Rémi VANSTEELANDT

Université de Picardie Jules Verne

Licence 3 Informatique

« Haunted Gardens »

Analyse du projet

Année universitaire 2011-2012

Enseignant : Dieudonné

Haunted Gardens

Analyse

But du document

Ce document a pour but de décrire le fonctionnement général du logiciel et de permettre son évolution. Les points principaux seront abordés : structure des classes, algorithmes particuliers, stockage des données.

Ce document n'a pas pour but de décrire chaque élément du logiciel. Certaines parties pourront évoluer en fonction du développement (optimisation), tandis que d'autres n'ont pas besoin d'être traitées (affichage).

Toute évolution majeure du logiciel ou d'une partie du logiciel devra faire l'objet d'une mise à jour de ce document.

Classes

Le jeu sera développé en Javascript. Or, Javascript est un langage objet, mais ne permet pas d'utiliser des classes et interfaces de la même façon que les langages traditionnels. Un objet Javascript n'a pas vraiment de type fixe, et on peut lui ajouter dynamiquement de nouveaux attributs et nouvelles méthodes.

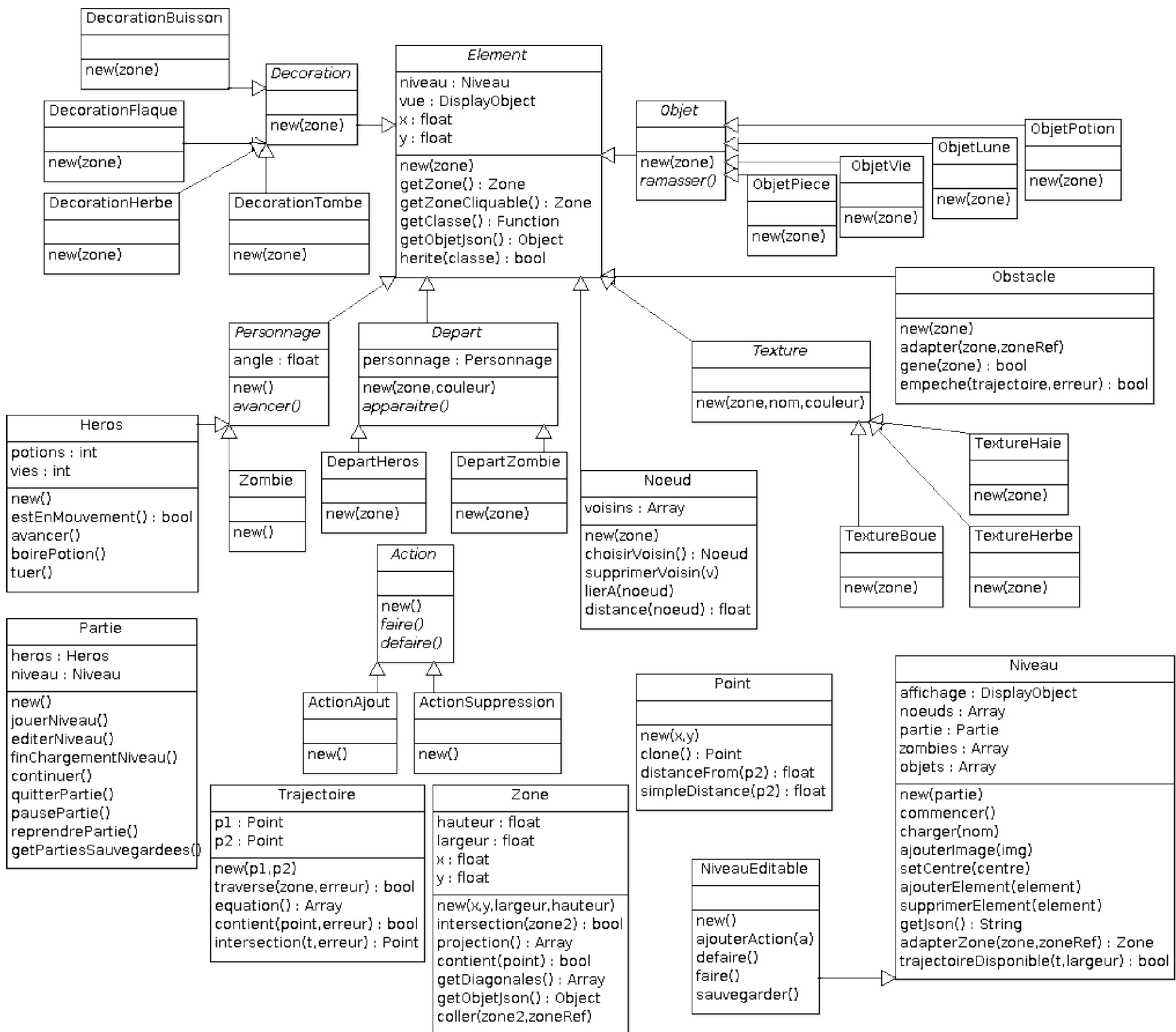
Cependant, il est possible d'écrire un programme en Javascript proprement en simulant l'utilisation de classes. J'utiliserai cette méthode pour garder un code le plus clair possible.

Il en sera de même pour l'accès aux variables d'instance : Javascript ne permet pas la création d'attributs privés, mais j'utiliserai quand même des méthodes d'accès et de modification pour permettre des traitements liés aux modifications (par exemple, la modification de l'angle d'un personnage doit provoquer la modification de la rotation de sa vue). Cela passera soit par des méthodes traditionnelles, soit par des setters et getters, qui permettent de ne pas voir la différence entre un attribut et un setter. De plus, les attributs d'une classe débuteront toujours par un underscore (« _ »), permettant de les différencier des setters/getters, et ne seront accédés que depuis la classe elle-même ou depuis une classe fille.

Enfin, du fait du « manque de typage » de Javascript, de nombreuses interfaces n'auront pas à être créées (pour les vues notamment). Cela sera également dû au besoin de légèreté (Javascript ne peut assurer des performances trop élevées). Le pattern MVC ne sera donc pas

suivi à la lettre.

Le diagramme de classes global est le suivant :



Les attributs de classes sont en fait des setters et getters (aucun accès direct à un attribut ne sera fait, sauf depuis une classe fille éventuellement). Les attributs et méthodes privés ne sont pas représentés, car ils seront écrits au fur et à mesure du développement.

