KEYS IN TIME

HTTP://PERSO.ESIEE.FR/~MAILHARE

A3P 2016 G8

Auteur

Erwan Mailharro

Thème

Perdu à travers le temps et l'espace, un voyageur doit retrouver ses clés pour retourner à son époque

Résumé du scénario

Vous jouez un scientifique qui a inventé une machine à voyager dans le temps. Lors du premier essai de la machine, vous perdez les clés dans le temps. Vous devez donc les retrouver.

Scénario détaillé

Vous jouez un scientifique qui a inventé une machine à voyager dans le temps. Lors du premier essai de la machine, vous perdez les clés dans le temps. Vous devez donc les retrouver. Elles sont entre les mains d'un collectionneur de katana. Il vous faudra donc voler un katana à un samurai, et éviter un dinosaure pour pouvoir rentrer chez vous.

Détails des Lieux, items, personnages

Présent : point de départ, il y a 2 personnages qui ne servent à rien pour l'histoire.

Faille : C'est le point central de l'histoire ; il y a trois portes vers les autres époques et la machine à voyager dans le temps pour revenir au présent. Revenir au présent requiert la clé.

Préhistoire : il y a un portail pour aller dans la 2e faille, un portail pour revenir à la 1ere faille, et un portail vers le moyen âge. Il y a un Dinosaure qui tue le joueur s'il tente de lui parler.

Moyen Age : Il y a un enfant, qui donne un indice sur la méthode à suivre pour ouvrir la maison. Il y a aussi une tomate, et la maison.

Maison : C'est la maison d'un vieil homme. Il a récupéré la clé de la machine. Le vieil homme est un collectionneur de katana, c'est pourquoi il faut un katana pour entrer dans la maison.

Japon : Il y a un temple, et un samurai en pleine méditation. Si le joueur parle au samurai avec le katana dans l'inventaire, le samurai tue le joueur et met fin à la partie.

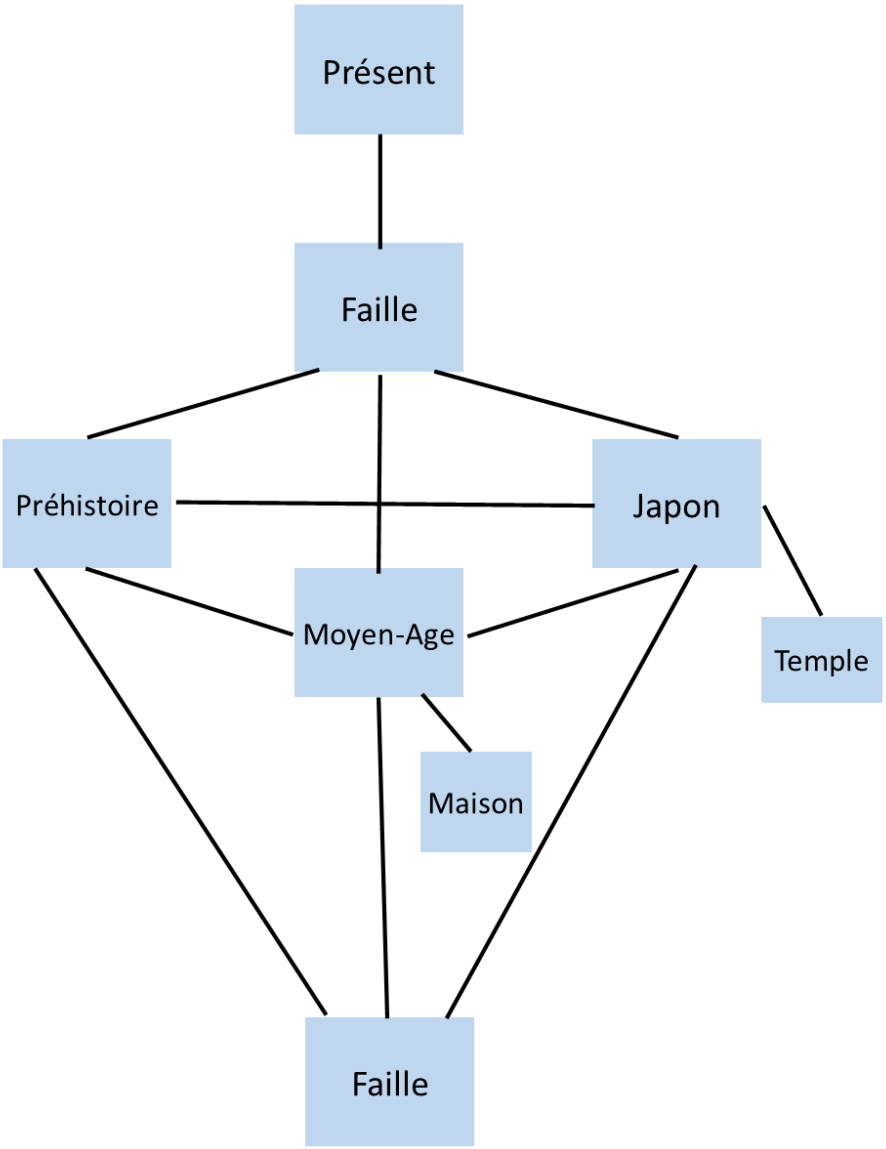
Temple : Dans le temple il y a le katana.

