Document de conception : comment répondre au besoin → description de la solution Description la plus précise possible, avec différentes versions de “finition” (et description de chaque version) : version “minimale”, version(s) “intermédiaire(s)”, et version “idéale”

Document de conception

Librairies:

- pygame
- tkinter
- pyyaml
- json

Le jeu est basé sur un système de tile pour la plupart de ses systèmes pour créer un fonctionnement en "LEGO" comme pour la map et le système d'inventaire qui est encore en dévellopement. La plupart des systèmes sont gérés à l'aide d'un tableau en 2 dimension qui contiennent des objets comme le tableau map qui stoque la totalité de la carte. Dans le futur en fonction des performances cette méthode sera changée pour un chargement partiel de la carte avec un système de tronçons. Tout le jeu est réalisé avec la librairie pygame avec une utilisation de tkinter pour l'éditeur car plus simple pour créer de nombreux boutons pyyaml et json pour charger les fichiers .yml et .json qui servent à la configuration du jeu Les fichiers se présentent de la façon suivante (ici le fichier de configuration pour le bananier présent dans content\tiles):

```
name : "Bananier"
fileName : "bananier.png"
doPass : false
type : "surmap"
```

Le jeu est chargé à l'aide du fichier init.py qui va parcourir les différents répertoires où se trouvent les fichiers de configuration dans le dossier content Cette méthode permet de rajouter de nouveaux élements comme des tiles ou des items dans le jeu très simplement et sans toucher à aucun moment le code du jeu

Exemple : chargement des différents tiles

```
files = os.listdir("content\\tiles")
for i in files:
    inf = yaml.load(open("content\\tiles\\"+i,"r"),Loader=Loader) # Chargement du fichier yaml
    print(inf)
    if(inf["doPass"] == True): # Résolution d'un bug assez étrange , cette solution semble fon
        inf["doPass"] = True
    else:
        inf["doPass"] = False
```

```
    Tile(inf["name"],inf) # Appel du constructeur
```

La méthode Tile() appelle un constructeur car Tile est une classe qui se trouve dans le dossier classes avec le reste des classes Il prend en argument le nom ainsi que de nombreuses options sur cette tile comme la texture , si elle est traversable (doPass) qui sont directement puisées dans les fichiers de configuration .yml

```
def __init__(self,name,opt1,**opt2):
```

```
Tile("nom",fileName="file.png")
Tile("nom",{doPass:False,fileName:"file.png"})
```

La solution utilisée pour la génération aléatoire et/ou procédurale de la carte de jeu n'a pas encore été décidée néanmoins un essai a été réalisé avec un algorythme de bruit de Perlin (Fichier perlinNoise.py pour l'instant l'option est activable avec l'argv perlin):

```
from perlin_noise import PerlinNoise
noise = PerlinNoise(octaves=0.1, seed=123456)
for i in range(200):
    for j in range(200):
        print(noise([i,j]))
```

Le code du jeu est fait pour s'adapter à de nombreuses résolutions et cette dernière est modifiable dans le fichier config.json

```
{
  "fen":{
    "height":720,
    "width":1080
  }
}
```

Les différentes versions visées

- [x] Minimale : Le moteur du jeu tourne fluidement mais il est impossible d'interragir avec la carte (cette version est atteinte à ce stade du projet)
- [ ] Intermédiaire : Les fonctions de base du jeu sont maintenant implémentées. Le joueur peut récuperer les ressources sur la carte , il peut utiliser des machines comme des fours , des coffres.
- [ ] Idéale : Un mode multijoueur est utilisable