Classe Element

Le jeu ayant besoin d'un éditeur de niveau, il faut pouvoir créer facilement tous les éléments de ce niveau (personnages, objets, décorations...) tout en gardant le tout facilement modulable (l'ajout d'un type d'élément ne doit pas demander une réécriture complète du code).

Un élément peut être défini et donc enregistré simplement par son type, par sa position, et par la zone qu'il occupe (parfois, la zone ne sera pas utile mais uniquement la position).

A partir de cela, j'ai pu déduire une classe abstraite Element qui généralise tous les éléments. Elle dispose d'un constructeur qui prend en paramètre une zone, ce qui rend simple son utilisation avec l'éditeur de niveau. Il suffira de renseigner une zone pour créer un élément.

De cette classe dérivent d'autres classes (abstraites ou concrètes) :

- Personnage
- Texture
- Objet
- Depart
- Noeud
- Decoration
- Obstacle

A partir de ces sous-classes, des types plus particuliers pourront être définis. Toujours pour simplifier, chaque type d'élément aura sa classe. Par exemple, une texture d'herbe sera une TextureHerbe, et non pas une Texture particulière. Cela permet de toujours garder un constructeur avec pour seul paramètre la zone occupée par l'élément, ce qui simplifiera son utilisation dans l'éditeur, et lors des chargements de niveaux.

Enfin, chaque élément aura sa propre vue. Une vue sera un objet graphique dérivant de DisplayObject (voir « Classes d'affichage »). La classe Element ne dérive pas d'une classe graphique afin de permettre plus de libertés (pour ne pas « cloisonner » les éléments à une seule classe graphique possible).

getZone()	Récupération de la zone occupée par l'élément.
getZoneCliquable()	Récupération de la zone cliquable occupée par l'élément (utile à l'éditeur de niveaux)
getClasse()	Récupération de la classe de l'élément.
getObjetJson()	Récupération d'un objet simplifié représentant l'élément. Il contient uniquement le strict nécessaire à l'enregistrement de l'élément.
herite(classe)	Test d'héritage de l'élément par rapport à une classe d'élément. Cette méthode est nécessaire à cause du typage particulier de

	Javascript.

Classe Personnage

Afin d'assurer les déplacements des personnages, ceux-ci seront représentés par leur position (x,y), leur angle, leur vitesse, ainsi que leur largeur (ou rayon). A partir de ces trois caractéristiques, il sera possible de calculer :

- la nouvelle position :
  *(x,y) → (x + cos(angle) * vitesse, y + sin(angle) * vitesse)*
- la validité d'une position (voir « Gestion des collisions »)

getZoneOccupee(position)	Récupération de la zone occupée par le personnage s'il était situé à la position spécifiée
avancer()	Avancement du personnage.
positionSuivante()	Récupération de la future position du personnage.

A partir de cette classe dérivent Heros et Zombie, dont la principale particularité est la méthode avancer.

La classe Heros comprend également les réglages propre à la partie : nombre de vies restantes, nombre de potions collectées.

Classe Depart

Lors de la création d'un niveau, ce ne sont pas les personnages qui sont ajoutés mais leurs points de départ. C'est dans ce but qu'existe la classe abstraite Depart. Ainsi, au début d'un niveau, on appellera la méthode apparaitre() de tous les éléments héritant de Depart, pour que les personnages correspondant soient créés et positionnés.

Classe Partie

La classe Partie sera la classe principale. C'est par elle qu'on entrera dans le programme. Elle se chargera de faire l'intermédiaire entre le jeu et l'affichage. Elle se chargera de faire avancer le jeu en appelant les méthodes des différents personnages. Elle fera office de contrôleur.

chargerListeNiveaux()	Chargement du fichier contenant l'ordre des niveaux.
jouerNiveau()	Lancement du jeu pour le niveau en cours
editerNiveau()	Lancement de l'éditeur de niveau
continuer()	Avancement de la partie
quitterPartie()	Mettre fin à la partie, réinitialisation des statistiques.
pausePartie()	Mise en pause de la partie
reprendrePartie()	Reprise de la partie après une mise en pause
niveauSuivant()	Passage au niveau suivant
chargerPartie(donneesPartie)	Chargement d'une partie précédemment sauvegardée.
sauvegarderPartie()	Sauvegarde de la partie en cours.
getPartiesSauvegardees()	Récupération de la liste des parties sauvegardées.

Classe Niveau

La classe Niveau permettra de représenter un environnement dans lequel le joueur pourra évoluer. Pour résumer, un niveau sera constitué d'éléments (classe Element), et d'un affichage qui contiendra les vues de tous ces éléments.

L'accès au niveau permettra aux différents personnages de s'orienter correctement : le niveau devra se charger de les empêcher de passer les obstacles, et permettra de ramasser les objets, de calculer les trajectoires des zombies... De plus, son optimisation sera essentielle car de nombreux calculs passeront par lui (calcul des positions et trajectoires).

commencer()	Début du niveau : apparition des personnages, optimisations, désactivation de certains éléments...
charger(nom)	Chargement du niveau spécifié.
setCentre(point)	Définition du point sur lequel doit être centré le niveau. Pour une partie, ce sera le héros, pour l'éditeur, cela

	dépendra du déplacement de la carte.
ajouterElement(elt)	Ajout de l'élément spécifié au niveau.
supprimerElement(elt)	Suppression d'un élément du niveau.
getJson()	Récupération de la représentation du niveau au format JSON, pour pouvoir l'enregistrer.
adapterZone(zone,zoneRef)	Adaptation de la zone spécifiée en fonction des obstacles et de la zone précédente.
trajectoireDisponible(traj,largeur)	Test de la faisabilité d'une trajectoire pour la largeur spécifiée.
zoneDisponible(zone)	Test de la disponibilité d'une zone en fonction des obstacles.

