Light | Dark

Développement d'un jeu 2D en Java

Rapport de projet

Réalisé par :
Mickaël CAMPMAS – Léo CASTERA
Jean-Joseph MARTY – Théo KRISZT

Sous la direction de :
Dr Abdelkader GOUAICH

Pour l'obtention du Diplôme Universitaire de Technologies - Informatique

Année universitaire 2013 - 2014

Remerciements

Nous tenons à remercier notre enseignant-tuteur, Dr Gouaich, pour le temps, les connaissances et l'aide précieuse qu'il nous a apportée tout au long de la réalisation de ce projet.

Nous souhaitons aussi remercier notre enseignante d'ergonomie et communication, Mme Gelsomino, pour son aide et ses conseils sur les méthodes de travail et de tenu de projet, qui nous ont permis de le réaliser dans les meilleures conditions.

Nous remercions aussi M. Stéphane Protic, artiste plasticien et enseignant à l'école d'architecture de Marseille Luminy, pour le travail et le soin qu'il a apporté à l'esthétique du jeu, mais aussi M. Luc Marty, compositeur, pour ses conseils et les ressources qu'il nous a fournis pour créer l'environnement sonore du jeu.

Enfin, nous voudrions remercier M. Alexandre Gilbert, M. Pierre-Michel Marty, Mlle Anne-Catherine Marty et Mlle Agnès Dedeire pour le temps qu'ils nous ont accordé et leurs retours lors des séances de test du jeu.

Sommaire

Table des figures

Glossaire

Artwork : Terme anglais désignant un dessin conceptuel, une ébauche d'un personnage, d'une scène ou tout autre élément d'un jeu. Ces dessins très travaillés ne sont pas à implémenter en jeu, mais permettent de définir l'orientation artistique du jeu tout en offrant un premier visuel.

Background : Dans le contexte, désigne l'ensemble des éléments de scénario constituant le passé et le caractère d'un personnage.

Core-gamer : Désigne une population de joueurs aguerris ayant une certaine connaissance du jeu vidéo. Se différencie du "hardcore-gamer" en jouant de manière régulière sans pour autant être dans l'excès.

Game design document : Document regroupant tous les détails des mécanismes d'un jeu. On y retrouve donc les règles et les lois qui régissent le jeu et qui aboutiront au gameplay.

Game design : Ensemble des mécanismes et règles régissant un jeu vidéo. L'ensemble de ces outils est ensuite écrit dans le game design document.

Game designer : Personne responsable de la création du game design et du game design document.

Gameplay : Terme difficile à expliciter. Série de règles définissant la jouabilité d'un jeu, ce qui comprend les contrôles, l'interaction entre le jeu et le joueur, les objectifs, le challenge, les récompenses...

Infiltration : Genre de jeu où la discrétion et la tactique sont de mise afin de réussir les épreuves du jeu. Le joueur peut ainsi être amené à élaborer un plan pour éviter des ennemis sans se faire repérer.

Level design : Processus de la création d'un jeu consistant à créer des scènes de jeu que l'utilisateur devra parcourir afin d'avancer dans le jeu. Ce processus tire parti des règles établies dans le document de game design.

Puzzle : Genre de jeu composé d'énigmes qui constituent le principal challenge du jeu.

RPG : Role-Playing Game ou jeu de rôle. Type de jeu reprenant dans ses mécanismes ceux des jeux du même nom sur plateau, où l'on joue avec un personnage que l'ont fait évoluer et dont certains évènements sont décidés grâce à un lancer de dés.

Tutoriel : Dans un jeu vidéo, premiers niveaux servant d'apprentissage aux mécaniques du jeu, la difficulté du jeu y est volontairement réduite. Ceci permet au joueur d'apprendre en douceur à utiliser les outils qui lui permettront d'avancer dans le jeu.

Introduction

Le but de ce projet est d'élaborer un jeu vidéo en deux dimensions, programmé en Java à l'aide de la librairie libGDX. Le jeu suivra des règles précises et essayera d'attirer le joueur en lui proposant des phases d'action et de réflexion.

Le gameplay* du jeu proposera au joueur d'incarner un personnage dont les aptitudes seront influencées par le cycle jour/nuit à travers des phases de combat ou d'infiltration* selon une orientation action/puzzle.

L'enjeu principal consiste à élaborer un gameplay innovant et indépendant du scénario, et ce malgré nos connaissances dans le domaine de la création de jeu vidéo encore peu développées. Nous avons alors suivi une méthode bien spécifique et propre à ce média. Il s'agit donc là, contrairement à la croyance populaire, de ne pas créer un jeu à partir d'un scénario, mais d'élaborer un gameplay dont le scénario permet de justifier les actions entreprises dans ce dernier. Cette composante du jeu doit donc être créée avant et séparément du scénario.

Ce projet a pour but de nous initier aux mécanismes principaux de développement d'un jeu vidéo. Il nous montre aussi les aspects vitaux de la réalisation d'un projet en groupe.

Pour réaliser un gameplay complet et original, nous avons commencé par nous documenter sur des règles existantes et méthodes de game designer. Aussi, nous nous sommes fortement inspirés de la mécanique principale des jeux de rôles, en particulier des premiers jeux de la série The Legend Of Zelda[1]. Le concept peut se résumer aux déplacements dans les huit directions et une attaque. Contrairement à la majorité des jeux de rôle, nous n'avons pas intégré de moyens directs de défense. De plus, notre personnage devait avoir une part importante de faiblesse afin d'accentuer la difficulté du jeu. Nous nous sommes donc orientés vers un gameplay en deux cœurs diamétralement opposés : l'un se basant sur l'action et l'autre se basant sur la réflexion.

Dans un premier temps nous parlerons du document de game design, fruit de cette réflexion, qui explicite ce que nous avons décidé de réaliser. Ensuite, nous expliquerons quels sont les mécanismes qui régissent le jeu, les méthodes, techniques et technologies mises en œuvre afin de réaliser ce projet. Enfin, nous présenterons notre expérience désormais enrichie en tant que développeurs de jeu vidéo grâce à l'utilisation de méthodes et d'outils de travail spécifiques.

