



Lacouture Anaïs
Meurillon Alex
Pressenda Ugo



ROGUE DEAD REDEMPTION



PIÈGES :

Une salle piégée qui contient des pièges placés de façon aléatoire, dont l'apparence ressemble aux cases de sol '.' mais qui font des dégâts au héros qui les traverse.

- On crée une nouvelle classe, Piège, qui hérite de la classe Element. (De sorte que le piège ne bouge pas)
- Dans cette classe on fait une méthode meet, qui lorsque le Héros tombe sur le piège, lui enlève 2 hp.
- Le piège est enlevé de la map, sinon on peut tomber plusieurs fois dedans et c'est embêtant
- On imprime un gentil message qui dit qu'on s'est pris un piège.



INVISIBLE :



Des monstres (par ex. fantômes) sont invisibles ('.') jusqu'à ce qu'ils frappent le héros ou que le héros les frappe. Nous avons décidé de faire ce petit monstre comme étant aussi un petit voleur, on ne le voit pas jusqu'à ce qu'il vienne nous taper, et nous voler de l'or ! Le voleur vole de l'or à chaque fois qu'il rencontre le héros

```
monsters = {0: [Creature("Natif", 4, "n", poison=True), Creature("Cowboy", 2, "w")],\n    1: [Creature("Sherif", 6, "S", strength=2), Creature("Loup", 10, "l")],\n    5: [Creature("Dutch", 20, strength=3, poison=False)],\n    3: [Creature("Voleur", 6, ".")]}
```

```
else:\n    self.hp -= other.strength\n    if other.name=="Voleur":\n        other.abbrv="V"\n        if self.gold>0:\n            self.gold-=1
```

- On rajoute la créature voleur dans le dictionnaire des monstre, sous forme d'un point, pour qu'il soit invisible.

- Dans la fonction meet de Creature, on regarde si la créature que l'on a touché est le voleur, si c'est le cas alors celui ci devient visible, abrv=V, et on enlève un or au héros.

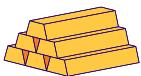


ARMES :

Le héros peut équiper une arme (action use) parmi plusieurs existantes qui ajoute des points de force à sa force initiale. On gère le changement d'arme.

La classe Weapon qui hérite de la classe Equipment permet au héros d'obtenir un nouvel objet lui attribuant plus de force. De plus une méthode Equip permet au héros d'équiper cet objet et de mettre à jour les nouvelles caractéristiques dans la description du héros.

```
class Weapon(Equipment):\n    def __init__(self, name, abr=False, incr=0, armorpene=0, damagetype=None, isrange=False):\n        Equipment.__init__(self, name, abr, False)\n        self.strength = incr\n        self.armor_penetration = armorpene\n        self.damage_type = damagetype\n        self.isrange = isrange\n\n    def equip(self, creature):\n        #équiper d'une arme la créature#\n        from theGame import theGame\n        # print(">- Dans Weapon equip with",self.name,"actual Weapon",creature.weapon)\n        if creature.weapon != self:\n            theGame().addMessage("The hero equiped a " + self.name+ " and gained "+str(self.strength-creature.weapon.strength)+" strength")\n            creature.strength -= creature.weapon.strength\n            creature.strength += self.strength\n            creature._inventory.pop(creature._inventory.index(self))\n            creature._inventory.append(creature.weapon)\n            creature.weapon = self\n            creature.armor_penetration = creature.weapon.armor_penetration\n            creature.damage_type = creature.weapon.damage_type
```



BOUTIQUE :

Une salle qui contient en son centre un marchand propose de vendre/d'acheter des équipements en échange de l'or du joueur.

