Le mystère des ruines Sheikah

A3P Java 2025/2026 G2

Description du jeu

Auteur

Benoît de Keyn

Thème

Dans des ruines anciennes, un archéologue doit trouver un artefact du peuple Sheikah.

Résumé du scénario

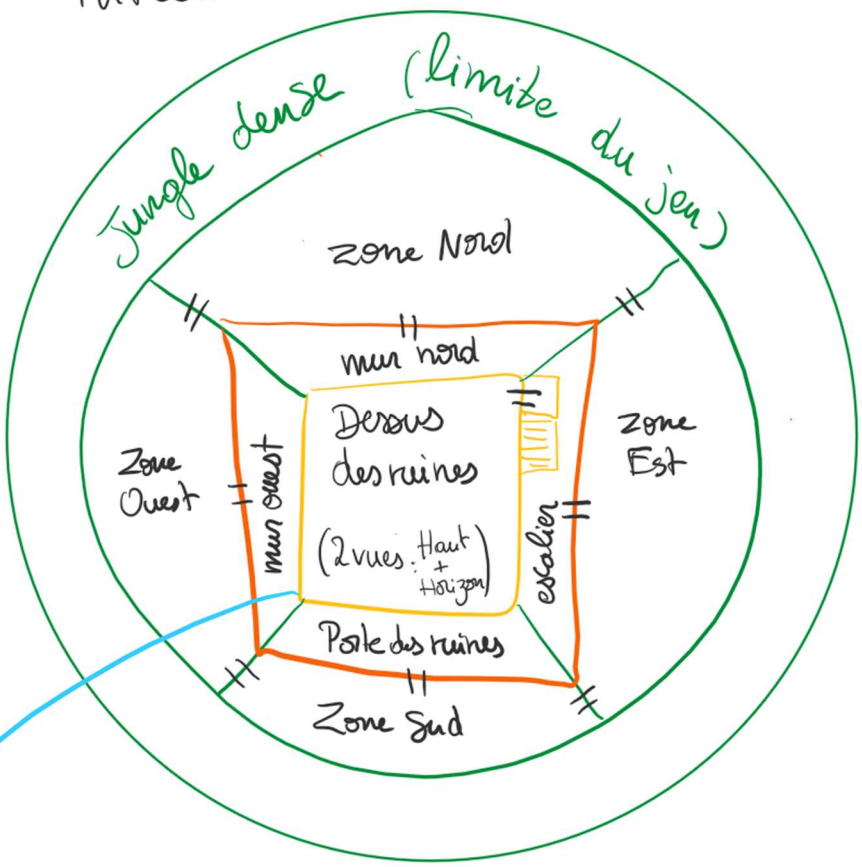
Un archéologue Sheikah découvre un manuscrit vieux de 5000 ans mentionnant un artefact capable de transformer la chaleur en mouvement. Il part à sa recherche dans la jungle Korogu, où une ruine souterraine l'attend.

Plan

Version réduite

La version réduite comprend les « rooms » du tout premier croquis ci-dessous :
- 4 zones (nord, est, sud, ouest)
- 4 murs (mur nord, escaliers, porte, mur ouest)
- la surface des ruines en version immersive
  (dans la version complète, une vue de haut sera accessible via l'arbre)

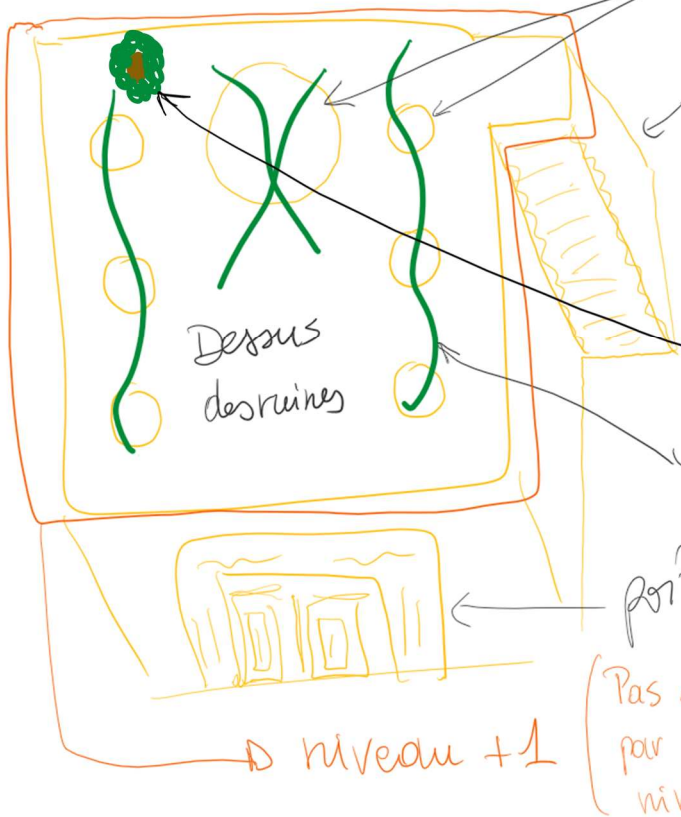
niveau 0 (Ground)