[1] Shigeru Miyamoto, Takashi Tezuka, *The Legend Of Zelda,* Nintendo©, Février 1986

1 Cahier des charges : game design document

1.1 Introduction

Le cahier des charges d'un jeu vidéo est différent du cahier des charges d'un programme plus commun : le développement et l'analyse sont fondamentalement différents. Pour la méthode employée dans ce projet, il est notamment crucial de procéder, en premier lieu, à l'élaboration d'un document de game design* : si la croyance populaire veut qu'un jeu vidéo se construise autour d'un scénario, il peut en être tout autrement. En effet, les game designers* doivent tout d'abord imaginer les règles, les mécanismes qui régiront le jeu, tout en faisant avancer le futur joueur au sein d'une boucle dite « OCR » : Objectif – Challenge - Récompense, qui rythme le level design*, l'agencement de niveaux qui utilisent les outils mis à disposition par le game design. Le scénario viendra alors justifier le gameplay* : pourquoi le joueur doit-il réaliser ces actions et dans quel but ? Vient alors le tour de l'environnement visuel et sonore.

Sans le respect de ces règles, le projet a de grandes chances de ne jamais voir le jour ou de devenir médiocre, simplement à cause des contraintes posées par le scénario ou de l'incohérence du gameplay. Il suffit par exemple de regarder certains jeux adaptés de licences cinématographiques : il en résulte des jeux médiocres et peu amusants, disposants d'éléments de gameplay incohérents et forcés pour coller au scénario du film. Il est toutefois à noter que ce sont deux visions différentes de la création d'un jeu vidéo, un jeu peut très bien aboutir avec un scénario prévu. Il n'est donc point ici question de porter un jugement sur l'une ou l'autre de ces méthodes, mais il peut être intéressant de s'interroger sur la dimension ludique des jeux qui découlerait de chacune de ces méthodes.

Cette partie est alors construite en respectant l'ordre de ce processus créatif : game design document, level design, scénario, environnement visuel et sonore.

1.2 Gameplay

1.2.1 Population Cible

Le jeu s'adresse aux core-gamers*, une catégorie de joueurs habitués aux jeux vidéo et ayant connu les premiers jeux vidéo ou du moins connaissant les grandes licences qui ont influencé ce milieu (notamment The Legend Of Zelda[2]). L'âge de ces personnes se situerait donc entre 12 et 35 ans.

[2] Shigeru Miyamoto, Takashi Tezuka, *The Legend Of Zelda,* Nintendo©, Février 1986

1.2.2 Classification du jeu

Les principaux types sur lesquels le gameplay se repose sont action et puzzle, chacun étant relié à l'une des formes du protagoniste, mais nous reviendrons sur ce point plus en détail plus tard. Un autre type, secondaire cette fois-ci, est l'aventure, qui découlerait de l'expérience du joueur à travers les différents niveaux qu'il parcourrait. L'inspiration vient très clairement des premiers épisodes de la série The Legend Of Zelda, dont la structure est facilement reconnaissable et s'adapte parfaitement au jeu que nous souhaitons réaliser.

1.2.3 Règles et structure d'action

Les différentes actions se déroulant au cours de l'expérience de joueur se déroulent sur des écrans fixes en 2D avec vue de dessus, où le personnage rentre d'un point de l'écran et sort d'un autre.

Les grandes fonctions du gameplay* se basent donc sur deux cœurs, qui sont directement régis par les formes que peut porter le personnage :

1. Light Form : orientée action
2. Shadow Form : orientée infiltration*, réflexion

Le jeu possède un système jour/nuit, qui influera sur la forme que possède le personnage, et donc sur le gameplay. Le joueur ne pourra passer de l'un à l'autre de manière définitive, mais obtiendra tôt dans le jeu un orbe lui donnant la possibilité de passer temporairement d'une forme à l'autre en inversant le cycle en cours, avant de ramener le personnage à sa position initiale et de se désactiver pendant un certain temps. Le joueur sera ainsi amené à utiliser les deux formes dans certaines scènes, le temps limité offert par l'orbe rajoutant un certain stress et challenge, tout en apportant dynamisme et richesse à l'expérience de jeu.

❖ *Actions détachables*

Les actions réalisables par l'agent seront très reproductibles, rythmées par les formes du personnage : attaques en Light Form, déplacement d'ombres en ombres et contrôles d'animaux en Shadow Form. Mais d'autres activités seront uniques, telles que l'obtention d'une clef ou le déverrouillage d'une porte, tout en restant un mécanisme que l'on retrouve plusieurs fois dans l'aventure

❖ *Règles constitutives - Dualité Shadow Form / Light Form*

Les règles formelles qui entourent le jeu restent classiques, une perte totale de point de vie entraine la fin de la partie. Toutefois, la façon de jouer étant différente d'une forme à l'autre, elles s'adaptent à celle arborée par l'agent. L'alternance entre ces deux formes sera donc au cœur de gameplay du jeu. Apportant deux expériences différentes, la forme d'ombre étant orientée puzzle* et infiltration tandis que la forme de lumière étant plus simplement

orientée action, leur utilisation croisée seront à la base de nombreuses énigmes dont le level design* disposera. Le jeu repose sur la différence entre ces deux formes, à l'opposé l'une de l'autre, tant visuellement que sur la façon d'y jouer.

Ce sera donc l'orbe qui permettra l'alternance dynamique entre ces deux formes.

- Shadow Form

À l'aube, des ombres apparaitront çà et là à l'écran, permettant au joueur, alors transformé en Shadow Form, de se déplacer jusqu'à sa destination. Pour cela, l'agent dispose d'un grappin lui permettant de se déplacer d'ombre en ombre ainsi que d'une capacité lui permettant, en se déplaçant sur l'ombre d'un animal, de le contrôler. L'agent ne pourra toutefois pas se déplacer en dehors d'une ombre.