- On crée une nouvelle classe Boutique, qui hérite de la classe Room, on décore celle-ci avec le Marchand que l'on met au centre, on fait donc appel à une autre classe (class Marchand)

```
class Marchand(Element):
    def __init__(self, name="Marchand", abr="M"):
        super().__init__(name, abr)
        self.elem = [theGame.theGame().randEquipment() for x in range(3)]

    def select(self, elem):
        from utils import getch
        print("Que voulez vous acheter ? : " + str([str(elem.index(e)) + ":" + e.name for e in elem]))
        c = getch()
        if c == '^[[[':
            return None
        elif c.isdigit() and int(c) in range(len(elem)):
            return elem[int(c)]

    def prix(self, elem):
        from theGame import theGame
        for l in theGame().equipments.items():
            for objet in l[1]:
                if type(elem) == type(objet):
                    return (l[0] + 1) * 2

    def meet(self, creature):
        from Hero import Hero
        from theGame import theGame
        if len(self.elem) != 0:
            if isinstance(creature, Hero):
                print("Prices are: " + " ".join([x.name + ":" + str(self.prix(x)) for x in self.elem]))
                elem = theGame().select(self.elem)
                if elem == None:
                    pass
                elif elem.name == "gold" and creature.gold - self.prix(elem) >= 0:
                    creature.gold -= self.prix(elem)
                    creature.gold += 1
                    self.elem.pop(self.elem.index(elem))
                elif elem.name != "gold" and creature.gold - self.prix(elem) >= 0:
                    creature.gold -= self.prix(elem)
                    creature._inventory.append(self.elem[0])
                    self.elem.pop(self.elem.index(elem))
            else:
                theGame().addMessage("Not enough resources. See you soon !")
        else:
            theGame().addMessage("Shop empty")
```

```
from Room import Room

class Boutique(Room):
    def __init__(self, c1, c2):
        super().__init__(c1, c2)
        self.centre = self.center()

    def decorate(self, map):
        from Marchand import Marchand
        map.put(self.centre, Marchand())
```

La classe Marchand() permet de mettre un marchand sur une carte afin de vendre des équipements.

La méthode select() affiche les objets vendus et renvoie l'élément sélectionné par le héro. Possibilité de sortir avec la touche échap.

La méthode prix() fixe le prix de chaque objet, 2 fois son taux d'apparition.

La méthode meet() gère la rencontre avec le héro et lui ajoute l'objet désiré

Le marchand gère le fait que le héro n'est pas assez de crédits et que le magasin soit vide.



NOURRITURE :

Le héros à un niveau de satiété (20) qui descend chaque 5 actions. Si le héros tombe à 0 en satiété, il perd un hp chaque 5 actions, jusqu'à ce qu'il utilise une nourriture, il reprend alors 5 niveau de sasiété.

- On crée une méthode nourriture qui ajoute 5 à la sasiété.
- Dans le dictionnaire d'équipement, on rajoute l'Equipment Bière, qui dès son utilisation appelle la méthode nourriture.

```
{0: [Equipment("Clopes", "c", usage=lambda self, hero: hero.heal()), \
      Equipment("gold", "o"), Equipment("Bière", "b", usage=lambda self, hero: hero.nourriture()), \
```

```
def nourriture(self):
    self._faim+=5
    return True|
```

```
def descriptionJeu(self):
    # Description du héros dans le jeu
    return '\n'+HP : '+ Creature.description(self) \
        + " sur : " + str(self.hpm) + '\n' \
        +'Inventaire : ' + str(self._inventory) + '\n' \
        +'XP : ' + str(self.xp) + '\n' \
        +'Niveau : ' + str(self.level) + '\n' \
        +'Faim : '+str(self._faim) + '\n' \
        +'Or : ' + str(self.gold) + '\n' \
        + ("Vous êtes empoisonné !" if self._empoisonné else "")+ '\n' \
        + ("Vous allez mourir de faim !" if self._faim==0 else "")
```

On rajoute dans la boucle (de 5 coups, comme celle du poison), de la méthode play, une commande qui enlève 1 niveau de faim au héros. Et si celui ci n'a plus de niveau de faim, alors tous les 5 coups il perd une vie



Si le héros dois manger (qu'il a 0 de sasiété), alors on affiche à chaque tour qu'il va mourir de faim !

POINT D'EXPÉRIENCE (XP) :

- Chaque monstre rapporte un nombre de points d'expérience, le héros a un niveau, qui monte suivant une échelle donnée. Quand il change de niveau il gagne en force et en hp maximum et regagne tous ses points de vie (hp)
- Dans la fonction meet de créature on ajoute une condition qui, quand le héros tue la créature, celui-ci prend son niveau de force + 1 en xp.
- A chaque fois que le nombre d'xp dépasse une dizaine, le héros augmente d'un niveau, et reprend toutes ses vies, +1 vie qu'il n'avait pas initialement.
- Dans les attributs d'instance du Heros, on lui a instancié un niveau, level. `self.level = level`

```
#gère les xp du héros, niveau et l'augmentation du niveau etc...
    if isinstance(other, Hero):
        a=other.level
        other.xp+=self.strength+1
        other.level+=other.xp//10
        other.xp=other.xp%10
        if a<other.level:
            other.strength+=1
            other.hpmax+=1
            other.hp=other.hpmax
    if self.hp > 0:
        return False
    return True
```

```
HP : <Héros>(19) sur : 21
Inventaire : [f]
XP : 1
Niveau : 1
Faim : 9
Or : 10
Vous avez ramassé : Fusil à pompe.
[]
```

