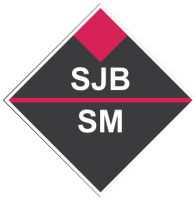
Institut Saint Jean Berchmans - Sainte Marie
Section informatique



Programmation
The Shell Adventure

Travail de fin d'étude
réalisé par
Jordan Dalcq

Année académique 2018 - 2019

Table des matières

Remerciments

Je tiens tout d'abord à remercier mon grand père Yvan, qui m'a introduit au monde de l'informatique dés mon plus jeune âge, et de m'avoir montré aussi les joies de Linux dés mes 7 ans.

Je remercie mes parents Fabienne et Marc pour avoir investis en moi afin de me permettre de continuer sur ma voie vers un métier qui me pasionne.

Je remercie Alex Roşca, d'avoir été mon premier amis, dans cette grande aventure, merci de m'avoir permis de découvrire la programmation

Je remercie Monsieur David Carrera pour avoir été le seule professeur qui a été capable à me poussé au meillieur dans ma passion.

Je tiens aussi à remercier toutes les personne que j'ai rencontré à l'Institut Saint Jean Berchmans, pour m'avoir influencé dans ce projet d'une manière ou d'une autre.

Première partie

Introductions

Contexte

Durant quelques années j'ai été mentor au Coderdojo de Liège durant 2 ans. J'animais un atelier Python / Linux auprès de jeunes âgés entre 14 et 18 ans. Le problème lors de l'apprentissage était qu'ils ne savaient pas quelle commande / fonction utiliser dans un cas précis. Dans ce projet, j'ai décidé de me focaliser sur le terminal bash et de présenter un outil d' apprentissage basé sur le visuel.

Chapitre 1

Présentation

1.1 Fonctionnement

L'idée de base est assez simple, il faut taper des commandes pour interagir avec le monde qui nous entoure.

Comme par exemple :

- cd : Pour se déplacer d'une pièce à une autre touch : Pour créer un objet ou bien faire apparaître une personne
- cp : Pour cloner un objet ou personnage
- mv : Pour déplacer un objet ou personnage
- cat : Pour connaître le contenu d'un objet ou alors l'identité d'un personnage
- rm : Pour jeter un objet ou "éliminer" un personnage
- tree : Scanner à rayon X pour voir à travers les murs

A cause du temps assez limité, j'ai implimenté un langage de programmation afin de laisser une liberté de choisir le pouvoir de chaque commande à l'utilisateur

1.2 Élement

Énumérez et décrivez, dans l'ordre et en français, les différentes parties de votre application. Il s'agit de présenter les différents éléments (ainsi que leurs caractéristiques) qui sont amenés à interagir au sein de votre programme.

FIGURE 1.1 – Capture d'écran de la disposition par défault

1.3 Interface utilisateur

Par défaut, l'interface du jeu se présente comme représenté sur la figure 1.1

Élément visuel / RPG : Là où seront dessinés les personnages et objet (armoir, chaise, garage, maison ...)

Ligne de commandes : Permet d'introduire des lignes de commande SH, qui seront interprétés par la suite.

Objectifs : Permet d'afficher les tâches que le joueur doit effectuer

Il est aussi important de noter que chaque scripts est libre de changer cette disposition comme bon l'entend

Chapitre 2

Analyse

2.1 Classes

2.1.1 game.mechanics.term.Term

La classe *Term* permet de créer des surfaces d'interaction tel que ligne de commande et entrée standard pour l'utilisateur, une fois initialisé elle charge la police de caractères monospace (en 22 pixels car c'est beaucoup plus lisible et agréable à utiliser), ensuite elle crée un dossier *.shelladv* dans le dossier personnel de l'utilisateur (Exemple : */home/dalcjor/.shelladv*)

Attributs

- `surface : pygame.Surface` → Surface principale du terminal
- `mono : pygame.font.Font` → Police de caractère monospace
- `visualLine : List[str]` → Lignes visibles sur la surface du terminal
- `lineRect : pygame.Rect` → Rectangle de l'entrée utilisateur
- `blinkRect : pygame.Rect` → Rectangle avec le rectangle clignotant
- `fontSurface : pygame.Rect` → Rectangle avec les lignes précédentes
- `inInput : bool` → Permet de savoir si l'utilisateur est en train d'entrer du texte
- `promptVisual: bool` → Permet de savoir si le terminal doit afficher le prompt
- `bash: bool` → Permet de savoir si le terminal doit exécuter les commandes entrées ou pas
- `currentTyping : str` → Stock l'entrée utilisateur
- `blinkX : int` → Position x du rectangle clignotant à côté du prompt
- `custom : str` → Stock les prompts customisés
- `history : mechanics.term.History` → Stock les prompts customisés
- `env : Dict[str, str]` → Variables d'environnements semblables à celles présentes dans les systèmes UNIX

— `prompt: str` → Permet de prendre connaissance du nom d'utilisateur, nom de la machine et le CWD (Current Work Directory)
— `tick: float` → Heure actuel pour permettre de réguler la vitesse du blink (le rectangle à côté du prompt)