Les trois verbes d'action sont donc:

- Se déplacer
- S'infiltrer (discrétion et reconnaissance)
- Contrôler (les animaux)

En Shadow Form, le personnage est faible, un coup d'un ennemi suffirait à le tuer. Le joueur peut donc être détecté et éliminé par des ennemis s'il se déplace d'une ombre à l'autre dans le champ de vision de l'ennemi (dont la taille peut différer selon la qualité de ce dernier). Le joueur doit donc réfléchir et faire preuve d'ingéniosité pour se déplacer d'un point A à un point B sans se faire repérer. Pour cela, il peut avoir recours au contrôle d'animaux qui peuvent permettre au joueur de passer outre la surveillance d'une sentinelle.

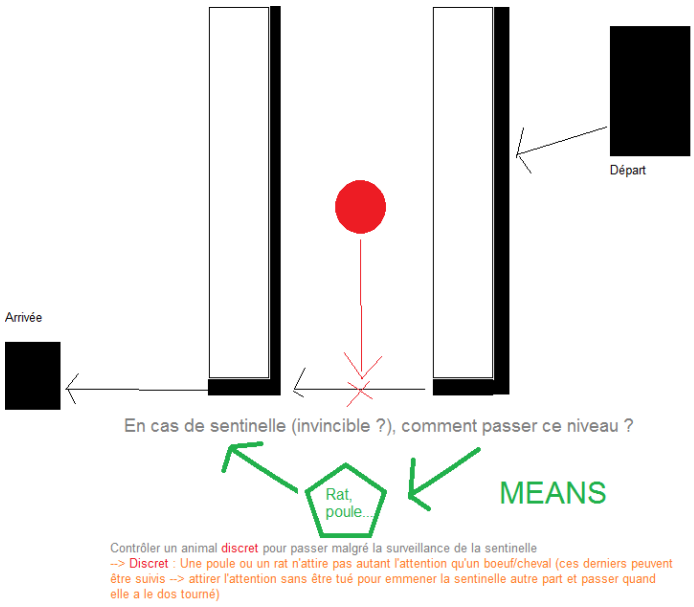


Figure 1 - Schéma type d'un niveau

Les animaux terrestres possèdent, en plus de pouvoir se déplacer, une capacité spéciale : un coq peut se mettre à chanter et donc attirer les ennemis à l'endroit où il se trouve, les obligeant à se déplacer. Une salamandre, elle, peut allumer une torche, le sanglier peut détruire une barrière en chargeant. Un poisson, lui, sera un « moyen de transport ». Comprenons par là qu'il ne permet que de se déplacer d'un point à un autre : le poisson remonte une rivière et s'arrête à un point précis.

La Shadow Form étant immatérielle, elle peut se jouer de certains obstacles : une grille, plongée dans la pénombre, pourra être traversée sans problèmes tandis que la Light Form en sera incapable. Elle ne pourra toutefois pas traverser certains passages où les ombres sont hors de portée ou quand le combat est inévitable.

- Light Form

Lors de la première apparition de cette forme, le joueur devra trouver une épée afin de disposer des capacités offensives de cette forme. La Light Form est à l'opposé de la Shadow Form : résolument orientée action, elle permet au joueur d'attaquer ses ennemis au corps à corps dans la direction voulue en appuyant rapidement sur la touche d'attaque, ou bien de charger une attaque en restant appuyé une à deux secondes de plus. Cette dernière attaque permet de relâcher un projectile unidirectionnel, arrêté par tout obstacle, ennemi ou décor, mais possède une plus grande efficacité que l'attaque au corps à corps. Cette forme permettra au joueur de « décompresser », de se défouler après des énigmes complexes avec la Shadow Form, mais elle lui sera aussi complémentaire afin de déloger des ennemis dont la Shadow Form ne peut se débarrasser. Elle apporte ainsi une nouvelle dimension de gameplay*.

Elle se définit donc en deux verbes :

- Se déplacer
- Attaquer

❖ *Orbe et utilisation croisée des formes*

Tôt dans l'histoire (vers la scène cinq environ), le joueur obtiendra un orbe lui donnant la possibilité de changer temporairement de cycle avant de se désactiver pendant un temps (40 secondes), de la même manière que les sables du temps de Prince Of Persia[3].

Le temps d'utilisation étant limité et son utilisation affublée d'un temps de rechargement, le joueur devra utiliser cet objet à bon escient. Le temps limité impliquera une dose de stress, qui dynamisera le gameplay et offrira un certain challenge.

Certaines scènes ne pourront pas être passées sans utiliser les deux formes. Le joueur devra donc œuvrer avec les propriétés de ces dernières pour surmonter certains obstacles. Une scène pourrait par exemple demander au joueur de passer la surveillance d'ennemis trop nombreux en Shadow Form pour ensuite arriver à une salle close qui nécessitera la Light Form pour se débarrasser d'un garde gênant puis pour abaisser un levier qui pourrait ouvrir une autre salle ou un barrage permettant la progression du joueur.

[3] Patrice Désilets, Jordan Mechner, Yannis Mallat, *Prince of Persia : The Sands Of Time,* Ubisoft™, 2003

❖ *Buts*

Afin d'encourager le joueur à avancer, il est important de lui fournir des buts variés sans dénaturer le gameplay ou le perdre dans une multitude d'objectifs dans laquelle il n'arrive plus à se retrouver : nous entrons ici dans la boucle OCR (Objectif, Challenge, Récompenses) que le joueur suit durant toute sa partie sans forcément s'en rendre compte.

Ces objectifs se doivent d'être ludiques et atteignables par l'agent en utilisant les moyens mis à sa disposition, mais aussi clairs et simples à comprendre. Un jeu aux mécanismes et buts trop flous risque de rebuter le joueur dès les 30 premières secondes. Si le joueur ne parvient pas à aimer le jeu dès ces 30 secondes, le joueur abandonne le plus souvent le jeu.