2e Faille : Dans cette salle se trouve le beamer, et un portail qui envoie à une époque aléatoire.

Situations gagnantes et perdantes

Situations gagnantes : le joueur repasse dans la machine avec la clé.ituations perdantes : le joueur parle au dinosaure, le joueur parle au samurai en ayant le katana dans l'inventaire, le joueur atteint la limite de temps (très très longue…)

Plan


Présent
Faille
Préhistoire
Japon
Moyen-Age
Temple
Maison
Faille

Réponses aux exercices

7.5

Dans createRooms() et printWelcome() , on utilise le même code pour afficher les informations sur l'emplacement actuel.

Pour éviter la duplication, j'ai déplacé ces morceaux de code par une procédure printLocationInfo(), qu'il suffit d'appeler dans createRooms() et printWelcome().

7.6

Dans la classe Game, on a besoin d'accéder aux attributs aNorthExit , aEastExit … de la classe Room pour avoir les informations sur les sorties. Pour respecter le principe d'encapsulation, ces attributs doivent être privés. Dans cet exercice, on effectue cette modification, et pour pouvoir y accéder depuis la classe Game on ajoute un accesseur qui permet d'accéder aux quatres sorties à partir de la String de direction

7.7

Dans cet exercice, il faut déplacer le code qui génère la String des Exits, qui se trouve dans la classe Game, dans la classe Room.

On crée une fonction publique dans Room qui renvoie cette String.

Cette modification permet de réduire le couplage de la classe Room et la classe Game.

7.8

Cet exercice a pour but de changer la gestion des sorties des Rooms. Avant cet exercie, une Room ne pouvait avoir que quatre sorties : nord , sud ,est et ouest ; qui sont stockées chacune dans un attribut.

Afin de simplifier le code, on remplace ces attributs par une HashMap qui relie une direction avec la Room correspondante.

On crée une nouvelle méthode setExit(String pDirection, Room, pRoom) qui ajoute la sortie a la HashMap avec sa direction.

La méthode setExits qu'on utilisait précédemment est désormais obsolète.

7.8.1

Dans cet exercice il faut ajoute un déplacement vertical. Pour cela, il faut ajouter dans la méthode createRooms() de la classe Game deux lignes :

Room1.setExit(« up »,Room2) ;

Room2.setExit(« down »,Room1) ;

7.9

La méthode keySet renvoie un Set qui contient toutes les String clés de la HashMap

7.10

La nouvelle méthode getExitString avec la HashMap initialise une String avec « Exits : «

Ensuite, une boucle parcours le keySet de la HashMap et ajoute chaque String a la String renvoyée.

7.11

Implémentation de la méthode getLongDescription() décrite dans le livre.

7.14

Ajout d'une commande look() au jeu comme décrit dans le livre.

7.15

Création d'une méthode eat() qui affiche juste un message, ajout de « eat » dans la classe CommandWords.

7.16

Implémentation des commandes showAll et showCommands comme décrites dans le livre.

7.18

Création de la méthode getCommandList() décrite dans le livre.

7.18.1

Ajout des modifications de zuul-better au projet.

7.18.3

Recherche des images pour les Rooms. Les images sont temporaires, et seront obsolète lors de l'implémentation de l'IHM.

7.18.6

Le but de cet exercice est d'ajouter des images, comme sur Zuul with images.
La classe game est donc réparties sur deux nouvelles classes : GameEngine et UserInterface .

