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

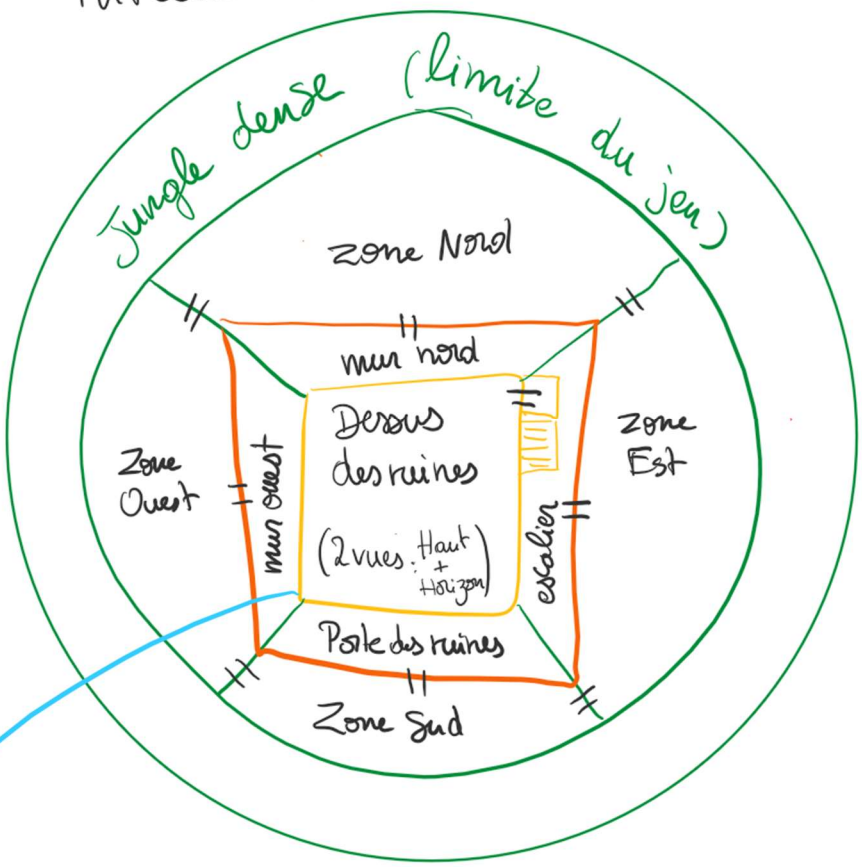
Un archéologue Sheikah découvre un manuscrit vieux de 5000 ans mentionnant un artefact capable de transformer la chaleur en mouvement. Il part à sa recherche dans la jungle Korogu, où une ruine souterraine l'attend.

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)

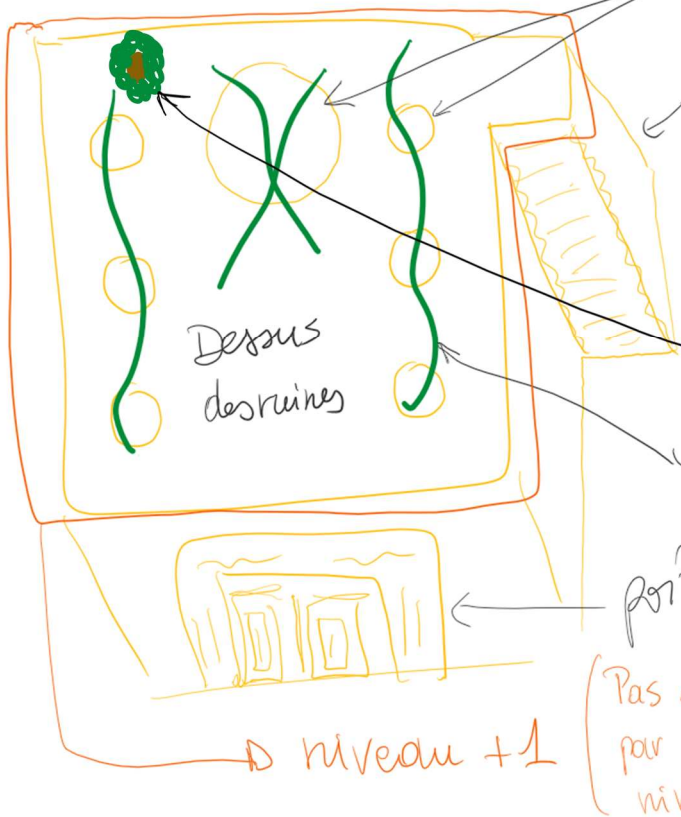
niveau 0 (Ground)