Lorsque le joueur arrive sur une scène de Light | Dark, son objectif premier est d'atteindre la sortie située autre part sur l'écran : nous avons ici un simple schéma d'un déplacement d'un point A à un point B. Toutefois, l'agent ne peut atteindre l'objectif immédiatement, car il doit suivre les règles explicitées ci-dessus et franchir certains obstacles qui apparaitront devant lui (ennemis, énigmes…). Les actions qu'il entreprend pour les surmonter constituent déjà de plus petits objectifs progressifs qu'il devra remplir pour accomplir le plus grand. Pour cela, certaines scènes demanderont donc au joueur d'effectuer des actions spécifiques bien qu'il n'y ait pas forcément d'ordre précis. Parmi ces actions on retrouve le déplacement d'ombre en ombre, le fait d'éviter un ennemi en Shadow Form ou de l'attaquer en Light Form et enfin, contrôler un animal pour déclencher certains mécanismes (attirer l'attention d'un ennemi à un endroit précis, allumer une torche…).

Ces mécanismes constituent des retardataires : ils ralentissent la progression du joueur afin qu'ils représentent un challenge pour le joueur. Certains d'entre eux ne seront franchissables qu'avec la Shadow Form en tirant parti des propriétés de cette dernière :

- Une barrière (grillage, portail) plongée dans la pénombre ne peut être franchie qu'avec cette forme, étant complètement immatérielle.
- Contrôler certains animaux permettra de briser un obstacle comme un rocher (sanglier) ou de traverser un point d'eau (poisson)

À contrario, certaines ombres étant trop éloignées, ou les animaux étant absents, seule la Light Form pourra passer, en combattant certains ennemis, attirés par la lumière qu'émet l'agent. Ces derniers représentent une menace tant en Light Form qu'en Shadow Form et sont l'un des principaux obstacles, retardataires et challenges du jeu.

❖ *Boucle OCR + Means*

Énoncée précédemment, la boule OCR, pour Objectif-Challenge-Récompense, est une boucle qui régit la progression du joueur dans le jeu. L'oubli d'un de ses éléments peut gravement nuire à la qualité du jeu final, et serait une grave erreur de game design.

Pour le jeu final, la boucle OCR de Light | Dark se présenterait comme ceci :

Objectif : Obtenir un nouveau fragment de l'orbe en notre possession se trouvant à la fin d'une zone.

Challenge : Énigmes, puzzles, obstacles, ennemis et retardataires. Potentiellement un monstre plus puissant gardant ledit fragment.

Récompense : Le joueur obtient un nouveau fragment lui offrant plus de temps d'utilisation de l'orbe. Elle donne aussi un indice sur la nouvelle zone à explorer, et donc l'accès à une nouvelle boucle OCR.

❖ *Actions vivantes, facteurs d'échecs*

Toutefois, pour éviter que le joueur ne se sente pris dans un « couloir » et qu'il ait l'impression que toutes ses actions sont prévues, le jeu doit donner une certaine liberté à ses actions et donc lui permettre de manquer certains objectifs ou de les réaliser dans l'ordre qu'il le souhaite. C'est notamment la raison pour laquelle l'orbe existe et que le jeu ne possède pas un cycle jour/nuit régulier qui aurait pu freiner le joueur à de nombreuses reprises dans son avancée. Afin de ne pas trop simplifier le jeu, l'orbe possède un temps de rechargement assez long pour amener le joueur à réfléchir avant d'agir, sans toutefois être contraignant.

En plus des retardataires, le jeu possède un mécanisme simple qui encourage le joueur à effectuer ces actions correctement : la mort. En Light Form, elle se traduit par une perte de tous ses points de vie. En Shadow Form, être simplement détecté et touché par un ennemi suffit. Lorsque cela arrive, le personnage réapparait à sa position de départ dans la scène, ce qui signifie qu'il devra répéter les actions précédentes pour revenir à l'endroit où il se trouvait avant de mourir. La mort ici n'est donc pas définitive, mais reste une contrainte.

1.2.4 Interactions entre le joueur et le micro monde

❖ *Le système de sauvegarde*

Afin que le joueur ne perde pas sa progression entre chaque session de jeu, le programme possède un système de sauvegarde qui mémorise automatiquement la position de départ de la dernière scène traversée.

❖ *Contrôles*

Le jeu dispose de deux configurations différentes selon le support utilisé (ordinateur ou alors sur un support Android), mais les contrôles sont simplement composés de touches pour le déplacement et d'un moyen et pointage et de clics (souris ou doigt) pour les attaques et les actions.

❖ *Caméras*

Vue de dessus, la caméra est fixe et le personnage se déplace d'un bord à l'autre de l'écran, chaque scène dispose d'une caméra ou « point de vue » fixe et lorsque le personnage change de scène, il en est de même pour la caméra.

❖ *Personnages*

Pour le prototype, le jeu possède trois types de personnages :

- L'agent, possédant deux formes
- Les animaux, pouvant être contrôlés par l'agent
- Les ennemis, hostiles à l'agent et étant un des principaux obstacles

Plus tard, des PNJ* seront ajoutés au jeu afin de renforcer la narration.

❖ *Feedback*

Les feedbacks sont des retours sur actions de l'agent. Lorsqu'une action s'effectue à l'écran, un retour doit être apporté, il peut être d'ordre visuel et/ou sonore, afin d'avertir le joueur d'une action et de confirmer que l'action a bien été réalisée. Pour Light | Dark, ces feedbacks concernent :

- Les dégâts reçus par le personnage : lorsque celui-ci subit une attaque, son apparence clignote et est repoussé.
- Lorsque l'orbe est utilisé, son apparence est grisée jusqu'à son rechargement
- D'autres retours visuels et sonores sur les actions des acteurs (attaquer, changer d'ombre…)

❖ *Assistance*

Afin de ne pas perdre le joueur durant ses sessions de jeu et de lui faire comprendre les mécaniques du jeu, une assistance, explicite ou non, doit accompagner le joueur dans son expérience. Elle peut aussi se manifester par l'architecture des niveaux : c'est l'approche que nous avons choisie pour ce prototype, nous y reviendrons dans le level design.

