télécharger un modèle à remplir avec les lettres souhaitées puis de convertir ensuite ces images en police de format trueTypeFont. Grâce à Littera, il est ensuite possible de convertir cette police en police bitmap pour être utilisée dans le jeu.

Un des problèmes rencontrés en cours de réalisation était la précision du rendu. Il fallait utiliser des équivalent de pixels, des carrés de 11X11px pour avoir un rendu pixelisé car les modèles de Calligraph font la grandeur d'une feuille A4.
Un autre problème était le rendu avec Littéra. En effet, le site a appliqué un filtre d'anti-alias automatiquement sur les lettres, ce qui gâchait l'effet du pixel art. Il a donc fallu retoucher toute les lettres à la main sur la feuille finale. Au cours de la réalisation, plusieurs lettres ont du être ajoutées, ce qui a eut comme effet de modifier la disposition des lettres sur la feuille finale et donc de devoir recommencer le travail fastidieux de les redessiner.

5.1.2 Sprites de dialogues

techniques(couches, dessins - restrictions), exemple , projet noir blanc

5.1.3 Sprites d'overworld

techniques, exemples

5.1.4 Tilesets

techniques, exemples

5.1.5 Tilemap

5.2 Musical

logiciel utilisé , techniques (trouver une suite d'accord, les instruments etc..) , éventuellement les bruitages , la campanella midi file

5.3 En résumé

Encore une fois, beaucoup de travail pour un petit résultat. Le pixel art demande beaucoup de patience et créer un univers cohérent et agréable visuellement est difficile sans assets complets. Malheureusement, une partie des assets n'est pas encore utilisée dans le projet rendu. En effet, ayant pensé

avoir le temps de développer plus d'histoire, d'autres personnages ont été dessinés au début du projet. Ils figureront tout de même en annexe.

6 Conclusion

6.1 Nina Ionescu ou l'optimisme

6.2 Expérience gagnée

6.3 Continuation du projet

7 Sources

8 Annexes