7.18.8

Dans cet exercice il a fallut ajouter un bouton. L'ajout du bouton a été compliqué à comprendre.
Premièrement, il a fallu déclarer un bouton avec la syntaxe JButton bouton= new JButton(« help ») ;
Pour ajouter ce boutons sur le panel ,j'ai suivi le modèle de entryfield :
panel.add(bouton, BorderLayout.WEST) ;
Il a également fallu ajouter bouton.addActionListener(this)
Il a ensuite fallu modifier ActionPerformed en ajoutant un test sur le ActionEvent pE.

```
if(pE.getActionCommand().equals(« help »))
{engine.processCommand(« help »);}
```

7.19.2

Choix des images pour le projet.

7.20

Dans cet exercice, il faut ajouter des items. Création de la classe Item avec les attributs aWeight, aName et aDesc pour le poids, le nom et la description de chaque item. Création de getters pour chaque attributs.

7.21

Dans cet exercice , il faut produire la description d'un item. La String avec la description est produite par la classe item, et est demandée pour chaque item lors de la description d'une Room.

7.22

Dans cet exercice, il faut avoir la possibilité d'intégrer plusieurs objets à une Room,
on utilise une HashMap<String, Item>. J'ai ajouté une méthode dans la classe Room permettant d'ajouter des items à la map depuis la classe GameEngine puisque c'est dans cette classe que les Rooms sont déclarées. Ainsi dans la méthode createRooms() dans GameEngine, en plus de créer les Room, on crée et ajoute les items.

7.22.2

Intégration de tous les items dans le jeu.

7.23

Dans cet exercice, il faut intégrer la commande
J'ai créé un attribut aPreviousRoom dans GameEngine
dans la méthode goRoom(), j'ai mémorisé la CurrentRoom dans aPreviousRoom
Ensuite dans la commande back, lorsqu'elle est appelée, aCurrentRoom prends la valeur de aPreviousRoom, et aPreviousRoom redevient null.

7.26

Dans cet exrcice, il faut pouvoir revenir plusieurs rooms en arrière.on nous propose d'utiliser une Stack. J'ai donc remplacé l'attribut Room PreviousRoom par une stack. Chaque fois que l'on utilise la méthode goRoom(),la aCurrentRoom est enregistrée dans la Stack.
Ensuite dans la méthode back, on utilise la méthode pop() pour récupérer la Room suivante.
Ces deux méthodes utilisent une nouvelle méthode setRoom, cependant goRoom enregistre la Current Room dans la stack, et back prend une room de la stack.

7.26.1

Génération des deux javadocs progdoc et userdoc.

7.28.1

Dans cet exercice, il faut intégrer une commande test qui permet de tester un fichier de commandes. Il faut vérifier si la commande a un second mot. Une fois cette condition vérifiée, j'ai utilisé un Scanner pour pouvoir lire le fichier. On utilise chaque ligne avec processCommand.

Pour finir cette méthode, il fallait traiter le cas où le fichier n'existe pas. La méthode décrite ci-dessus est donc retranscrite dans la partie « try », puis elle
attrape l'exception FileNotFound et affiche « File Not Found » dans ce cas.

7.28.2

Création des deux fichiers de commande.

7.29

Création de la classe Player. J'y ai déplacé toutes les méthodes qui incluent le joueur : back, look, eat, go, … ; les attributs aCurrentRoom, la stack de PreviousRoom, la map d'item, les méthodes additem, removeitem….
Cela a requis beaucoup de modifications, il a fallu par exemple ajouter le Game Engine en attribut du player, mais au final le code est plus propre et plus logique.

7.30

On ajoute les commandes take et drop. J'ai créé dans Player et dans Room des méthodes pour vérifier la présence d'objets, et pour transférer les objets.

Pour take, on vérifie que le second mot correspond a un item de la room, si c'est le cas on l'enlève de la room et on l'ajoute au player.

Pour drop, c'est l'inverse : on vérifie la présence de l'objet dans le player, puis on l'enlève pour le remettre dans la Room.

7.31

Dans cet exercice , il faut permettre au player de porter plusieurs items. J'avais déjà résolu cet exercice : j'ai intégré une HashMap dans le GameEngine a l'exercice 7.22, que j'ai transféré dans player au 7.30.

7.31.1

Dans cet exercice, on crée une classe ItemList qui a pour but de gérer les inventaires des rooms et du joueur.