Au-delà du projet, des panneaux d'indications et des menus contextuels pour les dialogues permettront d'orienter le joueur et de renforcer la narration.

❖ *Menus*

Les menus du jeu restent assez classiques. On retrouve donc les traditionnels menus de démarrage, permettant de commencer une nouvelle partie ou d'en continuer une précédente, et le menu pause, utilisable durant une partie.

1.3 Level design

1.3.1 Création de scènes pour le jeu

Le level design* constitue l'ensemble des scènes dans lequel l'agent progresse, il tire parti des outils créés et mis à disposition par le game design.

Pour ce prototype qui est composé des premiers niveaux d'apprentissage, plus souvent appelés tutoriel*, nous avons choisi l'approche qu'Edmund McMillen présente dans Indie Game : The Movie[4] : chacun des premiers niveaux est volontairement simpliste et sert à exploiter une seule mécanique. Ceci permet au joueur d'assimiler pleinement cette technique avant de passer à des niveaux qui combineront cette dernière avec d'autres mécanismes qui auront eux aussi profité d'un niveau semblable.

Les captures d'écran qui suivent sont toutefois des apparences temporaires, la version du jeu présentée lors de la soutenance de projet pourrait posséder une apparence plus soignée, notamment en ce qui concerne l'apparence des différents acteurs et des décors. Il en va de même pour l'identité sonore du jeu. Ces images permettent toutefois d'illustrer plus clairement le propos.

Pour ce premier niveau, nous avons choisi d'apprendre au joueur le premier des mouvements de la Shadow Form : le déplacement d'ombre en ombre. Le niveau est simplement composé d'ombres sur lesquelles il doit se déplacer et s'agripper. Ce niveau doit aussi lui permettre de connaitre la longueur du grappin grâce au positionnement d'ombres parfois situées trop loin du personnage et qui amènera le joueur à trouver un chemin alternatif.

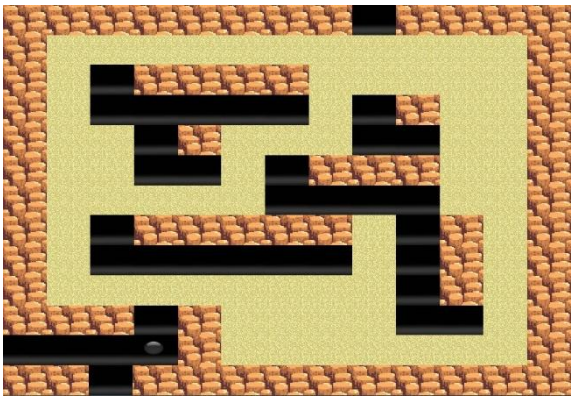


Figure 2 – Capture d'écran du niveau 1

[4] James Swirsky, Lisanne Pajot, *Indie Game : The Movie*, Film documentaire, Gowdy Manor Productions, 2012, 96 minutes

Le deuxième niveau met le joueur pour la première fois face à des ennemis hostiles, qui l'attaqueront immédiatement s'il venait à se déplacer devant eux.

Si le chemin le plus court mène donc irrémédiablement à l'échec, un chemin alternatif plus long l'amène vers un animal passant devant l'ennemi. Il peut donc contrôler cet animal pour passer furtivement devant le monstre et arriver à la sortie.

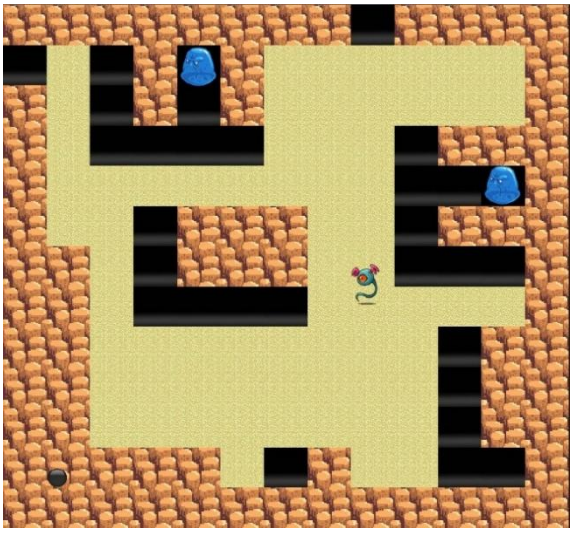


Figure 3 – Capture d'écran du niveau 2

Pour ce troisième niveau, un ennemi bloque l'accès à la sortie : il faudra donc trouver un moyen de l'en déloger.

Le joueur peut apercevoir un coq à proximité de cet ennemi, et grâce à l'expérience acquise au niveau précédent, il sait qu'il peut contrôler cet animal pour progresser.

C'est en effectuant cette action qu'il apprendra qu'il peut utiliser une compétence propre à l'animal (dans le cas ici présent, chanter), qui peut attirer l'ennemi et donc dégager l'entrée.

Figure 4 – Capture d'écran du niveau 3

En arrivant dans le quatrième niveau, la nuit est tombée sur le micro monde* et le personnage change de forme.

Désormais en Light Form, le personnage peut se déplacer à sa guise sur la carte et possède une nouvelle jauge de santé. Il doit alors se mouvoir, en évitant l'attention d'un garde, pour récupérer une épée dans un coffre. Cette épée lui permettra enfin de se défendre contre ses ennemis, dont celui qui gardait l'arme en question.