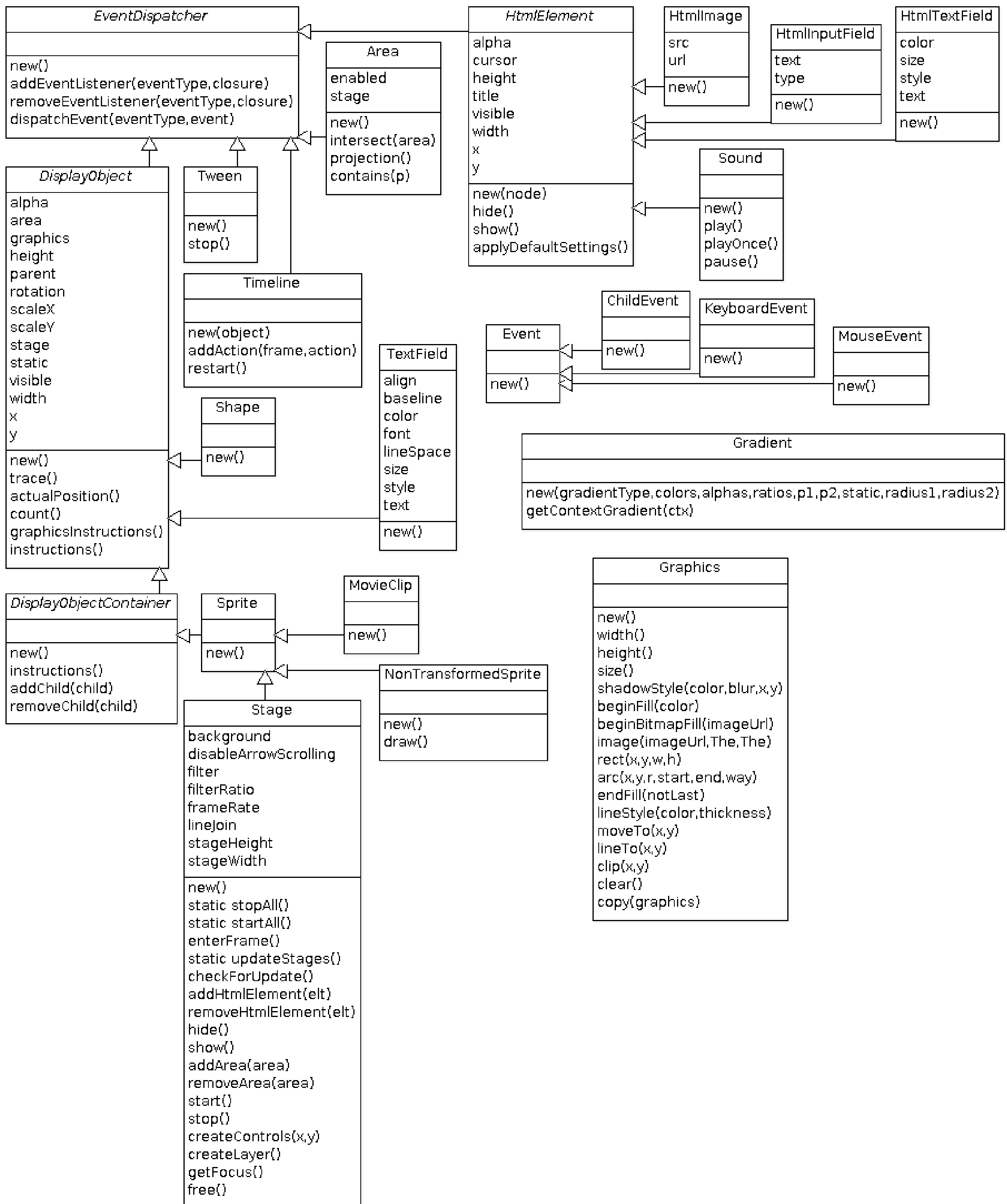
De cette classe héritera NiveauEditable, qui aura plusieurs méthodes supplémentaires permettant la création d'éléments, et qui permettra l'affichage d'éléments invisibles au cours du jeu (point de départ du joueur, nœuds, obstacles). Elle permettra également la sauvegarde de ce niveau.

Classes d'affichage

Le HTML5 propose une manière efficace de dessiner, mais celle-ci est loin de valoir celle d'autres technologies, comme Flash. En effet, l'API par défaut ne permet que de dessiner directement, mais rien n'est prévu pour stocker les formes et images. Pour pallier ce problème, j'utiliserai un moteur d'affichage réalisé par moi-même : JsAs. Ce moteur permet d'introduire les notions de conteneurs, et permet un affichage plus simple, de la même manière qu'en ActionScript. Ainsi, l'affichage ne sera pas un problème.

Chacun de ses éléments est caractérisé par sa position, sa rotation, et son échelle. Cela rendra simples et performants les déplacements d'éléments sur la scène.

Les vues des différents éléments et menus dériveront toutes de ces classes.



Pour résumer, les classes suivantes seront utilisées :

- **Stage :** scène générale, le plus haut conteneur
- **Shape :** forme sur laquelle le dessin est possible
- **Sprite :** conteneur
- **TextField :** texte affichable

De plus, deux classes abstraites pourront être « citées » :

- **DisplayObject :** désigne un objet graphique quelconque (**Shape**, **TextField**)
- **DisplayObjectContainer :** désigne un objet graphique contenant d'autres objets graphiques (**Stage**, **Sprite**).

Classes Zone, Trajectoire

La classe Zone sera utilisée pour créer tout élément (il s'agit de l'unique paramètre du constructeur), ainsi que pour gérer les collisions avec les obstacles (voir « Gestion des collisions »).

La classe Trajectoire, quant à elle, sera utilisée pour évaluer la faisabilité d'un chemin (voir « Comportement des zombies »).

La classe Zone ayant les méthodes suivantes :

coller(zone,zoneRef)	« Collage » de la zone spécifiée à la zone courante, en restant le plus proche possible de la zone de référence.
intersection(zone2)	Test d'intersection avec la zone spécifiée.
contient(point)	Test pour savoir si la zone contient le point spécifié.
getDiagonales()	Récupération des deux objets de type Trajectoire qui représentent les diagonales de la zone.

Et la classe Trajectoire les méthodes suivantes :

traverse(zone,erreur)	Test de la traversée de la trajectoire avec l'obstacle et la marge d'erreur spécifiée.
intersection(traj2)	Récupération de l'intersection des deux droites formées par la trajectoire courante et la trajectoire spécifiée.
contient(point,erreur)	Test d'appartenance du point à la trajectoire.