détails

= : passage possible

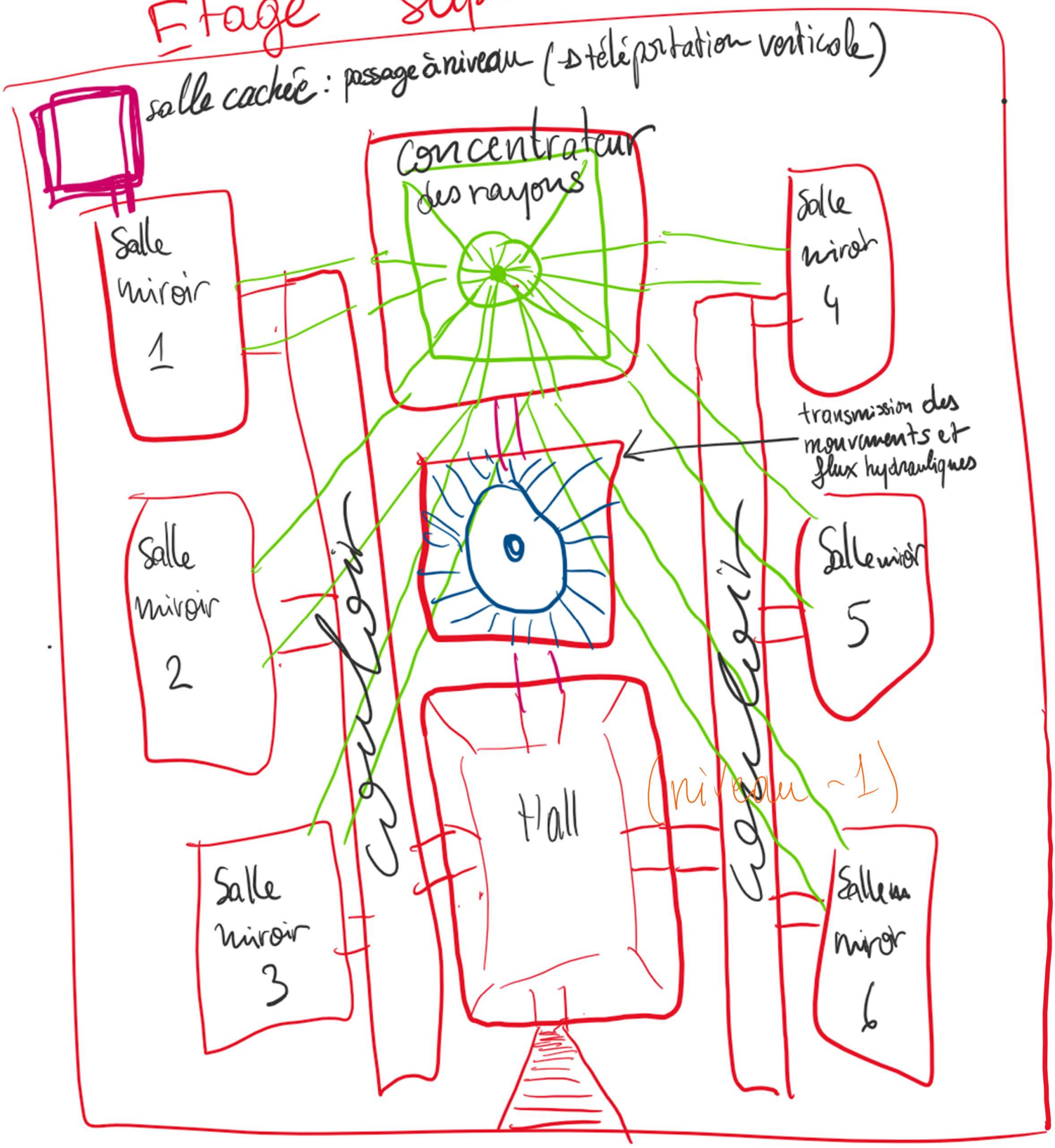
Vue 3D de la surface des ruines



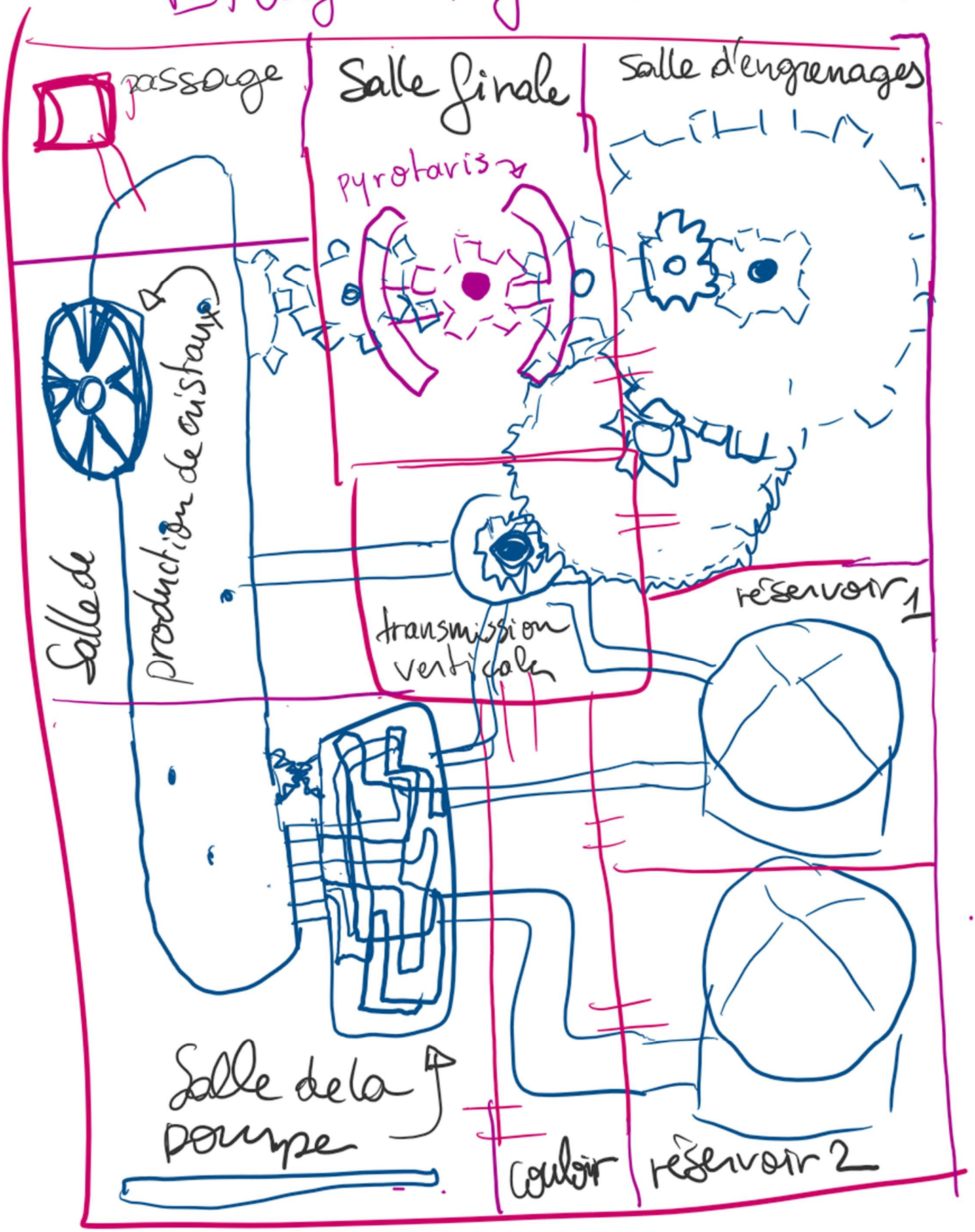
vue de Haut

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

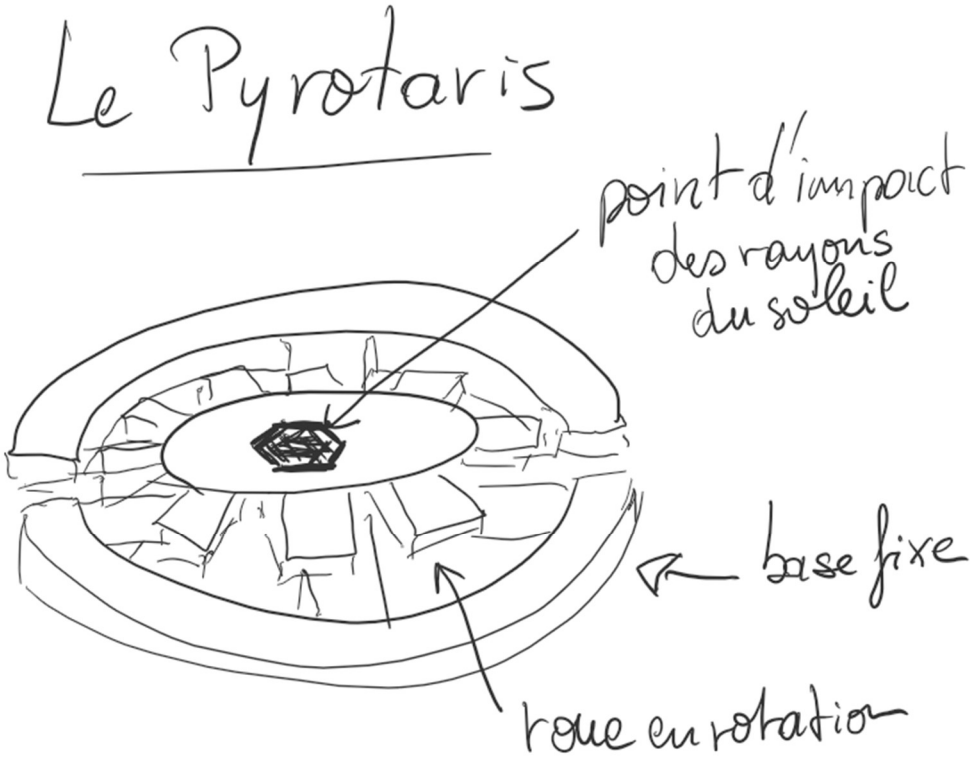
