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 Déroulement du jeu PAS FINI

Le joueur incarne Candide et peut se promener dans des situations inspirées du livre. Il peut dialoguer avec les pnjs présents dans les décors et faire des combats avec certains d'entre eux. Le joueur arrive dans la cour du château du Baron de Thunder-then-Tronck. sources : Phaser.js par Ri-

chard 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()*.

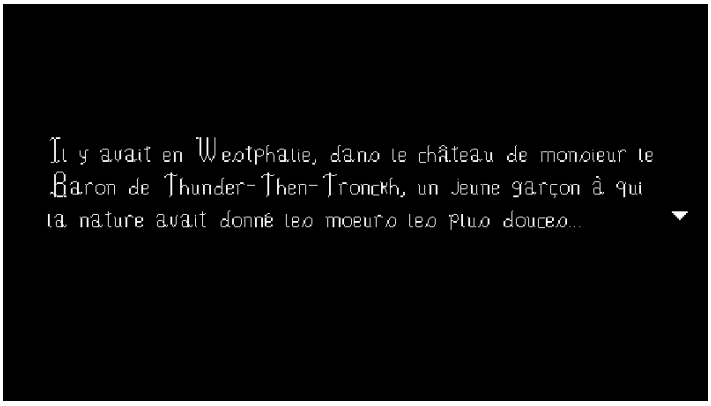
3.3.4 Battle

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()* (décrite dans la section gestion du combat), utilisée lors d'un tour au combat et une méthode *startCombat()* qui fait basculer le jeu sur la state de combat.

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 *Attack*

3.8 Les barres de vie : La classe *Healthbar*

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

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    [
3      "action avec choix",
4      ["choix 1","choix 2"],
5      [function(){callback 1},function(){callback 2}]
6    ]
7  ];
```

Exemple de syntaxe pour un texte qui comporte des choix

FIGURE 4 – 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()*

```
1  displayText(texts,index,isDialog,faceAnim,battleDesc)
2      if(typeof texts[index] == "string"){
```

```
3           var textArray = texts[index].split(" ");
4       }
5       else {
6           var textArray = texts[index][0].split(" ");
7           this.texts = texts;
8           this.index = index;
9       }
10
11      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. Comprend 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       // alert("selection");
3       input.enter.onDown.removeAll(this);
4       input.up.onDown.removeAll(this);
5       input.down.onDown.removeAll(this);
6
7       var max = this.answerBoxes.length;
8
9       input.up.onDown.addOnce(function(){
10          if(this.selectionBox.y > this.answerBoxes[max -1].y){
11              this.selectionBox.y -=51;
12          }
13          this.selection(texts,index);
14      },this);
15
16      input.down.onDown.addOnce(function(){
17          if(this.selectionBox.y < this.answerBoxes[0].y){
18              this.selectionBox.y +=51;
19          }
20          this.selection(texts,index);
```

```
21      },this);
22
23      input.enter.onDown.addOnce(function(){
24          input.up.onDown.removeAll(this);
25          input.down.onDown.removeAll(this);
26          for(let k=0;k<this.answerBoxes.length;k++){
27              if(checkSpriteOverlap(this.selectionBox,this.
    answerBoxes[k])){
28                  texts[index][2][k]();
29              }
30          }
31          for(let d=0;d<this.answerBoxes.length;d++){
32              this.answerBoxes[d].destroy();
33              this.answerList[d].destroy();
34          }
35          this.selectionBox.destroy();
36      },this);
37  }
```

3.10.7 méthode *stop()*

3.10.8 méthode *startDialog()*

3.10.9 méthode *endBattleScreen()*

3.11 Gestion du terrain : La classe *TerrainManager*

3.11.1 méthode *endBattleScreen()*

3.11.2 méthode *endBattleScreen()*

3.11.3 méthode *endBattleScreen()*

3.12 Gestion des combats : La classe *BattleManager*

3.13 Interaction avec les objets

à compléter idée : un objet possède ou non un callback

3.14 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.

Certains éléments qui ont été programmés mériteraient d'être réécrits avec les connaissances acquises lors de ce projets. 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 simple, propre et efficace.

4 Scénario alternatif

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..

5 Conception des assets

5.1 Visuel

5.1.1 Police personnalisée

5.1.2 Sprites de dialogues

techniques(couches, dessins - restrictions), exemple

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

1