Classe Parametres

La classe Parametres contiendra l'ensemble des éléments paramétrables du jeu. Cela permettra de n'avoir à modifier que très peu le code pour en changer certains éléments. Cela permettra également d'utiliser une version « allégée » du jeu, en modifiant ces paramètres.

Par exemple, la taille du cadre du jeu, le réglage de la météo, ou la vitesse des personnages sera définie dans cette classe. Cela permettra de regrouper toutes les « constantes » dans une même classe.

Le contenu de cette classe évoluera fortement au cours du développement, mais elle ne contiendra pas de méthodes. Elle peut donc être assimilée à un tableau associatif global.

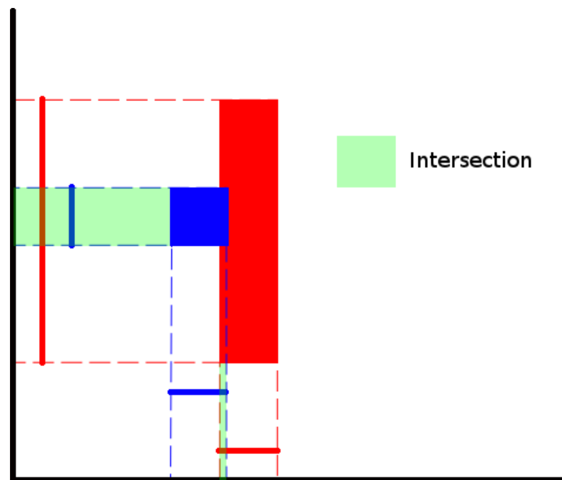
Gestion des collisions

Chaque niveau pouvant être apparenté à un labyrinthe, il contiendra de nombreux obstacles, que l'on appellera collisions. Cette gestion des collisions est essentielle au bon fonctionnement du jeu : les personnages ne doivent pas pouvoir traverser les obstacles, mais le tout doit rester performant.

Pour modéliser les collisions, j'ai choisi de les représenter sous forme de rectangles. Je pourrais également pu les représenter sous la forme de segments, mais étant donné que les obstacles pourront toujours être décomposés en rectangles (pas d'obstacles obliques ou complexes), l'utilisation de rectangle sera plus performante.

Pour gérer la collision entre un personnage et un obstacle, il faut également modéliser le personnage sous forme de rectangle. On a donc un premier rectangle pour l'obstacle, et un second rectangle pour la zone occupée par le personnage. Le rectangle du personnage n'aura pas de rotation quelle que soit la direction de ce dernier.

Il faut ensuite vérifier l'intersection entre les deux rectangles pour vérifier la validité de la position. Pour cela, il suffit d'effectuer une projection sur les axes x et y des deux rectangles, puis de vérifier que les deux intervalles de coordonnées (abscisse et ordonnée) ont une intersection vide.



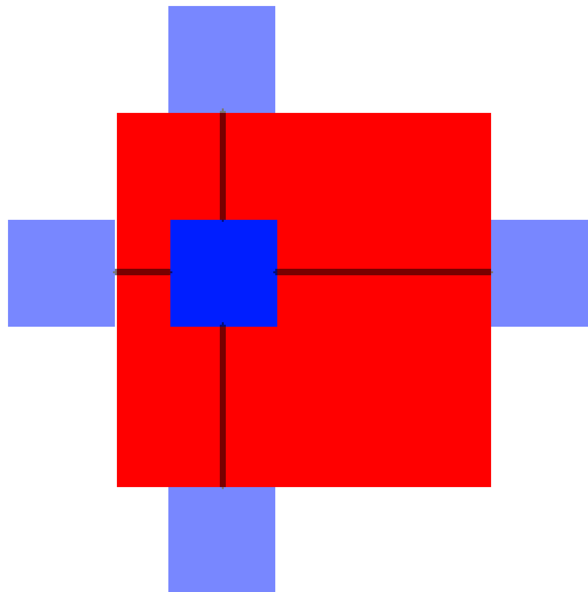
A partir de cet algorithme, on peut savoir si un obstacle gène la future position d'un personnage, et annuler son mouvement si tel est le cas. Cependant, pour plus de jouabilité, il vaut mieux que la position du joueur soit adaptée en fonction des obstacles pour que son mouvement soit gêné, mais pas arrêté. Par exemple, si le joueur se déplace en diagonale et rencontre le coin d'un obstacle, il ne devrait pas être bloqué mais simplement glisser légèrement le long de l'obstacle.

Pour répondre à ce problème, un algorithme supplémentaire adaptera la position d'une zone par rapport à une autre.

Lorsqu'un rectangle mobile a une intersection avec un autre rectangle, il existe quatre positions possibles pour coller le premier au second :

- directement au dessus
- directement en dessous
- directement à droite
- directement à gauche

A partir de ces quatre positions, il suffit de repérer celle qui sera la moins éloignée par rapport à la position précédente, puis de l'appliquer.

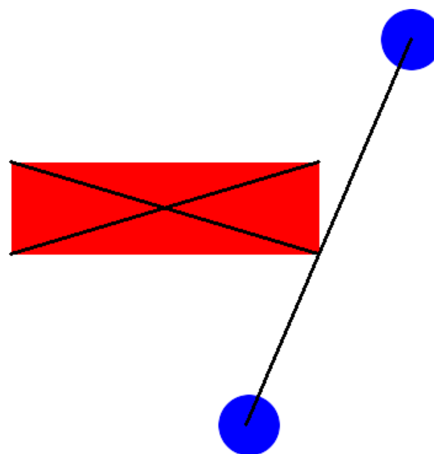


Sur le schéma ci-dessus, on voit bien que le rectangle bleu a quatre possibilités pour ne plus avoir d'intersection avec le rectangle rouge. On choisira alors la zone la moins éloignée de la zone de référence (zone précédemment occupée, avant le déplacement).

Grâce à cet algorithme, le déplacement du héros deviendra bien plus fluide et agréable.

Enfin, un zombie devra régulièrement vérifier s'il peut aller en direction du héros sans être gêné par un obstacle. Il faut donc vérifier que la trajectoire n'a pas d'intersection avec l'obstacle. Pour cela, j'utiliserai uniquement les diagonales : si la trajectoire a un point d'intersection avec l'une des deux diagonales, alors l'obstacle est gênant.