Figure 5 – Capture d'écran du niveau 4

Toujours en Light Form à l'entrée du cinquième niveau, trois ennemis aperçoivent de loin le personnage et se mettent à l'attaquer : le joueur comprend alors qu'il est fortement visible sous cette forme.

En se débarrassant de ces ennemis, il accède au coffre que ces derniers gardaient et récupère l'orbe.

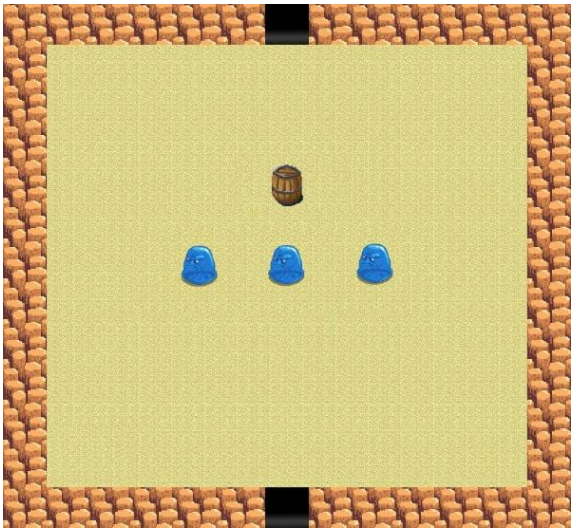


Figure 6 - Capture d'écran du niveau 5

Enfin, le dernier niveau tire parti de tous les mécanismes appris grâce aux niveaux précédents. Le personnage arrive en Shadow Form et doit allumer deux brasiers pour déverrouiller la porte. Pour cela, il peut se déplacer derrière la grille, utiliser l'orbe pour déclencher la Light Form, éliminer les ennemis, abaisser le levier pour ouvrir le passage à la salamandre que le joueur pourra contrôler en revenant en Shadow Form pour allumer les deux brasiers à proximité de la porte.

Figure 7 – Capture d'écran du niveau 6

1.3.2 Vérifier la cohérence du gameplay par le jeu de plateau

Une bonne pratique pour vérifier la cohérence et la fiabilité du gameplay* et du level design* consiste à adapter le jeu et les niveaux en jeu de plateau. En faisant essayer ce jeu réalisé avec du papier, des pièces de jeu d'échecs et des dessins à une tierce personne, on peut vérifier que les mécanismes sont facilement assimilés et compréhensibles.