REPOS :^{zzz}

Une fois par carte (cf étages), le héros peut regagner 5 hp, mais les monstres effectuent 10 déplacements.

```
'r': lambda hero: hero.__setattr__('_repos', True), \
```

- Le héros (joueur), peut appuyer sur r pour se reposer, et reprendre des hp, pendant que les monstres bougent toujours, l'appui de r appelle la méthode repos() dans Hero, qui, si repos est vrai (c'est à dire que le héros veut se reposer), alors pendant un certain temps (tempsrep) dans une boucle, reprend une vie par vie. On doit donc appuyer 5 fois sur une touche mais on est immobilisé, pour reprendre ses vies, pendant que les monstres bougent (si les hp ont atteint les hpmax, alors on arrête le repos).
- Dans le init de la classe Hero on instancie deux nouvelles variables, qui changeront d'état lors de l'appel de la fonction repos, lorsqu'on appuie sur r.

```
def repos(self):
    if self._repos==True:
        if self.tempsrep>0:
            self.tempsrep-=1
            if self.hp<self.hpmax:
                self.hp+=1
                return True
            else:
                self.tempsrep=0
    return False
```

```
self._repos = False
self.tempsrep = 5
```

```
for i in range (0,5):
    print()
    print(self._floor)
    print("Vous vous trouvez à l'étage " + str(self._floor.cachemap1()) + "#nuge de vi")
    print(self._hero.descriptionjeu())
    print(self.readMessages())

    if self._hero.repos()==False:
        c = getch()
        self._hero.détruire_inventory()
        if c in self._actions:
            self._actions[c](self._hero)
    else :
        c = getch()
        self._floor.deplIntelMonstre()
        if self._level==5 and len(self._floor._elem)=
```

C'est dans la boucle play que les 5 coups sont attendus, si repos est faux alors on joue normalement, sinon on attend juste une action de la part du héros, sans l'exécuter.

Enfin pour l'exécuter qu'une seule fois par étage, on réinitialise les variables dans buildfloor, au moins à chaque fois qu'on passe dans un escalier et que buildfloor est utilisé, alors repos se remet à False et on peut donc le remettre à True (quand il est utilisé).

```
def buildFloor(self):
    from Map import Map
    """Crée une carte pour l'étage actuel."""
    self._hero._repos=False
    self._hero.tempsrep=5
```

DÉPLACEMENT INTELLIGENTS :

- Les monstres sont capables de retrouver leur chemin vers le héros

```
class Pathfinder:  
    def __init__(self, matrix, start, end):  
        self.matrix = matrix  
        self.start = start  
        self.end = end  
        self.rows = len(matrix)  
        self.cols = len(matrix[0])  
        self.neighbors = [(1, 0), (-1, 0), (0, 1), (0, -1), (1, 1),  
                          (-1, 1), (-1, -1), (1, -1)]  
  
    def heuristic(self, p1, p2):  
        # Fonction heuristique (distance euclidienne)  
        x1, y1 = p1  
        x2, y2 = p2  
        return math.sqrt((x2 - x1) ** 2 + (y2 - y1) ** 2)  
  
    def find_path(self):  
        # Initialiser les ensembles ouverts et fermés  
        open_set = []  
        closed_set = set()  
  
        # Initialiser le noeud de départ  
        start_node = (self.start, 0)  
        # Utilisez une file d'attente prioritaire pour stocker les nœuds  
        heapq.heappush(open_set, start_node)  
  
        # Gardez une trace du chemin de chaque nœud  
        path = {}  
        # Gardez une trace du coût pour atteindre chaque nœud  
        g_score = {self.start: 0}  
  
        while open_set:  
            # Obtenez le nœud avec le coût le plus bas  
            current_node, current_cost = heapq.heappop(open_set)  
  
            # Vérifiez si nous avons atteint l'objectif  
            if current_node == self.end:  
                return self.reconstruct_path(path, current_node)
```

On crée une Classe Pathfinder qui permet de faire du A* pathfinding

Cet algorithme donne le chemin optimal entre un point A et B

Le principe de l'algorithme A* (A-star) dans un jeu Rogue-like est d'effectuer une recherche de chemin efficace et optimale pour trouver le meilleur chemin entre deux points dans une carte de jeu.