Il serait également possible d'utiliser les quatre côtés de l'obstacle, mais cela impliquerait le double de calculs pour le même résultat.

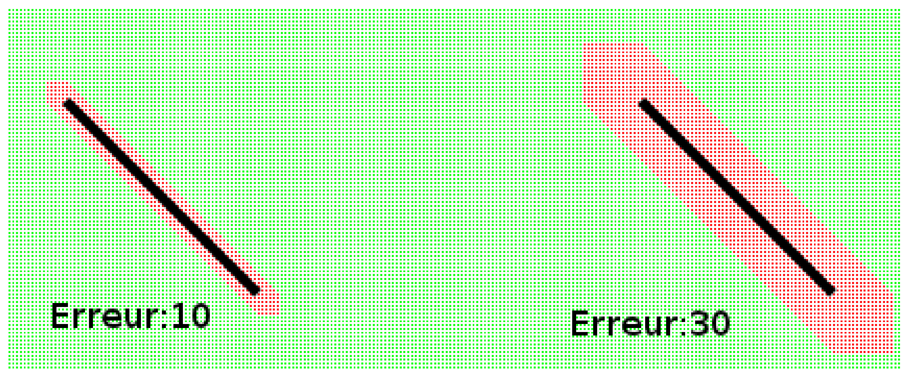


Cependant, cet algorithme a un défaut qui apparaît sur le schéma : la trajectoire est un

segment et n'a pas conséquent pas d'épaisseur, contrairement au zombie qui a une certaine largeur, et qui sera peut-être gêné par l'obstacle. Il arrivera donc qu'un zombie « croie » possible une trajectoire impossible.

Pour pallier ce problème, j'introduirai une marge d'erreur. En effet, pour tester l'intersection entre deux trajectoires, je commencerai par calculer l'intersection des deux droites « portées » par ces trajectoires (qui ne sont donc pas définies par une longueur).

Ensuite je vérifierai son appartenance aux deux trajectoires/segments : si l'intersection des droites semble appartenir aux deux segments, alors il y a intersection des segments. Cette vérification nécessite d'utiliser une marge d'erreur arbitraire, étant donné que les nombres à virgule flottante ne peuvent pas être comparés avec l'opérateur d'égalité. En augmentant la marge d'erreur, le segment contiendra un « nuage » de points plus large et donc l'intersection.



Le test ci-dessus montre le nuage de points appartenant à une trajectoire avec deux marges d'erreur différentes. La zone verte représente les points qui ne sont pas sur le segment, tandis que la zone rouge représente les points situés dessus. Ainsi, on voit qu'en augmentant la marge d'erreur, on donne à la trajectoire une largeur virtuelle.

Pour vérifier l'appartenance d'un point à un segment, une simple vérification sur les projections suffit. Ainsi, pour qu'un point $M(x_m,y_m)$ appartienne à un segment AB avec $A(x_a,y_a)$ et $B(x_b,y_b)$, on vérifie simplement que :

$$min(x_a,x_b) - e < x_m < max(x_a,x_b) + e$$
$$min(y_a,y_b) - e < x_m < max(y_a,y_b) + e$$

avec e la marge d'erreur utiliée.

Au final, pour vérifier qu'un zombie peut atteindre le héros en ligne droite, il suffira de donner une marge d'erreur légèrement supérieure à sa largeur.

Ces algorithmes seront cependant coûteux. Il faudra donc prendre garde, lors de la création

d'un niveau, à ne pas abuser des obstacles, et à les optimiser (si deux obstacles peuvent être fusionnés, il vaudra mieux le faire).

Cette gestion des collisions se fera par les classes Zone et Trajectoire.

Comportement du héros

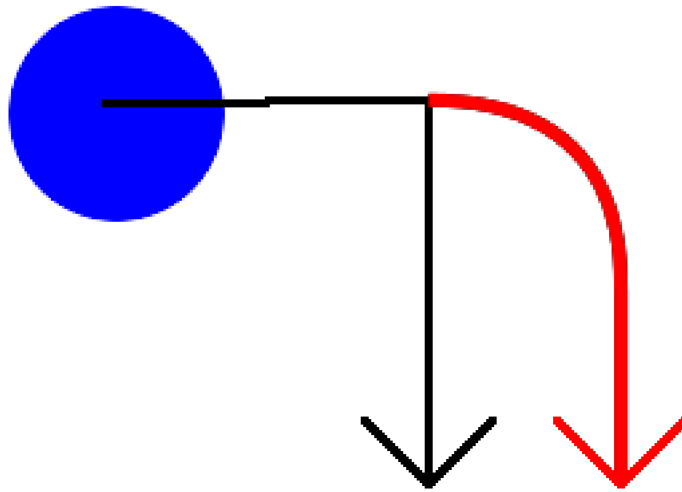
L'algorithme de gestion du héros sera assez simple. Il suffira d'effectuer le déplacement puis de vérifier s'il faut ramasser chaque objet du niveau en cas de mouvement.

```
Variable locale :
	nouvellePosition : Point
Début
	modifier angle en fonction des touches du clavier

	si mouvement alors
		nouvellePosition = positionSuivante()
		niveau.adapter(nouvellePosition)
		appliquer nouvellePosition

		pour chaque objet o de niveau faire
			si zone de o contient nouvellePosition alors
				ramasser o
			finsi
		finpour
	finsi
Fin
```

« Lissage » des angles

Afin de rendre les déplacements du héros plus fluides, un algorithme de « lissage » des angles sera utilisé. Ainsi, lorsqu'on voudra se retourner (passer de 0 à 180° par exemple), le héros ne se retournera pas directement mais effectuera une rotation progressive, Par exemple, il passera par les angles 180 à 90, puis de 90 à 45, et ainsi de suite jusqu'à atteindre l'angle voulu.

Le schéma ci-dessous représente le résultat attendu :



La flèche noire représente la trajectoire sans lissage des angles, et la rouge représente la trajectoire avec.

Pour cela, la difficulté est de trouver le sens de rotation le plus rapide, car les angles sont situés dans l'intervalle [0,360].