Les méthodes qui existent en double dans room et dans player sont donc transféré dans cette classe, cela inclut les méthodes de transfert, d'ajout, de vérification.

7.32

Dans cet exercice, il faut limiter le poids que peut porter le joueur.

Un nouvel attribut est créé dans player : le Carry Max.

Quand on utilise la commande take, on vérifie si le poids actuel de l'inventaire + le poids de l'item est inférieur au Carry Max. Il a fallu créer une nouvelle méthode dans ItemList pour renvoyer le poids de l'inventaire.

7.33
Ajout d'une commande inventaire, qui affiche tous les objets de l'inventaire, avec la méthode getItemString() de ItemList.

7.34
Pour cet exercice, il faut ajouter un item mangeable, qui permet d'augmenter le CarryMax.

J'ai décidé d'aller un peu plus loin : j'ai créé une nouvelle classe Food qui hérite d'Item, et qui possède un attribut CarryBonus ; qui définit l'augmentation du CarryMax quand on le mange.

J'ai ensuite modifié la commande eat, pour qu'elle prenne un second mot, vérifie si ce second mot correspond à un item de l'inventaire, et si c'est de la classe Food. Le CarryBonus est alors ajouté au CarryMax.

7.35
Dans cet exercice, il faut intégrer les enums de zuul-with-enums. Le plus complexe dans cet exercice était d'abord de comprendre les enums, puis de les intégrer sans faire de régression.

7.35.1
Dans cet exercice, il faut intéger le switch dans processCommand().

On remplace le grand if(…equals(…)) par un switch(vWord).

vWord étant d'un type énuméré , on peut comparer sa valeur directement avec ==, d'où la pertinence du switch, qui permet de simplifier l'écriture. Par exemple, case : LOOK ; permet de décrire le code a exécuter dans le cas ou vWord == LOOK. Ce code s'arrête au break ;.

default : permet de décrire le cas où aucune des valeurs ne corresponds.

7.41.1
Dans cet exercice, les enums sont modifiés pour intégrer des constructeurs.

7.42
Dans cet exercice, il faut intégrer une limite de temps ou de déplacement. J'ai donc ajouté un attribut au player correspondant au nombre de déplacement restant ; Lors de l'utilisation de la commande Go, cet attribut est décrémenté, et s'il atteint 0, le player est téléporté dans une room de game over.

Ne voulant pas de cette limite de temps dans mon jeu, je l'ai placé à un nombre de déplacement très grand (autour de 100 000) pour ne pas être gêné.

7.42.2
Début de la programmation de l'IHM. Plus de détails dessus dans la partie dédiée du rapport.

7.43
Dans cet exercice, il faut intégrer une porte à sens unique. En lisant les commentaires du forum sur cet exercice et les suivants, j'ai réalisé que la création d'une classe Door était nécessaire.

Chaque pièce possède maintenant une nouvelle HashMap <Door,String> en plus de la HashMap <String, Room>. J'ai ajouté dans back une vérification avec une nouvelle méthode canGoBack(), qui vérifie si les directions proposées par les portes mènent à la room ou l'on veut retourner .

7.44
Dans cet exercice il faut intégrer un beamer. Le beamer est une nouvelle classe qui hérite de Item, et qui possède un attribut aSavedRoom.

Une nouvelle commande charge est créée ; lors de son utilisation, on vérifie si le beamer est dans l'inventaire, puis on remplace la aSavedRoom par la CurrentRoom.

Une autre Commande Fire est créée : elle vérifie la présence du beamer dans l'inventaire, vérifie s'il est chargé, et si c'est le cas, le player et téléporté à l'endroit sauvegardé.

7.45

Dans cet exercice il faut ajouter des portes verrouillées.

Les portes ont déjà été ajoutées dans l'exercice 7.43, ici on les modifie pour ajouter deux paramètres : un Item, la clé, et un boolean , qui décrit si la porte est vérouillée ou non.

On crée alors un deuxieme constructeur, identique au premier, mais qui prends en paramètre un Item en plus. Ce constructeur initialise alors la porte comme vérouillée, avec l'item qui sert de clé.

L'autre constructeur crée une porte dévérouillée.

On modifie ensuite goRoom pour vérifier si la porte que l'on veut passer est dévérouillée, ou si l'on en possède la clé.