1. Initialisation : L'algorithme commence par initialiser les ensembles "ouvert" et "fermé", ainsi que les coûts g-score et h-score pour chaque nœud. Le nœud de départ est ajouté à l'ensemble ouvert.
2. Exploration : L'algorithme commence à explorer les nœuds en retirant le nœud avec le coût total le plus bas de l'ensemble ouvert. Il examine les voisins de ce nœud et calcule leur coût g-score en ajoutant le coût du nœud actuel et le coût du déplacement vers le voisin. Si le voisin n'est pas déjà dans l'ensemble fermé et que le nouveau g-score est inférieur à son g-score actuel (ou s'il n'est pas encore évalué), le voisin est ajouté à l'ensemble ouvert avec son coût total.
3. Évaluation : À chaque étape, l'algorithme évalue le voisin avec le coût total le plus bas en utilisant la fonction de score f-score (f-score = g-score + h-score). Cela garantit que le nœud le plus prometteur est exploré en premier.
4. Terminaison : L'algorithme continue d'explorer les nœuds jusqu'à ce que le nœud de destination soit atteint ou que l'ensemble ouvert soit vide. Si le nœud de destination est atteint, le chemin est reconstruit en suivant les liens de nœud en nœud depuis la destination jusqu'au départ.
5. Résultat : Une fois le chemin trouvé, il peut être utilisé pour guider le joueur à travers la carte en se déplaçant de nœud en nœud.

On fait aussi et surtout la méthode depIntelMonstre (déplacement intelligent monstre), dans laquelle on appelle la classe Pathfinder et la fonction find_path pour que celui-ci se déplace intelligemment.

Cette méthode remplace la méthode moveallmonster que l'on appelle dans la boucle play.

```
def depIntelMonstre(self):  
    from algo_djikvi import Pathfinder  
    # Position du héros  
    p_hero = self.pos(self._hero)  
    end = (p_hero.y, p_hero.x)  
    for e in self._elem:  
        if isinstance(e, Creature) and e != self._hero:  
            mapgraph=[[0 for j in range(self.size)] for i in range(self.size)]  
            for i in range(self.size):  
                for j in range(self.size):  
                    if self._mat[i][j]!=Map.ground and self._mat[i][j]!=e:  
                        mapgraph[i][j] = 1  
            mapgraph[p_hero.y][p_hero.x]=0  
            pos_monstre = self.pos(e)  
            # Position du monstre  
            start=(pos_monstre.y, pos_monstre.x)  
            path_finder = Pathfinder(mapgraph, start, end)  
            path = path_finder.find_path()  
            if path==None:  
                pass  
            else:  
                move_x=path[1][1]-path[0][1]  
                move_y=path[1][0]-path[0][0]  
                self.move(e,coord(move_x,move_y))
```

NUAGE DE VISIBILITÉ :

- Le joueur ne voit pas toute la carte, mais seulement les salles et les portions droites de couloirs déjà visitées (ou partant des salles déjà visités).
- On crée une méthode cachemap (dans la classe Map) qui cachent la partie de la map en remplaçant des 1 par des 0, en découvrant la map ceux ci deviennent des 1, donc on peut voir ce qui s'y trouve.
- On crée 2 copies de la vrai map : une pour avoir la carte cachée et une comme variable pour savoir si on est déjà passé dans cette salle "1"

```
# Méthode qui permet de cacher la carte et lorsque le joueur passe par un chemin, celui-ci reste découvert jusqu'à la fin du niveau
def cachemapv2(self):
    # Créer une copie de la matrice
    mapdujoueur=[row.copy() for row in self._mat]
    # Parcourir la copie de la matrice et associer le caractère "#"
    for i in range(self.size):
        for j in range(self.size):
            if mapdujoueur[i][j]!=Map.empty and self.pos(self._hero).distance(Coord(j,i))>4 and self.mapdonnee[i][j]=="0":
                mapdujoueur[i][j] = Map.hide
            else:
                self.mapdonnee[i][j]="1"
    s = ""
    for i in mapdujoueur:
        for j in i:
            s += str(j)
        s += '\n'
    return s
```

```
class Map(object):
    """A map of a game
    Contains game elements
    ground = '.' # A wall
    dir = {'z': Coord(0,1), 'x': Coord(1,0), 'y': Coord(-1,0), 'n': Coord(0,-1)}
    empty = ' ' # A node
    hide = '#' # A hidden node
```

- On ajoute un attribut de classe, hide = "#", l'élément qui va cacher la map, un peu comme un nuage, cependant c'est mieux de mettre un espace ' ', au moins on ne voit même pas la forme des salles etc...