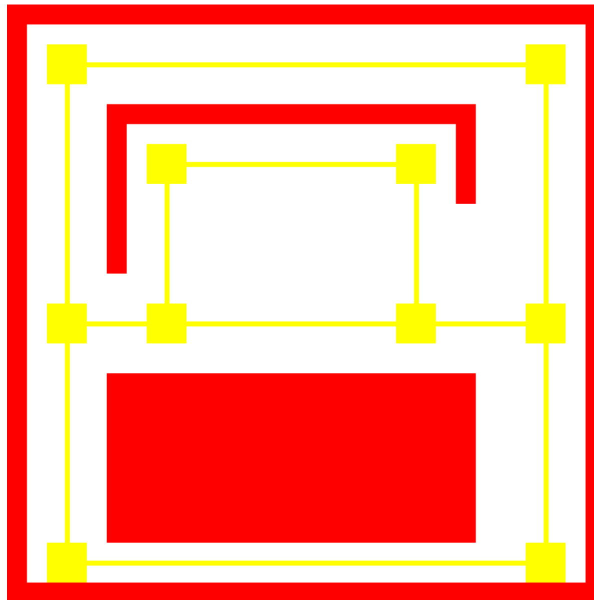
Cela sera réalisé par cet algorithme :

```
Fonction setAngle(a)
Variable d'instance :
	angle : réel
Paramètre :
	a : réel
Variable locale :
	diff : réel
Constante :
	facteurLissage : réel
Début
	diff = a - angle
	diff = (diff + 360) mod 360

	si |diff| < |360-diff| alors
		diff = diff / facteurLissage
	sinon
		diff = (diff - 360) / facteurLissage
	finsi

	angle = (angle + diff + 360) mod 360
Fin
```

Comportement des zombies (« intelligence artificielle »)

Les zombies sont gérés automatiquement par la machine. Le joueur n'a donc pas d'influence directe sur eux, mis à part le fait qu'il tenteront toujours d'aller vers lui, si cela est possible (si aucun obstacle ne gêne).

Leur déplacement dépend de chaque partie, mais sera défini par les nœuds placés au moment de la création du niveau. Un nœud est un élément sur le niveau qui peut être assimilé à un sommet d'un graphe.



Une arête relie deux nœuds si et seulement s'ils sont alignés, c'est-à-dire s'ils ont la même abscisse ou la même ordonnée. Il ne peut exister d'arête oblique (en diagonale). Les arêtes ne sont pas stockées dans le fichier de niveau, et devront donc être recréées à chaque chargement du niveau.

Lors de la création des arêtes entre les nœuds, il faudra faire attention à ne pas créer d'arcs « transitifs », c'est-à-dire qu'il ne devra pas y avoir d'arêtes entre deux nœuds si un troisième nœud se trouve entre les deux. Cela permettra d'empêcher un mauvais choix de nœud et des parcours trop longs. Ceci sera assuré à l'intérieur de la méthode « lierA() » de la classe Nœud.

L'algorithme consiste à ne garder que quatre voisins :

- le plus près au-dessus
- le plus près en dessous
- le plus près à gauche
- le plus près à droite

A chaque ajout d'une liaison, il suffira de vérifier si le nouveau voisin est plus près que celui qui existait déjà à la même position.

De plus, il faudra s'assurer qu'il n'existe aucun obstacle entre deux nœuds alignés, autrement un zombie pourra tenter de suivre un chemin gêné par un obstacle. Pour cela, un simple appel à la méthode trajectoireDisponible de la classe Niveau suffira.

Lors de l'apparition d'un zombie (au début du niveau), un zombie aura deux choix :

- il pourra tenter de suivre le héros, seulement s'il n'y a pas d'obstacle entre les deux (test de la possibilité de la trajectoire)
- il pourra rejoindre le nœud le plus proche non gêné par un obstacle

Lorsqu'un zombie a atteint un nœud, un algorithme similaire est exécuté :

- si la trajectoire entre le zombie et le héros est possible, alors il tentera de le rejoindre
- si cette trajectoire n'est pas disponible, alors il choisira un nœud voisin du nœud courant (un cours algorithme d'évaluation du meilleur choix sera exécuté (voire algorithme du plus cours chemin si nécessaire)

Au cours de son déplacement, le zombie pourra être gêné par des obstacles. Ceci ne pourra arriver que dans un seul cas :

- lorsque le zombie suit le héros, mais qu'un obstacle l'a finalement gêné (le joueur s'est caché). Dans ce cas, il tentera de rejoindre le nœud le plus proche.
- lorsqu'après avoir tenté de rejoindre le nœud le plus proche, un obstacle s'est malgré tout présenté. Dans ce cas, il tentera de rejoindre le nœud le plus proche différent du nœud vers lequel il se dirigeait.

Ce comportement nécessite donc que chaque niveau soit bien structuré en terme de nœuds. Le graphe formé par ces nœuds devra être connexe pour permettre aux zombies de se déplacer partout dans le niveau, et devra ressembler à une « carte » du niveau (les arêtes doivent suivre les chemins à emprunter). De plus, les nœuds ne doivent pas être trop espacés, afin de permettre aux zombies de réévaluer régulièrement leur comportement.

Il permet en plus de n'avoir à évaluer que ponctuellement le besoin d'attaquer le héros ou de prendre une nouvelle direction, ce qui rendra le jeu plus fluide. Cependant, un défaut devrait apparaître : lorsque le zombie se déplacera d'un nœud à un autre et croisera le héros, il ne tentera pas de l'attaquer avant d'avoir atteint le nœud visé. Il faudra donc que les niveaux soient composés de suffisamment de nœuds pour éviter ce problème.

L'utilisation de ces nœuds permettra un comportement plus réaliste : les zombies suivront des chemins prédéfinis, plutôt que de parcourir aléatoirement le niveau.

Enfin, une distance maximale par rapport au héros sera définie pour que les zombies puissent réagir. Cela permettra d'optimiser le jeu, car il est inutile de faire bouger des zombies qui ne sont pas visibles à l'écran. Cela sera utile pour les niveaux plus grands et plus complexes.