détails

= : passage possible

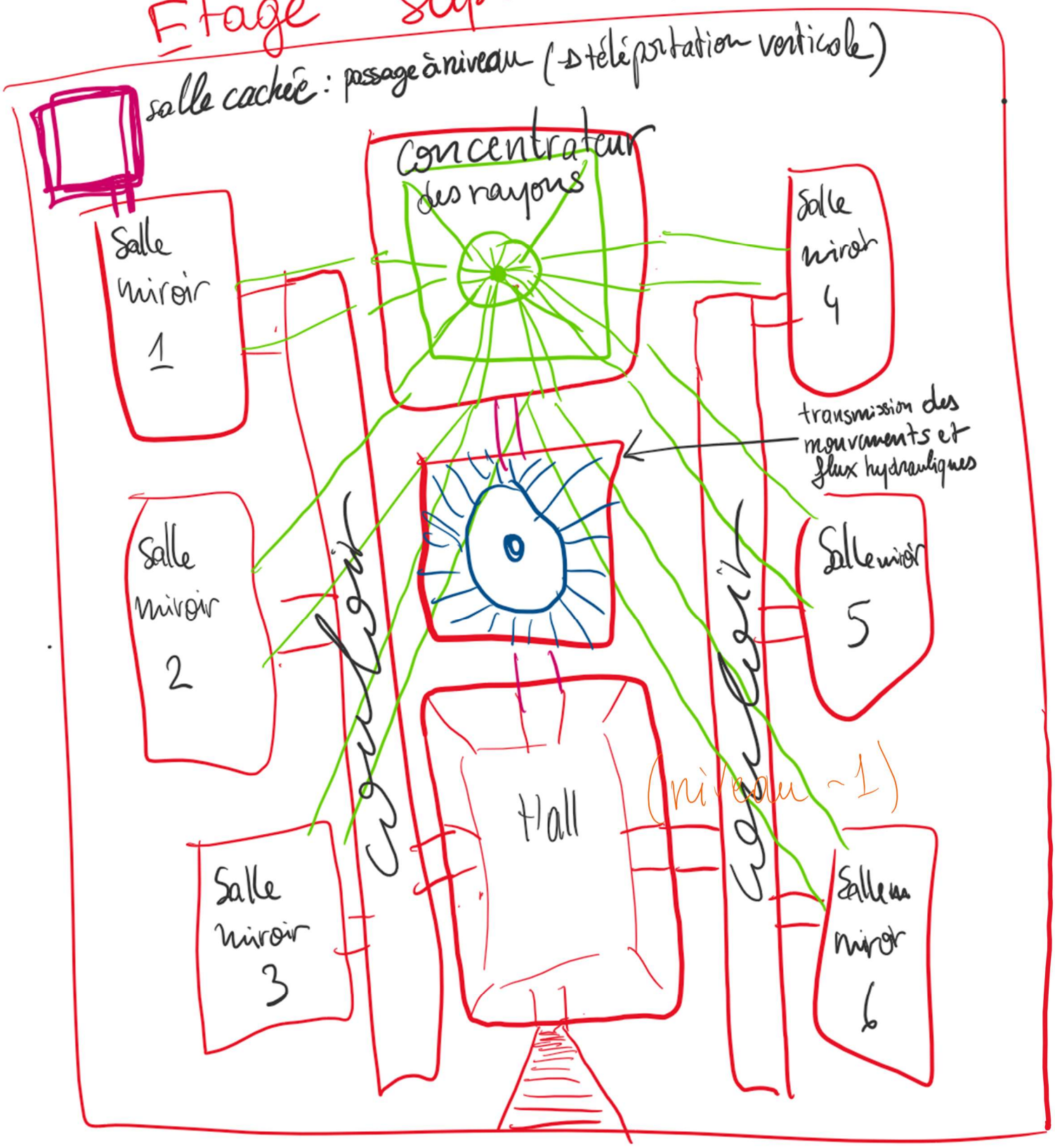
Vue 3D de la surface des ruines



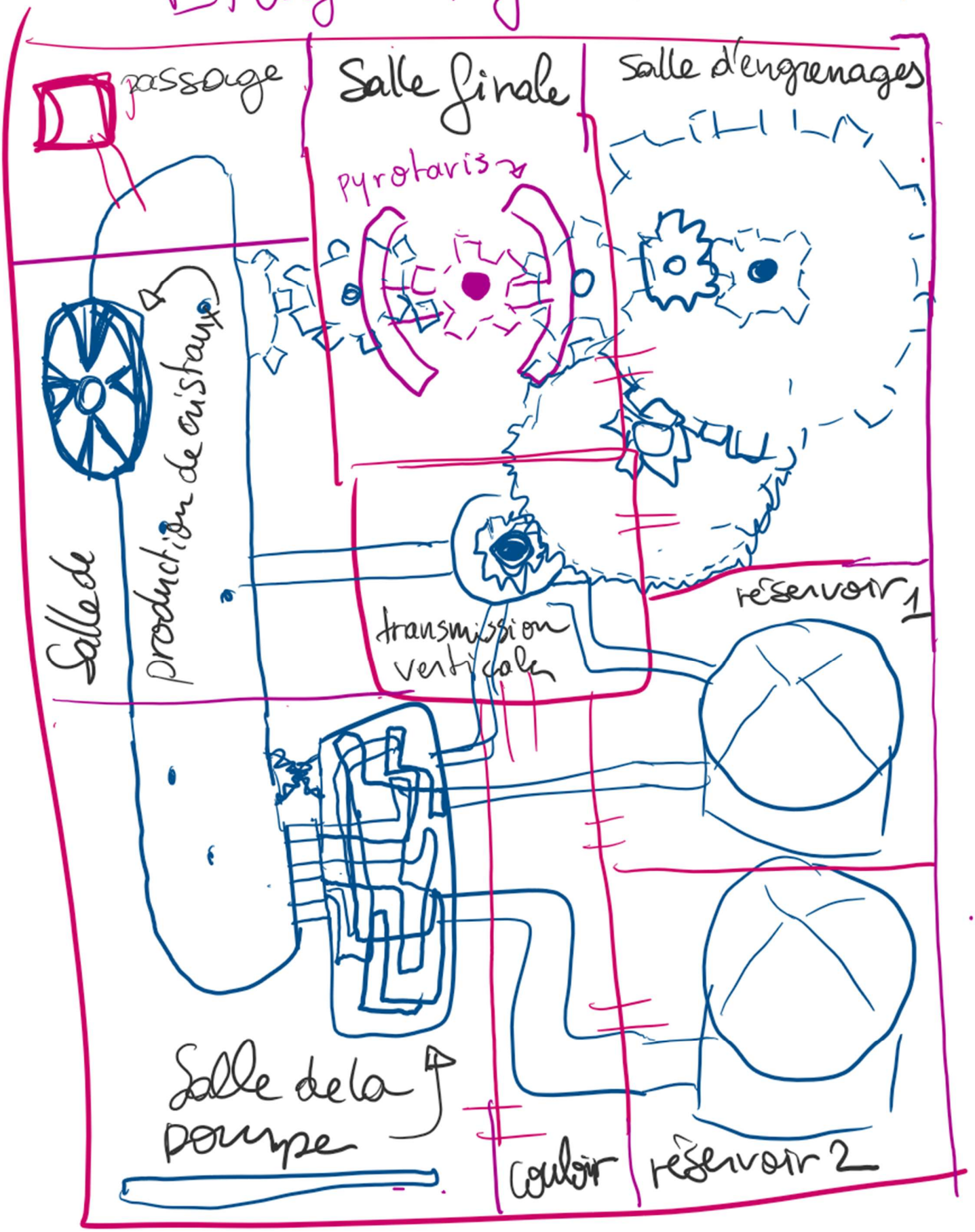
vue de Haut

niveau +1 (Pas de différence par rapport au niveau 0)

