Escape

**0/Vrac**

* S’échapper d’un labyrinthe/d’une salle/d’un ensemble de salle le plus vite possible
* Des mobs poursuivent le joueur
* Plusieurs type de mobs : suiveur débile, seulement tourner à droite, besoin de voir, renifleur
* Particules dissipés par le joueur lorsqu’il marche
* Jeu dynamique/rapide
* Enchainement direct, pas de transition entre les niveaux

**I/ Principe de jeu**

Jeu solo qui consiste en un personnage se déplaçant dans un environnement fermé (une salle, un labyrinthe, selon le niveau).

Le joueur est poursuivi par divers monstre et doit arriver à la fin du niveau avant de se faire attraper.

Les déplacements du joueur sont rapides et dynamiques. Ils doivent se faire dans 8 directions pour être plus pratique pour le joueur.

Il sera éventuellement possible de ramasser une arme/des grenades/des items afin de se défendre sommairement, et d’étoffer le gameplay.

Un niveau se divise en plusieurs salles plus ou moins tortueuses qui se succedent.

Lors du début du niveau, les contrôles sont rappelés au joueur via des icônes s’affichant à l’écran.

Lors de l’arrivée dans une salle, la direction à prendre pour en sortir est indiquée au joueur par une flèche pointant vers la sortie pendant quelques instants.

Lorsque le joueur atteint la fin de la salle, on fait un traveling de la vue sur la salle suivante (modifier la vue). Et on remet l’indication de sortie.

Lorsque le joueur atteint la fin du niveau, un écran affiche qu’il a réussi (et certaines stats ?) et on retourne à l’écran principale/des niveaux.

///Algorithme

1/ afficher l’écran principale  
2/ le joueur choisis un niveau  
3/ lancer le niveau, afficher les indications de touches  
4/ Entrée dans une salle, indiquer la direction de la sortie (flèche clignotante)  
5a/ le joueur atteint la limite de la salle, retour au 4  
5b/ le joueur se fait attraper, afficher animation de mort & panneau de défaire, retour au 1  
5c/ le joueur atteint la fin du niveau  
6/ on affiche qu’il a réussi (et des stats ?), retour au 1

**II/ Aspect**

Le joueur doit être relativement petit par rapport à son environnement, pour donner une impression de vitesse et être plus maniable

Le joueur est vu de dessus (façon hotline miami) mais légèrement décalé pour voir son visage (de face) ou son dos (de… dos…).

Des particules mimant la poussière sont générées par le déplacement du joueur et des autres personnages non joueurs.

L’interface est limitée à rien au début, et pourra être étoffé par un compteur de balle ou un indicateur des items disponibles.

L’apparence générale des environnements peut aller de la grotte à la forêt en passant par l’hopital, la ville ou autre TANT QUE la cohérence globale au sein d’un niveau/d’une zone et que la transition est fluide.

**II.1/ Vue**

Les niveaux sont divisées en salle de taille relativement identique, la vue s’adapte à la taille de la salle puis se déplace lorsque l’on atteint la fin de celle-ci. La vue doit être un attribut du niveau et présente dans le fichier (au chargement de la map ?) + changer grâce au script (translation + déterminer position).

Note : pour le moment, conservé une vue avec le joueur au centre. Attention : la vue ne doit pas sortir de la map (tester si vue.x < 0 ou > largeurMap)

Déterminer la taille de la vue par test.

|  |  |  |
| --- | --- | --- |
| Largeur Vue | Hauteur Vue | Verdict |
| 2000 | 1500 | Trop petit |
| 3000 | 2000 | Pas mal |

**III/ Bestiaire**

**Les IAs poursuivant le joueur utilisent un système de vue afin d’aller directement dans la direction du joueur dès qu’il n’y a plus d’obstacle devant**