Méthodes

— `resize(size: Tuple[int, int])` → Change la taille de la surface du terminal
— `disable_prompt()` → cache le prompt
— `enable_prompt()` → Ré-affiche le prompt
— `isprompt_enabled() : bool` → Retourne `True` si le prompt est affiché
— `disable_bash()` → Désactive l'exécution des entrées
— `enable_bash()` → Réactive l'exécution des entrées
— `getInput()` → Active l'entrée utilisateur
— `removeLine()` → Supprime la dernière ligne affichée dans le terminal
— `set_custom_prompt(string: str)` → Permet de mettre un prompt customisé
— `add_to_display(output: str)` → Permet d'ajouter une ligne à l'attributs `visualLine`, qui servira ensuite pour l'affichage
— `clear()` → Permet d'effacer les lignes affichées à l'écran
— `get_env(): Dict[str, str]` → Retourne les variables d'environnements
— `draw()` → Permet de dessiner les lignes de commandes entrées par l'utilisateur
— `drawBlink()` → Dessine le rectangle qui clignote à côté du prompt
— `keydown()` → Méthode de traitement des touches pressées, pour ensuite les affichéer dans la console
— `update()` → Méthode qui gère le curseur et appelle la méthode `draw`
— `get_surface(): pygame.Surface()` → Actualise et retourne la surface du terminal

2.1.2 game.mechanics.term.History

Comme dans tout bon shell, il y a un gestionnaire d'historique, cela permet de rechercher des commandes déjà tapées. Les historiques sont sauvegardé de façon à ce que l'utilisateur puisse retrouver ces commandes dans le jeu.

Attributs

- `home: str` → Chemin vers le dossier utilisateur
- `hist: file` → Lien en lecture vers le fichier d'historique (`\$HOME/.bash_history`)

Méthodes

- `__getitem__(index: int) : str` → retourne une ligne spécifique de l'historique
- `append(line: str)` → ajoute une ligne à l'historique
- `openFile()` → Ouvre le fichier d'historique
- `get_size() : int` → Retourne le nombre de commande présent dans l'historique
- `get_previous() : str` → Retourne la command qui a été précedemment introduite
- `get_next()` → Retourne la commande qui suis

2.1.3 game.mechanics.rpg.Rpg.Rpg

La classe `Rpg` permet d'afficher les éléments visuels.

Attributs

- `sprites: Dict[str, game.mechanics.rpg.sprite.Sprite]` → Dictionnaire qui lie un sprite à un nom (Le nom étant une variable du Adventure Script), ce dictionnaire contient tous les sprites à afficher à l'écran
- `surface: pygame.Surface` → Surface du Rpg

Métohdes

- `add_to_surface(name: str, sprite: game.mechanics.rpg.sprite.Sprite)` → Permet d'ajouter un sprite à l'écran
- `update()` → Actualise la surface et applique les sprites à la surface
- `resize(size: Tuple[int, int])` → Change la taille de la surface

2.1.4 game.mechanics.rpg.sprite.Sprite

Cette classe permet de gérer les spritesheet automatiquement, grâce au *Adventure Script* (voir autre dossier pour syntaxe), il est possible de gérer des spritesheet avec 4 actions dans cette ordre précis : Pars vers le haut, pars à Droite, pars à Gauche, pars vers le bas. Le premier sprite doit être celui par défaut (au repos). Par contre vous pouvez mettre autant de frames à une action que vous voulez. Toutes les configurations nécessaires sont faites via *Adventure Script*

FIGURE 2.1 – Exemple de spritesheet compatible

Attributs

- `size: Tuple[int, int]` → Taille d'une frame du sprite
- `finalSize: Tuple[int, int]` → Taille finale d'une frame (car elles sont souvent trop petites)
- `index: Tuple[int, int]` → Position de la frame actuelle
- `spriteSurface: pygame.Surface` → Surface dédiée à une frame
- `spriteSheet: pygame.Surface` → Surface qui contient toute la spritesheet au complet

Méthodes

- `get_pos() : Tuple[int, int]` → Retourne la position du sprite sur la Surface dédié au RPG
- `get_surface(): pygame.Surface` → Applique une frame à l'attribut
- `spriteSurface` et retourne la surface
- `reset(index: int)` → Met le sprite à une position repos en fonction de sa direction, s'il est à sa dernière frame ou s'il change de direction
- `move(self, way: int)` → Fait passer le sprite à sa frame suivante
- `stop(way: int)` → Met le sprite en repos, car il a fini de se déplacer

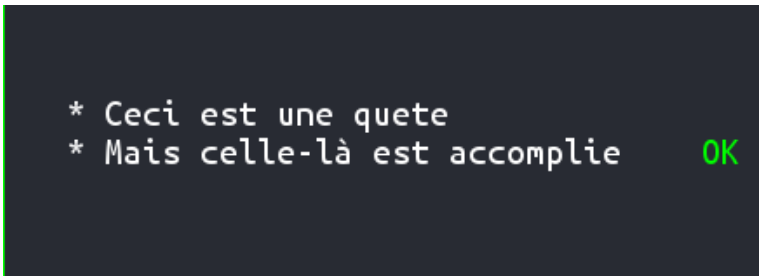


FIGURE 2.2 – Exemple de quêtes

2.1.5 game.mechanics.quest.Quest.Quest

Cette classe permet de gérer les quêtes grâce au langage implémenté dans le jeu (voir autre dossier pour syntaxe). Il est important de noter que le jeu est limité à 18 quêtes à la fois.