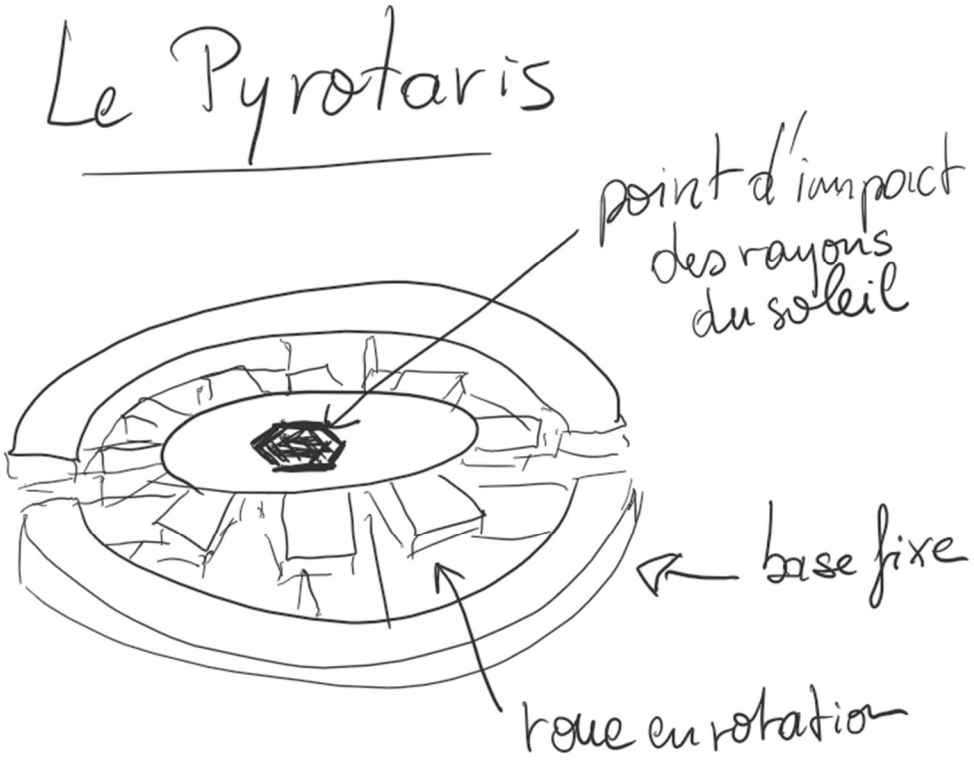
Étage supérieur



Étage inférieur (niveau -2)



□ → 1 room



Scénario détaillé

Un archéologue expert dans la civilisation Sheikah parvient à déchiffrer un vieux manuscrit datant de 5000 ans qui mentionne une pièce exceptionnelle de technologie qui permettrait de convertir la chaleur en mouvement. Cela expliquerait comment ils sont parvenus à réaliser des monuments titanesques il y a 5000 ans.

Il décide de se rendre dans ce qui semble être la jungle Korogu pour découvrir cet artefact.

Arrivé devant une ruine souterraine Sheikah au milieu de la forêt équatoriale, il doit trouver l'artefact. L'archéologue commence son aventure dans la zone sud, devant les ruines.

Attention cependant, dans ce profond labyrinthe de pierre, l'atmosphère millénaire n'est plus respirable.

Détail des lieux, items, personnages (non exhaustif)

- Le pyrotaris : convertit la chaleur en rotation
- La ruine émerge du sol. Elle possède 2 étages. Une porte et des puits solaires à sa surface.
- 6 pièces à l'étage supérieur permettent de capter les rayons du soleil grâce à des puits percé au plafond, tapissés de miroirs qui y débouchent.
- 1 pièce est un hall où on débouche en arrivé dans les ruines
- 2 couloirs pour accéder aux différentes pièces
- 1 pièce sert de concentrateur de lumière.
- 1 pièce sert à faire communiquer le mouvement et les flux hydrauliques
- L'archéologue est expert en traduction Sheikah
- Une clé
- Des réservoir de fluide pour des circuits hydrauliques

Situations gagnantes et perdantes

- Situation stagnante :
  L'archéologue cherche le pyrotaris dans les ruines ou reste dehors sans rien faire.
- Situation perdante :
  L'archéologue manque d'oxygène dans les ruines sans avoir trouver le pyrotaris
- Situation gagnante :
  L'archéologue trouve le pyrotaris sans manquer d'oxygène

Énigmes, mini-jeux, combats

- Beaucoup de casse-têtes logiques
- Chercher le couteau pour dégager les lianes du dessus des ruines pour laisser passer la lumière du soleil afin d'ouvrir la porte.
- Trouver la pièce qui permet de passer à l'étage inférieur
- Trouver la clé qui permet d'ouvrir les grilles de la salle finale
- Réparer le miroir cassé pour activer la pleine puissance
- Respirer des bouffées d'oxygènes assez régulièrement : trouver le pyrotaris avant de manquer d'oxygène
- Décrypter des inscriptions Sheikah pour des indices.
- Tuer des serpents venimeux.
- Reconnecter des circuits hydrauliques pour débloquer des mécanismes.