* Chien monstrueux débile : se contentent de courir dans la direction du joueur (prendre en compte le blocage ?). Vitesse supérieur ou égale à celle du joueur.  
  La méthode utilisé pour le déplacement et la recherche est le « crash&turn ». De cette façon l’IA semble désordonnée voir perdu, mais aussi pressée et agressive.  
  De plus on utilisera des générateurs pour en générer un nombre important de monstres afin de donner une impression de horde voir de vague déferlante.  
  Les chiens devront ressembler au lézard cornu du désert (celui qui crache du sang avec ses yeux), mais avec des teintes violettes, bleues et rouges, et avec une gueule de chien, afin de faire ressortir leur côté dangereux et monstrueux.
* Fantôme qui errent dans une direction aléatoire jusqu’à la prochaine intersection ?  
  Le fantôme a une vitesse très lente, mais est relativement imposant. Sa direction est toujours aléatoire et il prendra la décision de sa direction à chaque intersection. (Jusqu’à ce qu’il voit le joueur ? auquel cas il le poursuit jusqu’à le perdre de vue)
* Truc qui ne peut tourner qu’à droite.  
  Apparence indéterminée.  
  Le monstre se déplace très vite (plus que le joueur) et décide à chaque intersection s’il tourne à droite ou pas (aléatoire) (attention à ne pas le bloquer ? ou ballec ?).  
  Pas de système de « vue » pour poursuivre le joueur.
* Truc qui doit voir sa cible et la poursuit jusqu’à la perdre.  
  Apparence indéterminée (genre de goule ?).  
  La créature a deux états importants : recherche (vitesse assez lente, choix aléatoire des directions à chaque intersection) et chasse (vitesse rapide, poursuit le joueur jusqu’à le perdre de vue ou l’attraper). Si le monstre perd le joueur de vue dans cet état, il se rend à la dernière position connu du joueur puis reprend sa routine.
* Truc qui peut renifler sa proie et la trouve peu importe où elle se situe sur la carte.  
  Apparence : créature à 4 pattes et aveugles, avec des membres sensorielles surdimensionnés (à travailler).  
  Enregistre le chemin parcouru par le joueur et le remonte jusqu’à lui. Déplacement très rapide pour augmenter le stress.  
  A\* jusqu’à la première case connu du joueur (ou première case croisée) puis suivre le chemin jusqu’à la position suivante.
* Nom pour l’une des bêtes : Lia (L’IA) (le fantôme ?)

**IV/ Map**

La map est conçu grâce à des tiles, dispose de plusieurs couches dont une sur laquelle se déplace les acteurs

On y trouve un début (player start) et une fin (que le joueur doit atteindre) ainsi que d’autre « point d’intérêt » disposant d’un indice : point de spawn des monstres, objets à ramasser, interrupteur, etc

Il conviendra de redessiner la façon d’écrire la map dans un fichier pour la rendre plus modulable (méthode balise similaire au HTML), d’y changer la façon de positionner les points d’intérêts (par rapport au tiles et non à la position) et d’y ajouter la gestion des autres points d’intérêts (couplés à un autre fichier chargeant les monstres ou directement dans le fichier map -> préférer un « construction script » ?).

Edit : La gestion des scripts et des checkpoints sera déléguée au « level », nouvelle classe à implémenter qui gèrera le personnage, la map, les autres sprites, les **textures**, les checkpoints et les scripts (en bref le déroulement du niveau)

COMPOSANTS :

*<Map>*: les limites du fichier, pour contrôler son intégrité

*<Tileset>*: propriété de la texture du tileset, le nombre de tiles, leurs dimensions et leurs propriétés

*<MapProp>*: propriété de la map, dimension de la map (en tile), des tiles (en jeu, en pixel), du nombre de couches, de la couche du personnage et schéma des tiles.

*<Autre>*: position du départ et de l’arrivée.

**V/ Level**

Le level détermine les limites et les objets d’une partie dans un niveau. Il comprend la map, le joueur, tous les sprites (divers, crashTurn, etc), les effets de particules, les checkpoints, les scripts, etc, etc.

On y trouve un début (player start) et une fin (que le joueur doit atteindre) ainsi que d’autre « point d’intérêt » disposant d’un indice : point de spawn des monstres, objets à ramasser, interrupteur, etc

De la même façon que la map, on utilise un système de balise à la manière d’un site HTML. Cependant, contrairement à cette map il conviendrat de spécifier beaucoup plus d’arguments pour chaque objet (afin de les personnaliser autant que possible) : couleur, position de départ, fichier à charger, etc. **TOUTES** les options doivent être couvertes par **CHAQUE** ligne.