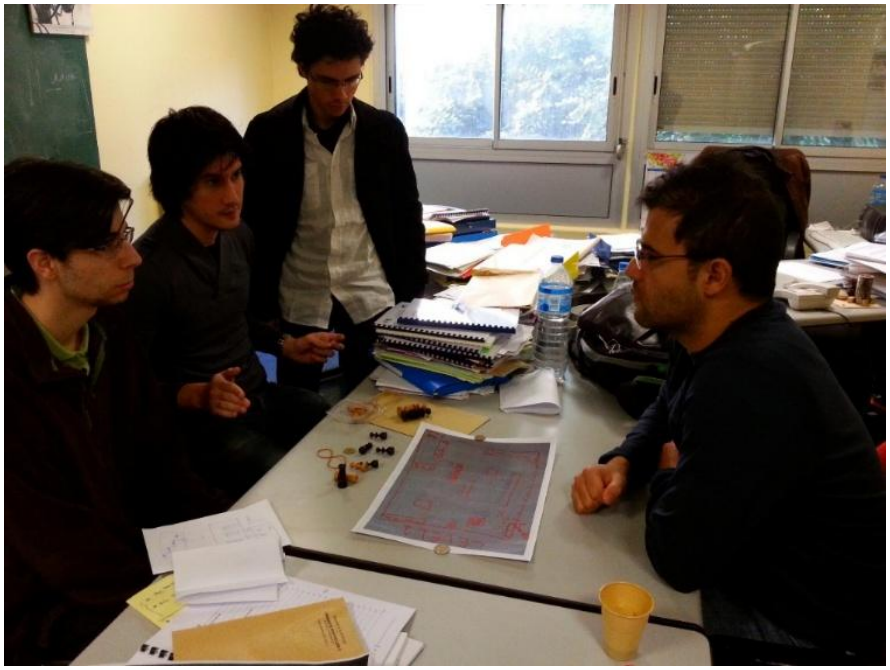


Figure 8 - Présentation du concept au tuteur

Figure 9 - Début d'une session de jeu

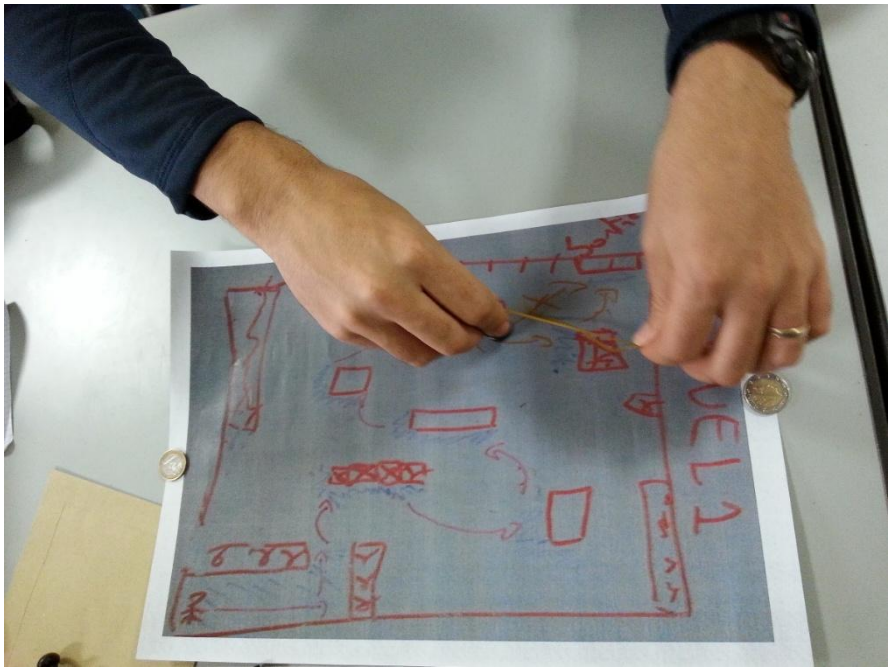


Figure 10 - Découverte du grappin

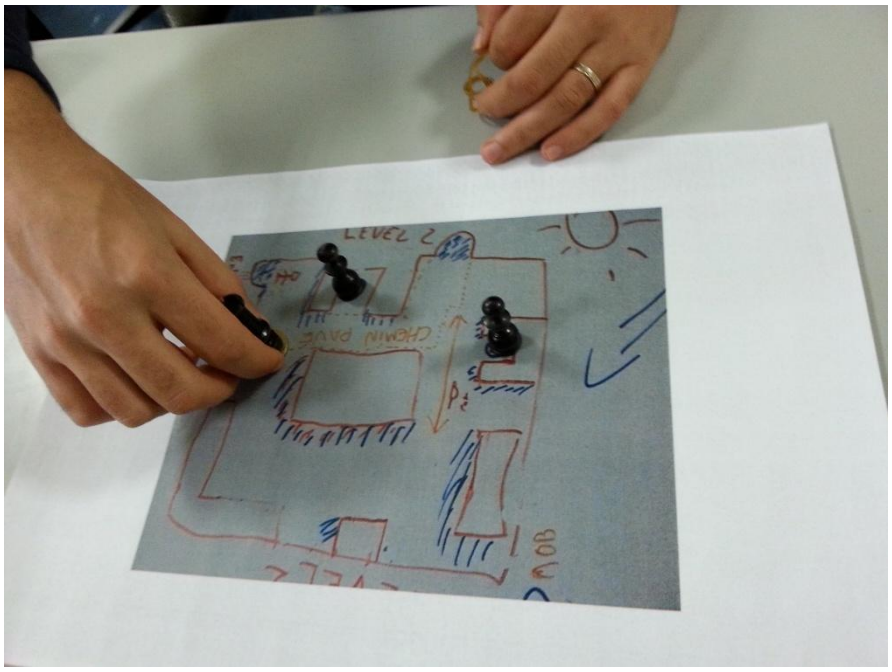


Figure 11 - Niveau 2 et premier contact avec les ennemis

Figure 12 - Découverte de la Light Form

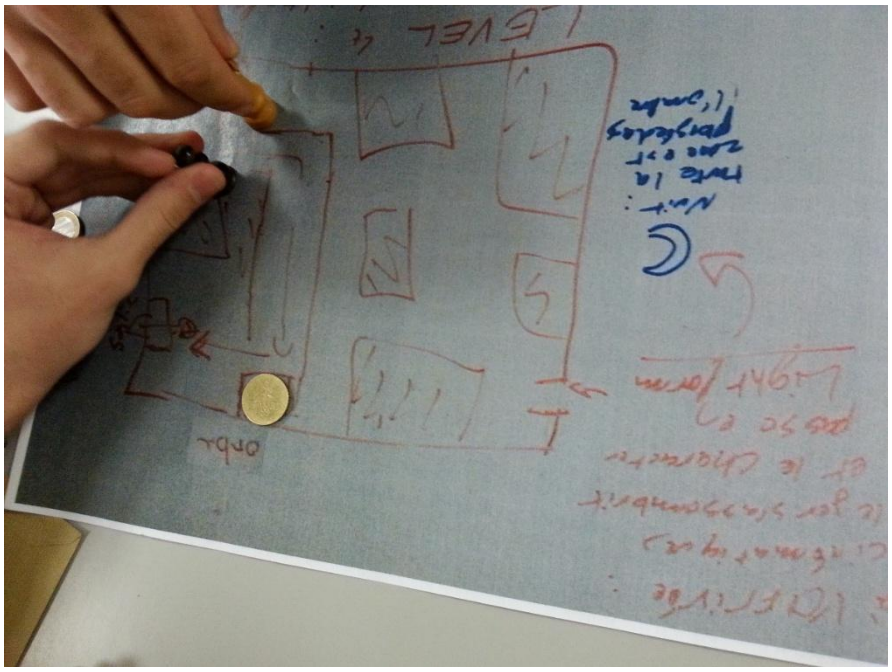


Figure 13 – Essai du quatrième niveau

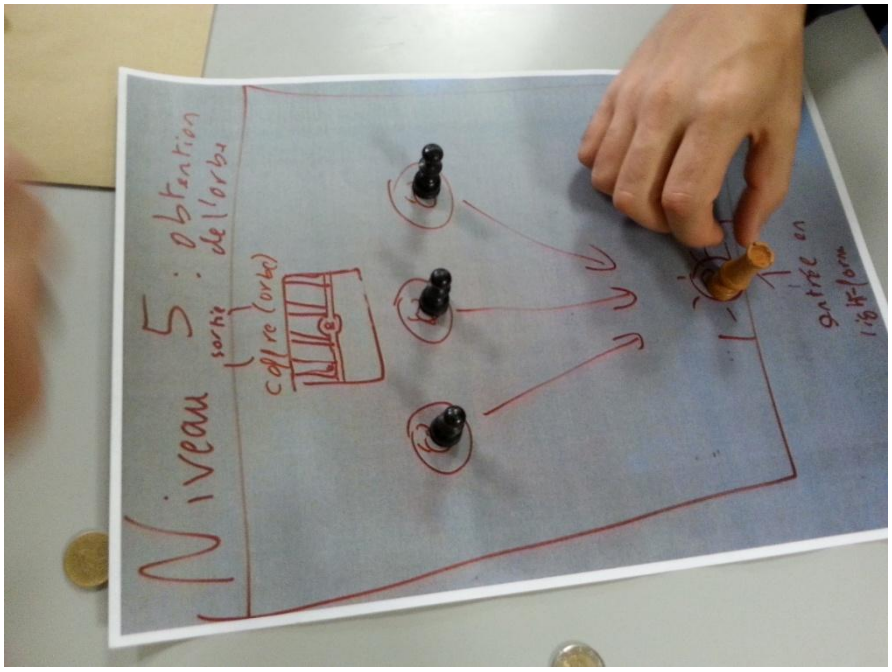


Figure 14 - Cinquième niveau et accès à l'orbe

Figure 15 - Dernier niveau et utilisation de l'orbe

1.4 Scénario

Comme énoncé précédemment, le scénario se doit d'être créé après avoir imaginé le gameplay*. Le scénario permet donc de justifier les éléments de ce dernier et non de le créer, un jeu vidéo peut d'ailleurs exister avec un scénario des plus simples voire sans scénario, sans que l'expérience de jeu n'en soit altérée.