L'algorithme du comportement des zombies sera le suivant :

```
Variables d'instance :
	suitHeros : booléen
	noeudVise : Noeud
Variable locale :  nouvellePosition : Point
Constante :	rayonNoeud : réel
Début
	Si noeudVise <> null alors
		Si distance de noeudVise < rayonNoeud alors
			Si trajectoire vers Heros disponible alors
				dirigerVers(Heros)
			Sinon
				dirigerVers(noeudVise.choisirVoisin())
			Finsi
		Finsi
	Sinon si suitHeros alors
		dirigerVers(Heros)
	Finsi
	nouvellePosition = this.positionSuivante();
	Si nouvellePosition disponible alors
		appliquer nouvellePosition

		si distance du Heros < rayon(Heros) + rayon(Zombie) alors
			Heros.mourir()
		finsi
	Sinon
		Si trajectoire vers Heros disponible alors
			dirigerVers(Heros)
		Sinon
			dirigerVers(meilleurNoeud différent de noeudVise)
		Finsi
	Finsi

	vérifier si le zombie touche le héros
Fin
```

L'algorithme de choix d'un nœud voisin sera le suivant :

```
Variable d'instance :
	voisins : liste de noeuds
Variables locales :
	meilleurNoeud : noeud
Début
	meilleurNoeud = null;
	distanceMinimumHeros = VALEUR_MAX
	pour chaque noeud n de voisins faire
		si distance de n à heros < distanceMinimumHeros alors
			meilleurNoeud = n
			distanceMinimumHeros = distance de n à heros
		finsi
	finpour
	si meilleurNoeud = null alors
		retourner this
	sinon
		retourner meilleurNoeud
	finsi
Fin
```

Enfin, l'algorithme de choix du meilleur nœud possible sera le suivant :

```
Variable d'instance :
	voisins : liste de noeuds
Variables locales :
	meilleurNoeud : noeud
	distanceMin : réel
Paramètre :
	different : noeud
Début
	distanceMin = VALEUR_MAXIMUM
	meilleurNoeud = null
	pour chaque noeud n <> different du niveau faire
		si distance jusqu'à n < distanceMin et trajectoire possible alors
			distanceMin = distance jusqu'à n
			meilleurNoeud = n
		finsi
	finpour
	si meilleurNoeud = null alors
		dirigerVers(direction aléatoire)
	sinon
		dirigerVers(meilleurNoeud)
	finsi
Fin
```

Éditeur de niveaux

Classe Action

L'éditeur de niveaux sera représenté par la classe NiveauEditable, qui dérive de la classe Niveau. Cette classe permettra, en plus des fonctionnalités de base d'un niveau, de créer de nouveaux éléments à l'aide de la souris, mais également d'en supprimer. Elle devra donc gérer les événements de la souris pour ajouter des éléments, mais passera pour cela par la classe Action.

La classe Action permettra de modéliser toute modification du niveau (ajout ou suppression). Chaque action peut être faite ou défaite. Ainsi, dans chaque sous-classe de Action, ces méthodes devront être définies. Par exemple, pour la classe ActionAjout, la méthode faire() consiste à ajouter l'élément, tandis que la méthode defaire() consiste à le supprimer. La classe NiveauEditable n'agira donc pas directement sur sa liste d'éléments.

Une liste d'actions est donc stockée au sein du niveau éditable, ainsi que l'indice de l'action en cours. Ainsi, un clic sur le bouton « Défaire » défera l'action en cours et fera diminuer ce compteur. Un clic sur le bouton « Faire » aura l'effet inverse. Cela permettra d'utiliser plus facilement l'éditeur.

Grille

La création d'éléments se fera sur une grille dont les cases seront des carrés de 25 pixels par 25. Cela permettra de simplifier la correspondance entre deux éléments (pour éviter par exemple un décalage entre deux parties d'obstacles). L'appui sur la touche P permettra d'être plus précis (ce sera utile pour l'alignement des objets et nœuds sur des chemins étroits par exemple).

La sauvegarde d'un niveau se fera manuellement : le niveau au format JSON sera transmis à l'utilisateur, qui devra créer un nouveau fichier dans le dossier « niveaux ». Par conséquent, un utilisateur n'ayant pas accès en écriture au dossier des niveaux ne pourra pas les enregistrer.

Enregistrement des niveaux

Les niveaux seront enregistrés sous format JSON, pour plusieurs raisons :

- le format est particulièrement adapté au langage, et donc facile et rapide à interpréter
- il est très léger (comparé au XML par exemple)

– il est facile à lire (en cas de modification manuelle)

Un dossier « niveaux » sera donc créé et contiendra l'ensemble des fichiers des niveaux.

Chaque fichier de niveau sera composé de deux parties principales :

– elements ; qui définira le contenu du niveau. Caque élément étant défini par son type et sa zone, ces deux renseignements suffiront à interpréter le fichier
– parametres : qui définira divers paramètres comme le titre du niveau, le temps imparti... voire d'autres paramètres par la suite

Un fichier de niveau ressemblera à ceci :

```
{
	"parametres" : {
		"titre" : "Exemple de niveau",
		"temps" : 60000
	},
	"elements" : [
		{	"type" : "Zombie",
			"zone" : { "x" : 0, "y" : 0, "largeur" : 100, "hauteur" : 100 }
		}
	]
}
```

Le jeu devant pouvoir être joué sans interactions avec le serveur (après chargement), la sauvegarde des niveaux se fera manuellement. En cliquant sur le bouton « Sauvegarder », l'utilisateur verra apparaître une chaîne de caractères qu'il devra enregistrer dans un fichier qu'il placera dans le dossier correspondant (voir « Arborescence »).

De plus, les niveaux suivent un ordre chronologique. Cet ordre sera défini par un fichier « liste,txt », présent dans le même dossier que les niveaux. Ce fichier contiendra seulement les noms des niveaux séparés par des sauts de lignes. Il sera chargé directement par la partie.

Enregistrement d'une partie

L'enregistrement d'une partie se fera par l'intermédiaire de cookies. Ainsi, les parties sauvegardées seront stockées localement sur la machine du joueur (il ne pourra pas réutiliser une partie d'un ordinateur distant).