Commentaires

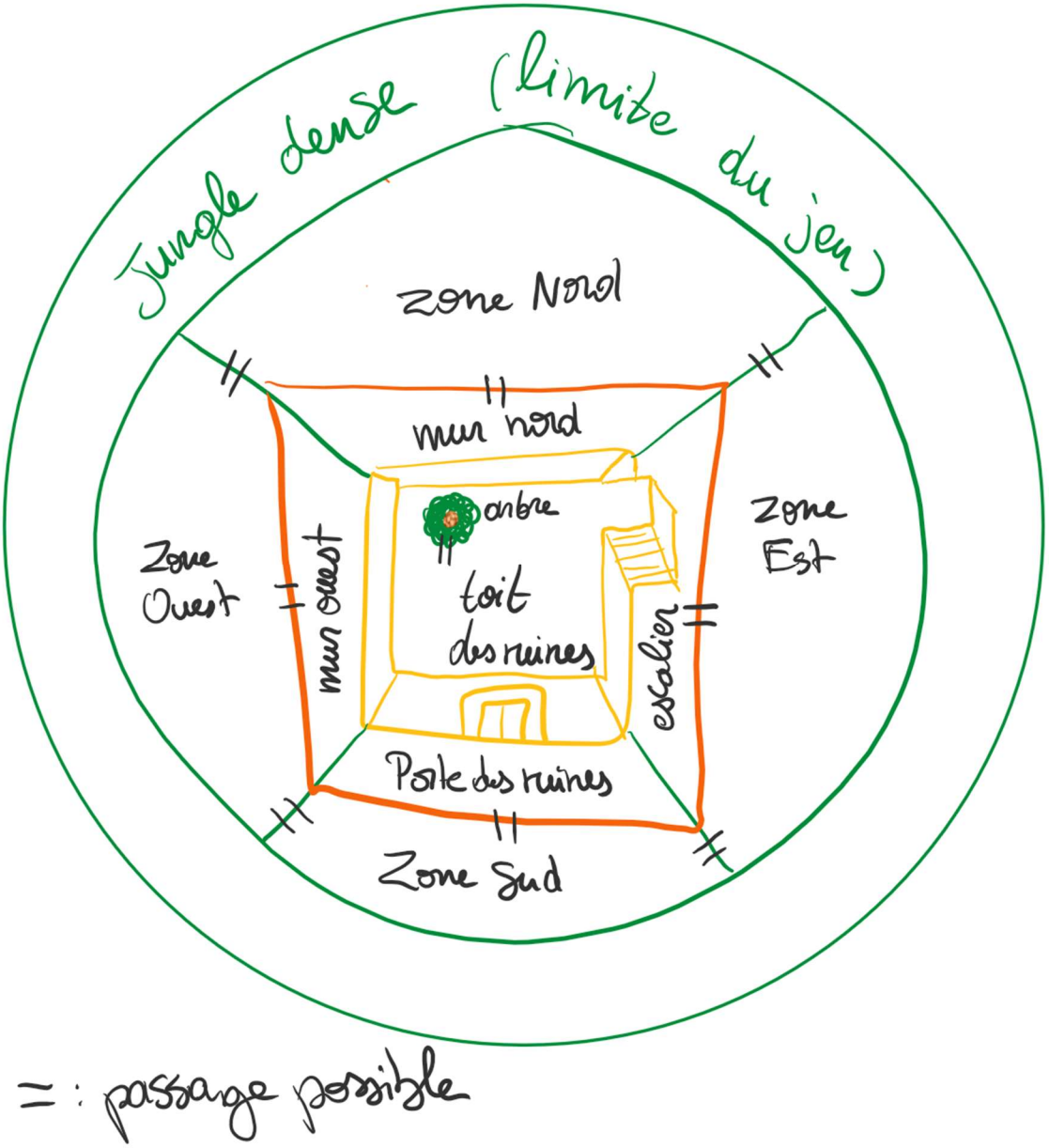
- Le but final est encore simple, j'aimerais trouver quelque-chose de plus concret que de juste trouver le pyrotaris. Le plan peut encore changer, les machines sont encore sans réelle utilité dans les ruines, il faut leur trouver un sens. Une fonction à cette ruine (fabriquer des cristaux peut être pour donner une récompense, d'où la salle de production).

Réponses aux exercices

Exercice 7.3.3

Le scénario réduit correspond à :

Plan Réduit



L'arbre est ajouté après l'implémentation des directions haut et bas

Scénario réduit

Le but est d'entrer dans les ruines par la porte du mur SUD des ruines.
La solution est de monter à l'arbre, prendre la clé qui s'y trouve, puis revenir à la porte et ouvrir la porte grâce à la clé.
Pour le moment on fera comme si l'archéologue avait besoin d'oxygène pour respirer dans la jungle.

Exercice 7.4

J'ai pris la version réduite du scénario pour implémenter mes rooms.
J'ai créé autant de variables de type Room que de zone de mon croquis, puis j'ai assigné les sorties nord, est, sud, est selon la direction du passage et le sens des pièces.

Aussi, j'ai traduit les commandes quit, help et go, ainsi que les textes affichés. J'ai adapté les textes d'introduction et de la commande 'aide' à mon jeu. Aussi j'ai personnalisé un peu les messages d'erreur pour plus de clarté.

Exercice 7.5

J'avais déjà empêché cette duplication de code lors du TP, mais uniquement pour afficher les sorties.
J'ai donc renommé la fonction, et ajouté la description de la room actuelle.
J'ai personnalisé un peu la fonction pour un affichage plus clair avec des flèches et des retours à la ligne.
J'ai aussi changé les descriptions de mes lieux pour que la phrase « Vous êtes … » soit cohérente avec la suite.

Exercice 7.6

J'ai mis en privé tous les attributs de la classe Room, puis créé une fonction accesseur getExit() qui prend en paramètre la chaîne de caractère qui correspond à la direction pDirection.

Au lieu de faire une suite de « if else » avec des equals, j'ai préféré un switch case sur le paramètre pDirection (car le switch case prend en charge les égalités de Strings nativement). Si la direction demandée n'est aucun des 4 points cardinaux, elle retourne la référence null.

Dans la classe game, dans goRoom, j'ai aussi préféré un switch case sur le 2nd mot de la commande (à savoir la direction).
Si le second mot ne correspond à aucun point cardinal, il y a le message d'erreur « Direction inconnue ! » Autrement, la variable vNextRoom stocke la référence renvoyée par getExit().

Ensuite, on vérifie si cette référence est null. Le cas échéant, on affiche le message d'erreur « Vous ne pouvez pas aller dans cette direction ! »

J'ai peut-être omis une situation d'erreur, mais je n'ai pas l'impression d'avoir eu besoin d'une des 3 solutions proposée pour prendre en charge une mauvaise direction demandée…

Exercice 7.7

Pour cet exercice j'ai créé la fonction getExitString() qui créée une variable contenant le début du message : « Les directions possibles sont : » .
Puis, avec une suite de 4 conditions indépendantes pour chacun des 4 points cardinaux, elle concatène (ou non) la direction formattée dans un format plus clair à lire pour le joueur.
La String est ensuite retournée.

Cette fonction est appelée dans printLocationInfo() juste après la description.

Il est préférable que les sorties disponibles soient lues dans la classe Room pour mieux garder les fonctions qui utilisent les attributs d'un objet Room uniquement dans la classe Room.