Étage supérieur



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

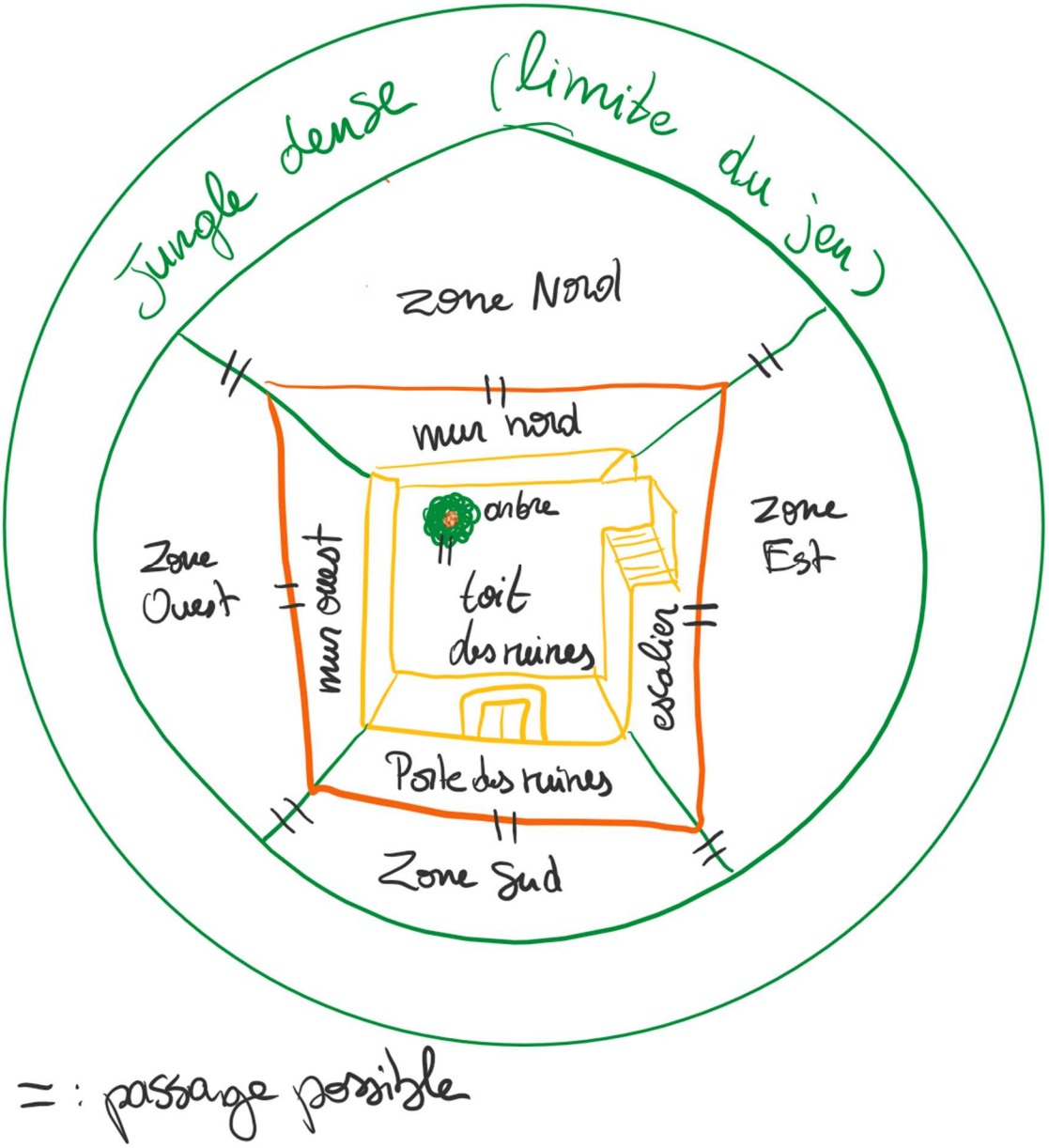
- Le but final est encore simple, j'aimerais trouver quelque-chose de plus concret que de juste trouver le pyrotaris. Le plan peut encore changer, les machines sont encore sans réelle utilité dans les ruines, il faut leur trouver un sens. Une fonction à cette ruine (fabriquer des cristaux peut être pour 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 se trouve dans un nid de pie dans l'arbre, 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 7.12

Non réalisé

Exercice 7.13

Non réalisé

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 7.18.4

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

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.