Les parties seront stockées au format JSON, toujours pour simplifier leur traitement. Un cookie de sauvegarde ressemblera à ceci :

```
{
	[
		{
			"titre" : "Première sauvegarde"
			"niveau" : 2,
			"vies" : 4,
			"potions" : 5,
			"score" : 1800
		},
		{
			"titre" : "Seconde sauvegarde"
			"niveau" : 1,
			"vies" : 3,
			"potions" : 5,
			"score" : 1880
		}
	]
}
```

Lorsque le joueur chargera une partie, il retrouvera son score, son nombre de vie, ses potions et commencera à partir du début niveau en cours.

Pour charger une partie, une liste déroulante des parties enregistrées apparaîtra, et il suffira au joueur de choisir laquelle reprendre.

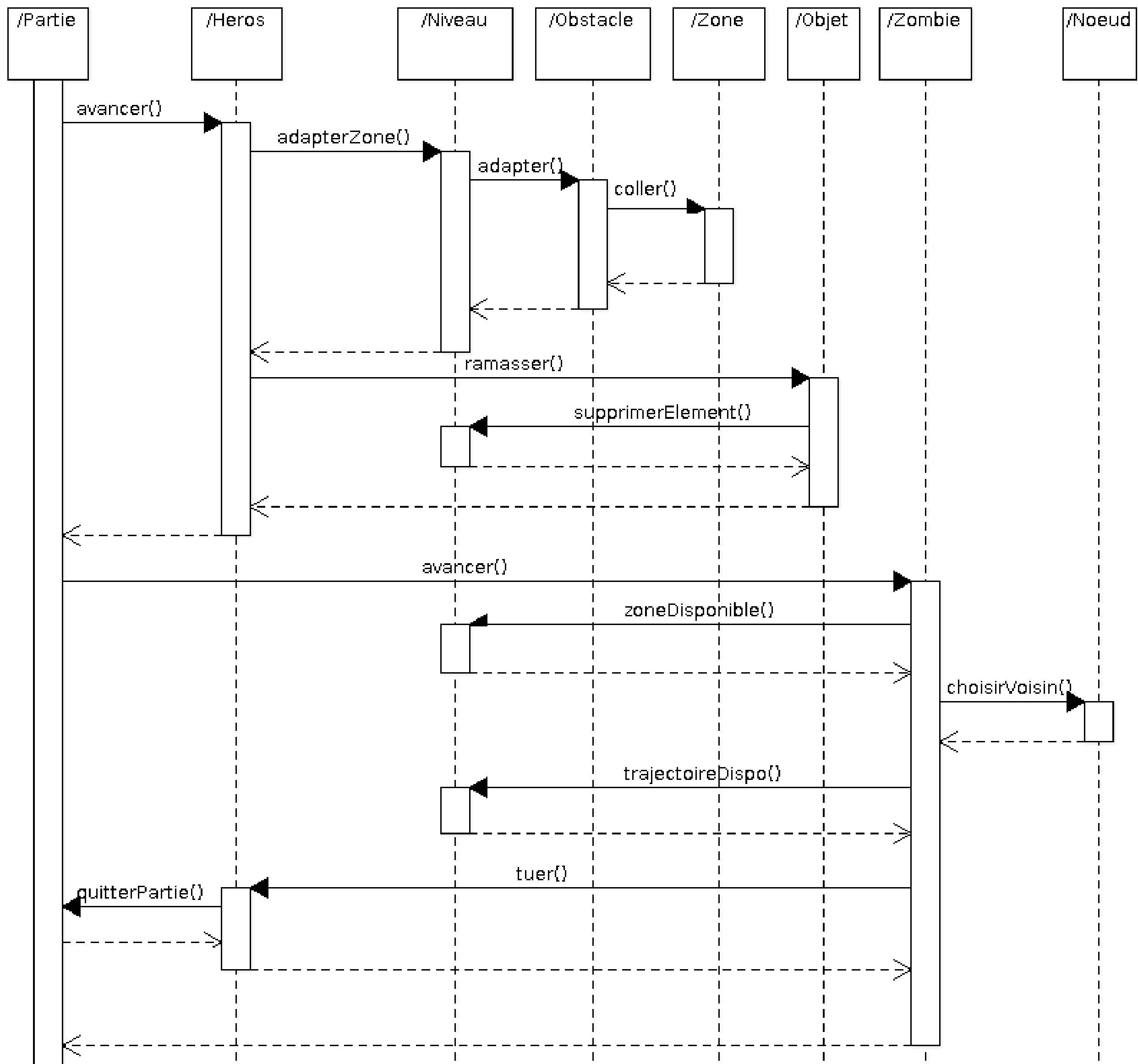
Avancement d'une partie

L'avancement d'une partie sera géré par la méthode « continuer() » de la classe Partie. Ainsi, à chaque nouvelle image, la partie devra simplement permettre :

- l'écoulement du temps
- le déplacement du héros
- le déplacement des zombies
- le recentrage de la « caméra » sur le héros

A l'intérieur des classes Heros et Zombie seront gérées les collisions avec obstacles, le comportement des zombies, le ramassage des objets... Il faudra donc optimiser au maximum toutes ces opérations car elles seront appelées de nombreuses fois par seconde.

Le déroulement simplifié de l'avancement de la partie serait le suivant :



La partie relative aux zombies étant celle qui variera le plus, car elle dépendra de la position du héros. Il faudra donc veiller à optimiser chacune des méthodes appelées pour ne pas ralentir le jeu. Il faudra d'autant plus être vigilant car l'affichage sera également très lourd.

Son

Une maigre part du développement fera place à l'utilisation du son. Le jeu étant développé en HTML5, la nouvelle balise <audio> suffira à sa gestion. Cette utilisation est déjà gérée par la bibliothèque JsAs.

Les sons seront récupérés sur le web et éventuellement retravaillés.

Images

Les images seront également récupérées sur le web (principalement de CGTextures.com) et retravaillées (redimensionnement, niveaux de couleurs...), ou créées entièrement pour le projet.

Jeu hors ligne

Le jeu devra pouvoir être joué hors ligne après un premier chargement, sans aucune interaction avec le serveur. Pour cela, un « cache manifest » sera utilisé. Le joueur n'aura donc à charger la page, les images, et les scripts et les niveaux qu'une seule fois. Il faudra cependant veiller à ce que le tout n'excède pas 5 Mo (notamment au niveau des sons).