Exercice 7.8

Dans la classe Room j'ai :

- Ajouté un attribut exits, une hashmap
- Ajouté la ligne du constructeur pour initialiser exits.
- Changé setExit() comme dans le livre, avec exits.put(pDirection, pNeighbor)
- Changé getExit() en renvoyant exits.get(pDirection) après avoir vérifié si la clé existe, sinon, null.
- Changé getExitString() en utilisant un for-each qui itère sur la liste des directions exits.keySet()

En revanche je n'ai pas eu besoin de la ligne import java.util.set comme proposé dans l'aide (j'ai peut-être pris une version trop récente de BlueJ, ou la version de Java inclue 'set' dans 'util')

Dans la classe Game j'ai :

- Changé dans la méthode goRoom, l'assignation de vNextRoom en la réduisant à une simple ligne qui appelle getExit(pDirection)
- Modifié l'appel de setExit() pour chaque connexion entre 2 pièces pour l'adapter à la nouvelle méthode, comme dans le livre.

Maintenant, on ne peut effectivement plus faire la différence entre une direction bloquée ou une direction non reconnue, car tous les noms de direction peuvent être ajouté en tant que clé dans la hashmap.

Pour résoudre ce problème, j'ai créé une liste des 6 directions possibles dans mon jeu, et goRoom vérifie si la direction demandée appartient à ce set pour spécifier le message d'erreur.
J'ai repris ce qui est fait dans la classe commandWords. J'y ai rajouté la liste des 6 directions pour vérifier si la direction souhaitée existe, avant d'aller appeler aCurrentRoom.getExit dans goRoom.

Pour cela j'ai modifié les dernières lignes de getCommand() de la classe Parser, en rajoutant des tests sur vWord2 (si elle vaut null et si aValidCommands.isDirection(vWord2) est vrai) en fonction, le second word pour la commande go est soit : une direction valide, soit "invalid", soit null). Ces 3 valeurs sont vérifiées dans goRoom pour afficher le bon message d'erreur.

Exercice 7.8.1

J'ai ajouté un arbre sur le toit des ruines, avec les 2 directions « haut» et « bas» entre l'arbre et le toit.
Aussi j'ai remplacé les directions est/ouest pour passer de l'escalier au toit par bas/haut.

Exercice 7.9

Ah mince, j'avais déjà utilisé une boucle for each pour getExitString. J'utilisais déjà ce type de boucle l'année dernière, surtout en javascript avec for (value of array) { … }.
Et la concaténation de String avec += aussi m'a paru naturelle.

Ma fonction n'a donc pas changé ici, par contre elle est différente du livre dans le sens où je n'ai pas de variable intermédiaire qui stocke la liste des directions possible mais directement :
for (String vDirection : this.exits.keySet()) { … }
Mais je comprends bien la différence entre un set (Set<Type>) et un tableau Type[], et que la méthode keySet() renvoie un set et non un tableau.

Exercice 7.10

On commence par créer une chaîne de caractères initiale contenant le texte d'introduction : « Les directions possibles sont : ».
Cette String est la base pour construire le message final.
La méthode ne reçoit aucun paramètre.

Sur la même ligne, on déclare un Set (collection d'éléments uniques) de type String et on lui assigne le résultat de la méthode .keySet() appliquée à la hashmap exits qui contient les différentes associations { clé String -> référence d'une Room } qui correspondent aux différentes directions possibles (les clés) associées aux différentes Rooms de sorties.

Une boucle for each répète le code entre crochets autant de fois qu'il y a de clés (donc de directions) trouvées dans la hashmap exits de la room désignée par this.
A chaque fois que la ligne entre crochet s'exécute, la variable vDirection prend pour valeur une des clés de a hashmap.

La ligne ainsi répétée concatène à la string de base cette direction pour pouvoir finalement retourner à la fin de la méthode, toutes les directions possibles .

Exercice 7.10.1

J'ai écrit plein de commentaires javadoc pour expliquer le but de chaque méthode, ses paramètres et ses sorties. Notamment avec les mots clé comme @param et @return
J'ai traduit en français aussi les commentaires de Parser, Command et Command Words pour la cohérence.
Je me suis ajouté en co-auteur pour les classes Parser et Command
J'ai changé le nom du jeu.

Exercice 7.10.2

J'ai résolu des erreurs de génération dans BlueJ en allant dans chaque classe puis en affichant la vue Interface. Alors les classe devenaient hachurée pour être recompilée. Cela corrigeait des erreurs que je ne connais toujours pas, mais j'ai l'impression que ça n'a rien changé à mes commentaires.

Bien sûr, la classe game est comme une fonction main, qui sert à avoir une game loop et à centraliser les appels des différentes fonctions définies ailleurs, afin de bien découpler les classes, et de pouvoir modifier du code sans avoir à changer des appels sur toutes les couches.

Exercice 7.11

J'ai ajouté à la classe Room la méthode getLongDescription qui renvoie la String que printLocationInfo affichait, et j'ai réduis cette dernière dans game, à simplement afficher le résultat de cette nouvelle méthode.

Exercice 7.14

J'ai renommé printLocationInfo par look dans la classe Game et renommé les 2 appels à cette fonction.
J'y ai aussi ajouté une clause dans le switch case de processCommand qui renvoie vers look si la commande est « regarder »

Dans CommandWords, j'ai eu une erreur parce que j'ai rajouté un String « regarder » au tableau des commandes valides, car il était encore écrit comme dans le TP.
J'ai donc changé la déclaration des attributs conctant en leur assignant directement la liste de valeurs comme ils font dans le livre, et aussi supprimé le contenu du constructeur de CommandWords.

Exercice 7.15

J'ai rajouté la méthode breathe() qui affiche "Vous venez de consommer une bouffée d'oxygène de votre réserve."
J'ai ajouté la String « respirer » à la liste des commandes, et comme pour « regarder », une clause au switch case de processCommand.

Exercice 7.16

J'ai ajouté

- La méthode showAll à CommandWords qui un peu de la même manière que pour getExitString, utilise une boucle for each mais cette fois sur un tableau de Strings, pour afficher la liste des commandes valides.
- La méthode showCommands à Parser qui se contente d'appeler showAll.
- L'appel à showAll dans la méthode printHelp de la classe Game.

Exercice 7.17

Oui, il faut changer Game à chaque fois qu'on ajoute une nouvelle commande valide.
Car maintenant que cette commande est disponible, il faut lui assigner une action à exécuter quand le joueur l'utilise dans processCommand.