On créera un tableau de pointeur d’Updatable pour faciliter le traitement des différentes classes.

Le level contrôle la partie gameplay. Pour cela on créera une fonction « play » qui prend la fenêtre en paramètre et agis comme la fonction main du prototypage. Cette fonction renvoi un entier qui détermine si le joueur a gagné, est mort ou souhaite arrêter de jouer.

Cette classe fonctionne de la même manière que slick, avec une fonction *update()* qui fonctionne dans un thread séparé (ou pas ?) (utiliser plusieurs thread pour partager la gestion des IA, des particules, du joueurs, etc ?) et une fonction *render()* appelé à chaque fin de boucle.

COMPOSANTS :

*<Level>*: les limites du fichier déterminant son intégrité.

*<Texture>*: la liste des textures utilisées par le niveau pour qu’elles soient disponibles et chargées une seule fois

*<Map>*: le fichier map à charger et d’autres spécificités potentielles

*<Sprite>*: les sprites à charger, ils se divisent en plusieurs catégories (non limités)

* *<joueur>* Le sprite du joueur
* *<limites>* Les sprites réprésentant les limites du niveau (début et fin)
* *<crashturn>* Les sprites des IA Crash&Turn (à revoir, peut être utilisé un émetteur similaire à un émetteur de particules ?)
* *<autre>*

*<Particle>*: les différents émetteurs de particules. Il conviendra de les diviser comme les sprites le sont, pour séparer les linkedParticleSystem des simples ParticleSystem ainsi que de rajouter l’attribut couleur (et les autres manquants).

*<Checkpoint>* : les différents checkpoint du niveau. Les checkpoints sont de types point, ligne ou carré, et déclenche des évènements (sauvegarde, script, transition de vue, etc) A TRAVAILLE

*<Script>*: les scripts déclenchés dans le niveau PARTIE A CONCEVOIR

<Music> : liste des musiques à jouer pendant le déroulement du niveau. Se contenter d’une seule pour l’instant

**VI/ Personnage**

Idée : lorsqu’il est attrapé par un monstre, le personnage peut essayer de se dégager pour ne pas mourir

**VI/ Generateur de monstre (crashTurn & autre)**

Il est nécessaire de générer des points de spawn pour faire apparaître les monstres de façon autonome (voir dynamique).

Les monstres ne sont pas tous générés en même temps mais chacun leur tour selon un laps de temps défini.

Problématiques :

* Comment gérer la génération et l’updates de tous les monstres ?
  + S’inspirer des générateurs de particules
* Comment initialiser tous les monstres avec les bonnes valeurs ?
  + Toutes les valeurs nécessaires doivent être stockées dans la classe émettrice
  + Créer un monstre « modèle » qui sert à stocker les valeurs mais n’est pas utilisé
* Comment générer des monstres de façons décalées (pas tous en même temps) ?
  + Tester si le nombre max de monstre n’a pas encore été créé et si un certain laps de temps (déterminé) est passé.
  + Dans ce cas, appelé fonction « create() » qui ajoute un monstre au tableau.
* Comment gérer l’update de tous les monstres ?
  + Fonctions updates sur le générateur de monstres et boucle sur le tableau de monstres

**VII/ Sound design**

1. *Musique* : toujours très rythmée et rapide - style variant en fonction de l’environnement
   1. Flamme/métal -> rock/métal
   2. Désert -> guitare acoustique, rythme style cow-boy
   3. Futuriste -> électro/dubstep
   4. Forêt -> instrumentales
2. *Bruitage*: bruits de pas (différents bruits pour différentes surfaces), d’aboiement pour les chiens, de cris pour les poursuivants (récupérer cris zombie CoD ?)  
   Bruitage d’environnement : vent, bois qui craque, flammes, électricité

Boost ::python

<http://www.boost.org/doc/libs/1_62_0/libs/python/doc/html/index.html>

<http://www.boost.org/doc/libs/1_47_0/libs/python/doc/tutorial/doc/html/index.html#python.quickstart>