7.46

Dans cet exercice, il faut créer une Room qui renvoie dans une room aléatoire du jeu. Pour cela j'ai créé deux nouvelles classes : TransporterRoom, qui hérite de Room, et RoomRandomizer.

TransporterRoom ne possède qu'une seule porte, mais une multitude de sorties. On Override le getExit pour faire appel au RoomRandomizer, qui renvoie une sortie aléatoire parmis toutes les sorties de la transporterRoom.

RoomRandomizer est une classe qui ne possède que des méthodes static : la plus importante, getRandomRoom accepte une HashMap en paramètre, la convertit en tableau, génère un nombre aléatoire compris entre 0 et le nombre d'élément du tableau, et renvoie l'élément correspondant du tableau.

7.46.1

Dans cet exercice, il faut ajouter une commande alea, qui permet de forcer la transporter room à renvoyer une room spécifique. Pour cela j'ai créé un attribut static dans RoomRandomizer, un entier.

Si cet entier a une valeur différente de null, alors RoomRandomizer renvoie forcément la Room correspondant à cet entier du tableau.

Cet entier est initialisé avec le second mot de la commande, et si la commande est réutilisée sans second mot, l'entier redevient null.

Ce procédé permet beaucoup d'erreur, cependant cette commande n'a pas pour but d'être utilisée en dehors des tests, donc en théorie l'utilisateur ne devrait pas s'en servir.

7.47

Dans cet exercice, il faut intégrer l'organisation de zuul-even-better avec les abstract command.

Cet exercice est très long, mais très simple : en dehors des modifications à faire aux enums, et au parser , il suffit juste de déplacer le code depuis Player et GameEngine dans de nouvelles classes, et de changer les accesseurs utilisés.

7.47.1

Dans cet exercice il faut utiliser des packages dans notre jeu.

J'ai séparé mon jeu en 3 packages : le Core, qui contient le GameEngine, le UserInterface, Room, ItemList et Parser.

Le package Command : qui contient toutes les commandes, la classe CommandWord, l'enum CommandWord et la classe abstraite command.

Le package GameElement : qui contient Player, Item, Food, Beamer, Door, Transporter Room.

7.48

Dans cet exercice, il faut ajouter des Characters. Les characters sont très similaires aux premières versions d'Items, donc le concept et une partie du code a été repris : les characters possèdent des attributs nom et dialogue, les rooms possèdent une nouvelle HashMap<String Characters> pour contenir les characters, une nouvelle commande Talk est créée pour afficher le dialogue.

7.49

Dans cet exercice il faut faire en sorte que certains characters puissent se déplacer.

J'ai revu complètement la conception de Characters et de Player :

Characters devient maintenant NonPlayerCharacters ; cette classe et player héritent d'une classe Characters : Toutes les méthodes relatives au mouvement de player, ainsi que la Stack qui enregistrait le trajet sont transférées dans Characters une méthode setTrajet est créé dans Characters, pour pouvoir donner une stack déjà remplie en tant que trajet.

Dans NonPlayerCharacters une nouvelle méthode et un nouvel attribut sont créés : un attribut aBackTrajet, dont l'utilitée est expliquée après ; et une méthode getNextMove : cette méthode renvoie la prochaine room ou se trouvera le NPC, pour cela il utilise le Trajet, qu'il vide dans BackTrajet, puis quand Trajet est vide, Trajet et BackTrajet sont échangés.

De cette façon, on peut initialiser le déplacement d'un NPC en initialisant sa stack avec setTrajet, il parcourra ce Trajet à l'endroit et à l'envers indéfiniment. Cela ouvre la possibilité de créer un NPC qui imite les déplacements du joueur par exemple.

7.53

Création de la méthode main dans la classe Game.

7.54

Exécution du jeu avec le terminal de Linux et de Windows.

7.58

Création d'un fichier .jar exécutable. Il faut modifier toutes les URL des images.

7.60.2

Intégration Complète de l'IHM : de nombreuses fonctionnalités des exercices décrit précédemment ont été totalement ré imaginés. Plus de détails dans la partie dédiée à l'IHM.

60.3 et 60.4

Avec l'ajout de l'IHM, de nombreuses méthodes ont été ajoutées, et pas forcément documentées. Je ne pense pas avoir le temps de toutes les documenter. Cependant les plus importantes ont été commentées.