Exercice 7.18

J'ai échangé la méthode showAll de CommandsWord par par une méthode dont la signature est :

```
public String getValidCommandsString()
```

et qui renvoie la String qui contient la liste des commandes :
Elle crée une String vide de base, puis avec une boucle for each qui itère sur le tableau des commandes valides, elle concatène chaque commande valide à cette base avec +=, avant de retourner la String ainsi complétée.

Puis dans parser j'ai échangé showCommands par getCommandsList() qui transmet simplement le résultat précédent par un appel de la méthode getValidCommandsString sur l'objet CommandWords.

Finalement dans print help, j'ai mis la ligne :

```
s.o.p(aParser.getCommandsList())
```

au moment d'afficher la liste des commandes.

Exercice 7.18.1

La version zuul-better téléchargée ne compilais pas.
J'ai corrigé la mise en commentaires des lignes de la classe Game qui concernaient les images.
Mais après ça j'ai toujours une erreur de compilation dans Game à l'appel de parser.getCommand(), car dans la classe parser, cette méthode attendait le paramètre pInputLine, alors j'ai modifié cette classe en ajoutant l'import de la bibliothèque Scanner, et j'ai copié ce que nous avions fait dans notre jeu pour que getCommand() ne lise plus le paramètre mais une variable locale vInputLine correspondant à la dernière ligne entrée dans l'invite de commande.
Ensuite j'ai aussi corrigé l'erreur de compilation dans Game() car printHelp appelait showCommands alors que maintenant le parser avait une méthode qui retournait la String contenant la liste des commandes.

Je crois que c'était inutile que zuul-better s'exécute bien parce que ça ne m'a pas empêché de comparer et de voir les différences suivantes :

- Dans la classe CommandWords : outre le fait que j'ai appelée ma fonction qui liste les commandes getValidCommandsString, la première différence est l'utilisation d'une boucle for au lieu de for-each (ce que vous avez dit ne pas changer dans la FAQ). La deuxième différence c'est que j'ai déclaré/assigné un String vide, que j'ai concaténé dans la boucle for-each avec l'opérateur +=, tandis que dans zuul-better, ils utilisent respectivement stringbuilder et .append().
- Dans zuul-better, s'il n'y a pas de sorties dans une direction demandée, la command getExit renvoie null et le message d'erreur est toujours « there is no door ». Tandis que j'ai différencié l'absence de sortie et l'invalidité d'une direction dans mes messages d'erreur en ajoutant une liste des directions valides dans CommandWords et les méthodes correspondantes.

Autrement j'ai lu dans la FAQ que vous préferiez avoir dans la classe Game :

- Une fonction printLocationInfo qui appelle getLongDescription sur aCurrentRoom
- Une fonction look qui se content d'appeler printLocationInfo

Alors que j'avais remplacé printLocationInfo et ses appels par l'unique fonction look().
J'ai donc suivi ce que vous indiquez dans la FAQ en appelant dans Game : soit look() (dans processCommand) soit printLocationInfo (dans printWelcome et goRoom) pour afficher la description de la pièce et les directions disponibles.

Autrement, les autres différences me paraissent négligeables.

Exercice 7.18.2

L'avantage d'utiliser StringBuilder au lieu de simplement manipuler des String, se fait ressentir lorsqu'il faut beaucoup éditer le contenu d'une chaîne de caractères.
En fait si j'ai bien compris, quand on additionne 2 strings, cela créée une nouvelle String qui vaut la somme des 2 précédente, mais gardes les 3 en mémoire. Il y a autant de nouveaux objets de type String qui sont créé à chaque itération de la boucle for-each dans getExitString ou getValidCommandsString. Le type String est dit immutable.
Alors qu'en utilisant Stringbuilder, la variable qui contient la chaîne de caractère devient mutable. Donc on économise de la mémoire, et on garde la même référence au passage.

J'ai donc changé getExitString et getValidCommandsString pour qu'elle utilise StringBuilder comme on peut le voir dans zuul-better : Initialisation (vide ou non), puis des .append(...) qui remplacent += ..., et finalement l'utilisation de .toString() pour générer un objet de type String.

Exercice 7.18.3

10 images créées avec PowerPoint en .png 800x600.
Je les aies mises dans un dossier « images » dans le dossier BlueJ du projet.

Exercice 7.18.4

Le titre du jeu : *Le mystère des ruines Sheikah*

Exercice 7.18.6

En suivant l'énumération au point 2 :

a) Dans **Command** et **CommandWord** : Pas de changement

b) Dans **Room** : j'ai ajouté l'attribut « aImageName », la définition de cet attribut dans le constructeur ainsi que la méthode getImageName().

c) Dans **Parser** :
Au lieu de lire la ligne du terminal dans la fonction getCommand(), on la prend en paramètre. Nous n'avons donc plus besoin du Scanner ni des lignes qui servaient à acquérir ce qui est maintenant donné en paramètre : la ligne du terminal.
Aussi pour pouvoir extraire les 2 mots de commande de cette ligne, on n'utilise plus les méthodes de la bibliothèque Scanner mais celles de StringTokenizer. (Avec un changement de la bibliothèque importée au début de la classe).

d) Dans **Game** : Après avoir fini de transvaser son contenu la classe GameEngine, j'ai copié celle de zuul-with-images : Création du moteur et de l'interface puis liaison des 2.
Aussi, j'ai ajouté la spécification du dossier contenant les images.
(C'est plus propre dans un dossier)

e) Dans **GameEngine** :
- J'ai transformé tous les « System.out » par « this.aGui » après avoir ajouté le nouvel attribut.
- J'ai ajouté le chemin relatif de l'image pour chaque room lors de sa création
- j'ai ajouté l'affichage d'image dans différentes fonctions (goRoom, printWelcome, ...)
- j'ai incorporé le changement de processCommand pour lire la String de la ligne et non plus une commande directement. Notamment en supprimant le retour d'un boolen à cette fonction interpretCommand() malgré la contradiction des commentaires présents dans zuul-with-images.
- j'ai édité ma fonction « quit » pour qu'elle utilise la méthode enable() de la GUI.
- j'ai supprimé la fonction play()

f) Comme la fonction printHelp() et look() utilisent encore la fonction printLocationInfo() sans avoir à afficher une nouvelle image, j'ai préféré ajouter la fonction displayLocationImage() qui affiche l'image à condition qu'elle existe. Je l'appelle juste après printLocationInfo() quand nécessaire.

g) Dans **UserInterface** :
- J'ai ajouté l'attribut « aImageFolder » et son assignation dans le constructeur, ainsi qu'une méthode pour changer sa valeur.
Dans la fonction « showImage » j'ai ajouté la prise en compte du dossier pur le chemin des images (le travail avait été commencé par l'auteur, les guillemets étaient simplement vides).
- Dans cette même fonction, j'ai également changé l'échelle des images avec la classe « Image » de java.awt pour qu'elles soient moins grandes, et comme ça j'ai pu aussi agrandir la hauteur de la zone de texte ailleurs dans la classe UserInterface.
- J'ai remplacé les importations de listes de classes par chaque classe utile unique.