NUAGE DE VISIBILITÉ + :

Le joueur ne voit les monstres et équipements qu'à l'endroit où il se trouve (avec nuage de visibilité).

On utilise le même principe que pour l'autre cachemap.

```
def cachemapv1(self):
    # Créer une copie de la matrice
    mapdujoueur=[row.copy() for row in self._mat]

    # Parcourir la copie de la matrice et associer le caractère "#"
    for i in range(self.size):
        for j in range(self.size):
            if mapdujoueur[i][j]!=Map.empty and self.pos(self._hero).distance(Coord(j,i))>4:
                mapdujoueur[i][j] = Map.hide
    s = ""
    for i in mapdujoueur:
        for j in i:
            s += str(j)
        s += '\n'
    return s
```

```
print("--- Bienvenue au héros ! ---")
while self._hero.hp > 0 and not self._end:
    for i in range (0,5):
        print()
        print(self._floor)
        print("Vous vous trouvez à l'étage " + str(self._level))
        print(self._floor.cachemapv1()) #Nuage de visibilité + (=
```

Dans le play (boucle de jeu), on rajoute à la map le cachemap (v1 ou v2).

DIAGONALES :

Gérez le déplacement du héros et des monstres en diagonale.

```
cos45 = 1 / math.sqrt(2)

def direction(self, other):
    """Returns the direction between two coordinates."""
    d = self - other
    cos = d.x / self.distance(other)
    if cos > Coord.cos45:
        if d.y<0:
            return Coord(-1,1)
        elif d.y>0:
            return Coord(-1,-1)
        return Coord(-1, 0)
    elif cos < -Coord.cos45:
        if d.y>0:
            return Coord(1,-1)
        elif d.y<0:
            return Coord(1,1)
        return Coord(1, 0)
    elif d.y > 0:
        return Coord(0, -1)
    return Coord(0, 1)
```

Comme nous avons fait le déplacement intelligent pour les monstres, la fonction direction ne nous sert plus, mais celle ci servait à savoir où se trouvait le héros par rapport au monstre et donc dans quelle direction le monstre devait se déplacer, y compris les diagonales.

Evidemment on rajoute les touches qui permettent au joueur de se déplacer en diagonales, dans le dictionnaire actions.

```
'a': lambda h: theGame.theGame().___floor__.move(h, Coord(-1, -1)), \
'w': lambda h: theGame.theGame().___floor__.move(h, Coord(-1, 1)), \
'e': lambda h: theGame.theGame().___floor__.move(h, Coord(1, -1)), \
'c': lambda h: theGame.theGame().___floor__.move(h, Coord(1, 1)), \
```



INVENTAIRE LIMITÉ :

- Le héros ne peut stocker que 10 équipements maximum, ajoutez une action permettant de détruire un équipement. L'or est compté à part, il ne fait plus partie de l'inventaire.
- Nous avons créer une fonction qui demande au héros de choisir un élément à jeter lorsqu'il prend un élément mais que son inventaire est plein.

```
def detruire_inventory(self):
    from utils import getch
    from Equipment import Equipment
    if len(self._inventory)>3:
        retire=True
        print("Choose item> " + str([str(self._inventory.index(e))+": " + e.name for e in self._inventory]))
        while retire:
            c = getch()
            if c.isdigit() and int(c) in range(len(self._inventory)) and int(c)!=Equipment("gold","o"):
                self._inventory.remove(self._inventory[int(c)])
                retire=False
```

POISON:



Des monstres peuvent empoisonner le joueur, il perd un hp chaque 5 actions, jusqu'à ce qu'il se soigne.

```
def poison(self):
    if self._poison==True:
        self.hp-=1
    return True
```

- On ajoute dans le dictionnaire des équipements une potion (ici des cigarettes), qui appelle la méthode heal.
- La méthode heal redonne 3 hp au héros et enlève l'état empoisonné du héros.

```
equipments = {0: [Equipment("Clopes", "c", usage=lambda self, hero: hero.heal())]}
```