63

Les scénarios de test sont créés, cependant depuis le fichier .jar il semble impossible d'y accéder.

63.2 et 63.3

Le scénario du jeu est finalisé et intégré, il est possible de gagner et perdre selon les conditions décrites au début du rapport.

L'IHM

J'ai voulu pour mon jeu créer une IHM vraiment développée. L'idée de base était d'avoir un jeu en vue du dessus contrôlé au clavier, sans avoir à taper de commandes, ou l'on pourrait déplacer un personnage selon une grille, un peu comme dans certains RPG japonais et autres Pokémon.

Le développement de l'IHM a donc commencé par cela : afficher un carré, et réussir à le déplacer sur une grille définie. J'ai beaucoup expérimenté avec des composants de Swing avant de réussir à obtenir quelque chose de correct.

Pour faire le lien avec mon jeu, j'ai créé trois nouvelles classes :la classe Entity , qui va correspondre aux éléments représentés sur le joueur, la classe Grid, qui permet le positionnement des Entity dans la room , et permet de s'assurer qu'il n'y ait pas 2 Entity au même endroit, et la classe Coordinates, qui permet de positionner les Entity sur la Grid.

Les classes Character, Item, Door, ainsi qu'une nouvelle classe Terrain héritent de la classe Entity, afin de pouvoir être affichés, et interagir entre eux.

Dans le but de permettre l'interaction entre ces différents éléments, une méthode abstraite interact a été créée dans Entity, et définit l'action qui va se produire quand le joueur utilisera une de ces entitées.

-Pour le Terrain, il ne se passe rien : c'est le but du terrain ; servir juste de décor et bloquer les déplacements du joueur.

- Pour le NPC : la commande talk <nom du npc> est exécutée, ce qui affiche le dialogue du NPC.

-Pour les Item : la commande take <nom de l'item> est exécutée.

-Pour les Doors : interagir avec la porte déplace le joueur à la destination de la porte, qui peut se trouver à un autre endroit de la même Room ou dans une autre Room.

Afin de pouvoir interagir avec les Entity devant lui, le Player doit avoir une direction dans laquelle il regarde, c'est pour cela que j'ai créé la classe enum Direction, qui décrit les quatre directions auxquelles le joueur peut faire face, et dans lesquelles il peut se déplacer. Cela permet de simplifier grandement toutes les méthodes de déplacement, et d'affichage.

L'attribut Trajet de Characters, qui permettait au Player d'enregistrer ses déplacements, et au NPC de savoir où aller, n'utilise plus de Room ; il utilise à la place des Directions.

Le changement de Room passant maintenant par les Doors et non plus par les sorties de la Room, il a fallu réinventer complètement la TransporterRoom : Il n'est plus question de TransporterRoom, mais de TransporterDoor, qui envoie le joueur a un des emplacements aléatoires parmi ceux désignés. Les Classes TransporterRoom et RoomRandomizer sont donc désormais inutiles.

Pour l'inventaire, j'ai créé un menu qui peut apparaitre et disparaitre quand on appuie sur la touche I. Ce menu contient tous les objets de l'inventaire sous forme de JRadioButton, ainsi que deux boutons pour faire les commandes use et drop. La commande use est une commande qui regroupe les commandes charge, fire , et eat, qui n'avaient pas d'intérêt à être gardées séparées. Selectionner un item dans l'inventaire et appuyer sur un bouton permet d'executer la commande

<Commande du bouton> <nom de l'objet>

Pour la commande test, j'ai dû ré imaginer la classe : il n'est plus possible de faire juste processCommand sur des lignes tirées d'un fichier.

J'ai donc utilisé la classe Java.awt.Robot, qui permet de prendre le contrôle du clavier et de la souris.

Les actions du robot sont décrites dans les fichiers de tests, qui sont d'abords lu par un Scanner, caractère par caractère, au lieu de ligne par ligne.

Mode d'emploi

Les touches ZQSD du clavier permettent de se déplacer, la touche E permet d'interagir, F permet d'ouvrir ou fermer la boite de dialogue, et I permet d'ouvrir ou fermer l'inventaire.

Déclaration anti plagiat

L'intégralité du code, à l'exception des parties fournies dans les versions de zuul, ont été rédigé par moi-même.

J'ai créé toutes les images des Rooms.

Toutes les autres images utilisées sont libres de droits, de créateurs anonymes, et réutilisables à des fins commerciales.