Exercice 7.18.8

- Dans UserInterface : j'ai ajouté l'importation de la classe javax.swing.JButton
- Dans le constructeur j'ai ajouté le bouton comme attribut

- Dans createGui : j'ai ajouté la création du bouton qui servira donc à respirer, et je lui ai ajouté un Action Listener.
- J'ai ajouté ce bouton dans la zone Est du panel.
- Dans actionPerformed : je vérifie d'abord si l'objet qui a déclenché l'action est le bouton, auquel cas on fait comme si le programme avait reçu le texte « respirer » pour déclencher la commande associée.
  Autrement, on appelle processCommand() qui lira l'entrée du terminal.
- J'ai fait la même chose pour le bouton « regarder » dans la zone ouest.

Exercice 7.19.2

Déjà réalisé. J'ai simplement renommé le dossier avec une majuscule.
J'ai également déplacé la définition du dossier contenant les images en tant qu'attribut de GameEngine, et assigné dans setGUI().

Exercice 7.20

- J'ai créé une nouvelle classe **Item** avec
  - 2 attributs : le poids et la description
  - leur constructeur et accesseurs
  - une fonction getLongDescription qui renvoie la description complète de l'item comme pour les rooms.
- Dans la classe **Room** j'ai ajouté :
  - un nouvel attribut aItem
  - une fonction pour le définir si voulu.
  - une fonction qui renvoie la string « Auncun objet » ou la decription longue de l'objet s'il existe.
  - dans getLongDescription(), j'ai ajouté la string renvoyée par la fonction précédente.

Exercice 7.21

Déjà réalisé dans l'exercice précédent.
La description de l'item doit être produite par l'item qui va lire ses attributs privés pour créer la String.
La description de la Room est générée dans la classe Room qui lit ses attributs privés (dont l'item)
L'affichage de toutes ces descriptions se fait dans la classe GameEngine qui n'a pas besoin d'accéder aux attributs de l'item ou de la room. On veut découpler les classes au maximum.

Exercice 7.21.1

Dans item, j'ai remplacé la description seule par un nom et une description.
J'ai ajouté 3 objets dans 3 salles.
Je n'ai pas modifié la commande look.

Exercice 7.22

Dans **Room** :
- J'ai remplacé l'attribut aItem par une hashmap d'items ou le nom de l'objet est la clé.
- J'ai changé setItem() par addItem() avec .put
- J'ai changé la fonction getItemString() pour qu'elle liste tous les objets de la hashmap s'il y en a, en

utilisant un Stringbuilder.
Dans **GameEngine**, j'ai remplacé le nom de la fonction pour ajouter un item, et j'ai ajouté une branche en plus de la clé dans l'arbre pour avoir plusieurs items dans une room.

Exercice 7.22.1

J'ai préféré une hashmap pour pouvoir récupérer un item par son nom par exemple si l'utilisateur le désigne par son nom dans l'invite de commande. Sinon il faudrait itérer chaque élément de la collection jusqu'à tomber sur l'objet dont le nom correspond (après y avoir accédé), ce qui serait beaucoup moins optimisé, et ajouterai des lignes de codes.

Exercice 7.22.2

J'avais déjà voulu tester en ajoutant des objets, mais je vais ajouter un objet dans la salle initiale aussi. Disons une buche.

Exercice 7.23

J'ai ajouté un attribut aPreviousRoom au GameEngine, initialisé à null dans createRooms().
Dans goRoom(), avant de changer la valeur de aCurrentRoom, je stocke sa valeur dans aPreviousRoom.

J'ai ajouté le traitement de la commande « retour » au switch case de interpretCommand().
j'ai créé la fonction goBack() qui prend le commande en paramètre. Comme pour quit(), elle vérifie qu'il n'y ait pas de second mot (auquel cas elle prévient le joueur).
Si aPreviousRoom est null, le joeur est informé qu'il ne peut pas revenir en arrière. Autrement, elle remet la salle précédente en tant que courante, avant d'appeler printLocationinfo() et displayLocationImage().
J'ai testé tous les cas possibles de retour et tout fonctionne.

Exercice 7.26

J'ai changé comme indiqué l'attribut précédent par une Stack<Room> déclarée en début de classe puis créée dans le constructeur. J'ai remplacé aux 3 endroits correspondants :
.push() dans goRoom, .empty() pour vérifier si il y a un historique et .pop() pour revenir en arrière.
Tout est parfaitement fonctionnel.

Exercice 7.26.1

J'ai ajouté au PATH le dossier bin de BlueJ, j'ai généré les 2 javadoc (prog et user).
J'ai vu qu'ils attendaient des commentaires pour les attributs alors je les ai ajoutés.
Maintenant il n'y a plus de warnings.

Exercice 7.28.1

Pour implémenter cette nouvelle commande de test, j'ai dû changer plusieurs fonctions au niveau de la lecture des commandes :
Jusqu'à maintenant, dans la classe Parser, la fonction getCommand() en charge d'extraire les 2 mots de la commande, vérifiait si le premier mot appartenait à la liste des commandes valides et si le second mot appartenait à la liste des direction valides (en utilisant les méthodes de la classe CommandWords).

Cela fonctionnait car la seule commande à accepter un second mot était « aller », et mon implémentation (proposée dans le forum sur l'exercice qui traitait du sujet) permettait de renvoyer au joueur 1 des 2 messages suivants en fonction de la situation :
- "Cette direction n'existe pas."
- "Vous ne pouvez pas aller dans cette direction !"