- Ajout d'une boucle de 5 dans la fonction play (comme pour la nourriture), car si le héros est empoisonné, le poison agit sur lui tous les 3 coups, on appelle ici la fonction poison, qui fait que si le héros est empoisonné (self._poison=True) alors il perd de la vie.

```
def heal(self):
    self.hp += 3
    self._poison=False
    return True
```

```
if self._poison==True:
    if other._empoisonné==False:
        theGame.theGame().addMessage("Le " + self.description() + " a empoisonné le " +other.name +' !')
        other._empoisonné=True
theGame.theGame().addMessage("Le " + other.name + " a frappé : " + self.description())
if self.hp<=0:
```

Dans le meet de Creature, on empoisonne le héros, si jamais la créature porte la caractéristique poison. Dans ce cas, et si le héros n'est pas déjà empoisonné alors on affiche un message pour tenir le joueur au courant. Et comme pour la nourriture, on affiche que le héros est empoisonné dans sa description à chaque tour si c'est le cas.

```
+ ("Vous êtes empoisonné !" if self._empoisonné else "")+ '\n' \
```

ARMURES :

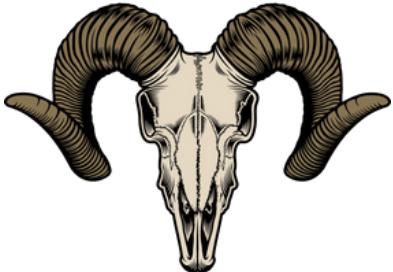


Certains équipements, une fois portés (action use) permettent au héros de réduire des dégâts qu'il subit. Gérez le changement d'armure

```
La creature est rencontrée par une autre créature
L'autre touche la créature. Renvoie True si
if self.armure==1:
    if other.strength>=7:
        self.hp -=7
    else:
        self.hp -= other.strength
elif self.armure==2:
    if other.strength>=6:
        self.hp -=6
    else:
        self.hp -= other.strength
elif self.armure==3:
    if other.strength>=5:
        self.hp -=5
    else:
        self.hp -= other.strength
elif self.armure==4:
    if other.strength>=4:
```

```
Equipment("gold", "o"), Equipment("Biere", "o", usage=lambda self, hero: hero.nourriture()),
Equipment("Armure de paille", "a", usage=lambda self, hero: armure_paille(hero)), \
1: [Equipment("Petit tonerre", "t", usage=lambda self, hero: teleport(hero, True)), \
Equipment("Armure cow-boy", "A", usage=lambda self, hero: armure_cb(hero)), \
Weapon("TNT", "T", 60)], \
#2: [Equipment("bow", usage=lambda self, hero: throw(1, True))], \
2: [Weapon("Pistolet", "p", 15, armorpene=0.5), Weapon("Fusil à pompe", "f", 15, damagetype=["frozen"]),
Equipment("Armure de plomb", "P", usage=lambda self, hero: armure_plomb(hero))]
```

Les armures sont des Equipements. Une fois équipée, les dommages des ennemis sur le héros sont limités. Tout ceci est géré dans le meet() de créature. Celles-ci sont ajoutées dans le dictionnaire des équipements, et lorsqu'elles sont utilisées, appellent leur fonction associée.



FIN DU JEU :

```
def play(self):
    """Boucle de jeu principale"""
    self.buildFloor()
    print("--- Bienvenue au héros ! ---")
    while self._hero.hp > 0 and not self._end:
        for i in range (0,5):
            print()
            print(self._floor)
            print("Vous vous trouvez à l'étage " + str(self._level))
            #print(self._floor.cachemapv1()) #Nuage de visibilité
            print(self._hero.descriptionJeu())
            print(self.readMessages())
```

Le héros a fini le jeu et est invité à passer un bon séjour dans le far west lorsqu'il arrive dans une certaine salle, dans laquelle il n'y a pas d'escalier, mais un boss et beaucoup d'élément, qu'il doit tout ramasser avant de finir.

Dans la méthode buildfloor, si le dernier level est atteint alors on implémente un étage particulier, grâce à self.last_dance. Dans le last_dance c'est la potion, l'armure et l'arme la plus puissante et le boss final.

```
elif self._level==3:
    self._floor = Map(hero=self._hero)
    self._floor.put(self._floor._rooms[-1].center(), self.last_dance["monstre"])
    map=self._floor
    self.addMessage("Vous êtes dans la salle du boss, finissez là")
    for i in range(1,4):
        if i==1:
            boost=self.last_dance["potion"]
        elif i==2:
            boost=self.last_dance["armure"]
        else:
            boost=self.last_dance["arme"]
```