Le problème c'est que maintenant que « test » accepte aussi un second mot je ne pouvais plus me contenter de vérifier si le second mot est valide dans la fonction de Parser qui transforme l'entrée utilisateur en objet commande.
J'ai pallié le problème tout en gardant 2 messages d'erreur différents de la sorte :

- Un peu comme pour la fonction de Parser getCommandsList() qui renvoie la liste des commandes disponibles en appelant une fonction de CommandWords qui crée la String originale : j'ai créé une fonction isDirection() dans Parser qui se contente de renvoyer le résultat de la fonction isDirection() appelée cette fois sur un objet CommandWords créé pour l'occasion.
- Dans ComandWords, j'ai bien-sûr ajouté le mot « test » à la liste des commandes valides
- Dans la classe GameEngine j'ai modifié la fonction goRoom() :
  Anciennement, la fonction vérifiait si le second mot était « invalid » (valeur anciennement assignée au second mot, directement par getCommand() si la direction était invalide).
  Maintenant la fonction vérifie par elle-même si la direction est invalide grâce au booléen pCommand.isDirection(vDirection).

Pour ce qui est du test en lui-même :

Dans la classe GameEngine, j'ai presque recopié l'exemple d'utilisation fourni par la section « Lecture simple de fichiers de texte » de la page « Plus de Technique » dans la liste officielle des exercices :

- J'ai ajouté l'import des classes Scanner, File et FileNotFoundException
- J'ai ajouté le mot clé « test » au switch case de interpretCommand () qui execute alors la fonction « executeTest(vCommand) » décrite ci-dessous :

J'ai donc ajouté la fonction executeTest() qui reprend donc l'exemple de lecture simple de fichier.
J'ai concaténé le second mot de la commande avec « .txt » puis remplacé « **/// traitement de la ligne lue »** l'appel de interpretCommand(vLigne) où vLigne correspond à une nouvelle ligne du fichier .txt.
J'ai aussi remplacé « **/// traitement en cas d'exception »** par un message dans le terminale qui indique que le fichier est introuvable en affichant le nom complet du fichier auquel on a essayé d'accéder.

J'ai aussi créé le fichier essai.txt à la racine avec quelques commandes à tester.
Tout fonctionne correctement lors de son exécution et les erreurs liés à la commande (ex : fichier inexistant) sont bien prises en charges.

Exercice 7.28.2

J'ai créé 3 fichiers :
- court.txt
- optimal.txt : on va chercher la clé dans l'arbre et on va à la porte
- complet.txt : tout est exploré, et les cas d'erreurs sont aussi testés

Exercice 7.29

Dans la classe Player :

- J'ai ajouté l'import de la classe Stack.
- J'ai créé l'attribut aName pour stocker le nom du joueur avec son accesseur
- Anciennement dans GameEngine j'y ai déplacé les attributs aCurrentRoom (avec son getter et son setter) et aPreviousRooms .
- J'ai créé un constructeur sans paramètres qui initialise aPreviousRooms ainsi que le nom du joueur en demandant dans un popup le nom au joueur, et si la String est vide ou ne contient que des espaces, on assigne le nom « Link » par défaut.
- J'ai aussi déplacé les assignations de currentRoom de GameEngine dans les 2 fonctions goRoom() (qui prend en paramètre la Room de destination) et goBack() (qui renvoie un booléen pour indiquer si on pouvait effectivement retourner en arrière).

Dans la classe GameEngine() :

- J'ai supprimé les attributs et l'import déplacés dans Player
- J'ai ajouté un attribut aPlayer et sa fonction setPlayer() (car l'objet Player est créé dans la classe Game). Cette fonction appelle aussi this.aPlayer.setCurrentRoom(this.aStartRoom) car la première pièce est définie dans createRoom() qui est appelée par le constructeur de GameEngine avant qu'on lui donne son joueur. Donc j'ai créé un attribut aStartRoom qui est assigné dans createRooms() et lu dans setPlayer().
- Maintenant, à la fin de createRooms(), on se contente d'assigner la première Room dans l'attribut créé à cet effet.
- Dans goRoom() on appelle simplement la méthode du même nom sur l'attribut aPlayer si le déplacement est valide.
- Dans goBack(), on appelle simplement la fonction du même nom sur l'attribut aPlayer. Cette fonction renvoie « true » ou « false » si l'historique est vide, permettant ainsi à GameEngine d'indiquer si le retour est possible dans la GUI.
- J'ai personnalisé un peu le message de bienvenue et d'au revoir avec le nom du joueur

Dans la classe Game :

- Je déclare un nouvel attribut aPlayer et le créée dans le constructeur.
- Comme pour la GUI, je le connecte au GameEngine avec this.aEngine.setPlayer(this.aPlayer)

Exercice 7.30

Dans ajouté à la classe Player :

- un attribut aCurrentItem
- addItem() qui assigne l'objet Item passé en paramètre
- removeItem() qui lui donne null comme valeur
- getItem() son accesseur
- hasItem() qui renvoie true ou false pour savoir si le joueur a déjà un objet.

Dans la classe Room :

- Une nouvelle fonction removeItem() permet de retirer l'objet de la hashmap via son nom passé en paramètre.
- Une nouvelle fonction getItem() permet de retourner l'objet Item de la pièce, qui n'avait pas besoin d'accesseur auparavant (seulement de renvoyer une String pour l'affichage).

Dans la classe GameEngine :

- J'ai ajouté la prise en comptes des 2 nouvelle commandes « prendre » et « poser » dans interpretCommand()
- J'ai donc ajouté la fonction take() qui vérifie d'abord s'il y a bien un 2nd mot à la commande et si le joueur a déjà un Objet.
  Puis qui récupère créé un objet de type Item qui stocke l'objet dans la currentRoom qui porte ce nom.
  Si l'objet se trouve être null, c'est qu'il n'y a pas de tel objet dans la currentRoom, autrement, l'Item est ajouté au Player et retiré de la pièce courante.
  Un message est renvoyé au joueur en fonction de chaque situation.
- Ainsi que la fonction drop() qui ne prend pas de paramètre car il n'y a qu'un objet potentiel à déposer dans la pièce courante.
  Après avoir vérifié l'absence de second mot et le port d'un objet par le Player, drop() ajoute à la pièce courante l'item du joueur et lui retire donc ce dernier.
  Un message est renvoyé au joueur en fonction de chaque situation.

Bien-sûr, j'ai aussi ajouté à CommandWords les 2 mots clé « prendre » et « poser ».

Mode d'emploi

Aucun pour le moment.

Déclaration anti-plagiat

Je déclare n'avoir plagié aucun scénario, ni aucun code déjà existant excepté ceux fournis par les professeurs.