Le scénario peut toutefois aider non seulement à captiver l'attention du joueur et l'amener à ses objectifs, mais aussi à imaginer les scènes qui composent le level design, ce dernier utilisant les outils mis à disposition par le game design.

Pour Light | Dark, nous avons choisi un scénario simpliste, mais permettant de justifier la totalité des éléments de gameplay*, tout en fournissant un background* crédible pour l'intégration des différents acteurs et la progression du joueur.

« Light | Dark met en scène les Nocten, peuple humanoïde vénérant l'orbe de lune, seul garant de l'équilibre entre la nuit et le jour, la lumière et les ténèbres. À son contact, le corps de ses gardiens a changé et s'adapte au moment de la journée, leur permettant de se cacher du reste du monde : de jour, ils arborent une forme lumineuse, éblouissante et de nuit, ils sont des ombres parfaitement invisibles.

Vous incarnez Ekhan, un Nocten dont l'horloge biologique est inversée par rapport à celle de ses congénères, et ce depuis sa naissance, ce qui lui a valu d'être rejeté par son propre peuple et d'être le premier accusé lorsque l'orbe disparut après avoir été brisé.

Banni puis chassé par les Nocten, il se retrouve ainsi seul dans un monde désormais instable où l'alternance entre le jour et la nuit semble complètement soumise au hasard. Dans sa fuite, Ekhan trouvera un fragment de l'artefact perdu par son peuple. En suivant les traces laissées par celui qui semble être le véritable coupable, Ekhan s'engagera dans une quête pour restaurer la relique sacrée et ainsi rétablir l'équilibre du monde. »

L'objectif principal du jeu sera ainsi de récupérer les tessons de l'orbe, disséminés çà et là par le voleur. À chaque fois que le joueur se rapprochera d'un fragment, il devra se confronter à un monstre plus puissant, par exemple un ennemi corrompu par l'essence de l'orbe. Une fois vaincu, le joueur obtient un nouveau fragment, réduisant le temps de rechargement de l'orbe tout en donnant un indice sur la position du voleur, et donc d'un nouveau fragment. Le dernier fragment le mènera irrémédiablement vers cet ennemi afin d'en comprendre les motivations, mais aussi de découvrir la véritable nature du personnage : pourquoi est-il le seul de sa race à être inversé ? On retrouve d'ailleurs ici le concept du monomythe et le schéma universel du voyage du héros aux mille et un visages.[5]

[5] Joseph Campbell, *Le héros aux milles et un visages*, Etats-Unis, Pantheon Books, 1949

Le scénario ci-dessus permet non seulement de fournir une raison à la présence des deux formes, de l'orbe, mais permet aussi de fournir au joueur un but plus significatif que de simplement récupérer des objets se trouvant à la fin de donjons. L'écriture d'un tel scénario n'a à aucun moment modifié le document de game design* et permet déjà d'entrevoir le cheminement entrepris par le level design.

1.5 Environnement graphique

Nous avons choisi pour le prototype de nous servir de banques d'images libres de droits pour concevoir les décors et les ennemis. L'orientation du style de dessin se rapproche donc de ceux utilisés dans de nombreux RPG* en deux dimensions : le medieval-fantasy.

Figure 16 - Artwork Light Form

Figure 17 - Artwork Shadow Form

En ce qui concerne le design futur du jeu, nous avons toutefois tenu à demander les services d'un artiste plasticien et de réaliser quelques artworks* pour le projet. Ce professionnel s'est également proposé pour réaliser le reste de l'esthétique du jeu au-delà du projet, le temps lui manquant durant la période de développement du prototype pour réaliser la totalité des éléments et animations du jeu.

L'une des principales difficultés pour cette tâche est la communication avec le graphiste : les corps de métier étant complètement différents, les hésitations et incompréhensions sont fréquentes. La première chose à effectuer est donc de bien convenir des éléments voulus, du style de dessin, des inspirations possibles, de l'allure générale du personnage et des éléments que l'on souhaiterait apparaitre sur celui-ci.

Dans le cas présent, il s'agit d'avoir une influence médiévale, d'un effet dessiné bien présent et de faire figurer des armes et outils sur chacune des formes : épée pour la Light Form, grappin pour la Shadow Form. Le scénario peut aussi aider le graphiste à trouver les éléments de dessins que nous aurions pu oublier, grâce au passif du personnage, l'environnement dans lequel il évolue, etc.

Un contact fréquent avec celui-ci reste aussi primordial afin de s'assurer que le résultat obtenu correspond bien à nos attentes ou de le corriger avant qu'il ne soit trop tard.

Nous incluons ici uniquement des artworks* et des ébauches, le temps de réalisation de dessins pour les animations et déplacements du personnage étant beaucoup trop long pour pouvoir être adapté au prototype dans le temps imparti. Ceux-ci sont visibles en annexes 7 à 11.

1.6 Environnement sonore

Pour l'environnement sonore, nous avons aussi choisi de nous servir de sons et musiques libres de droits, mais nous pourrons aussi produire nous-mêmes certains bruitages (épée, bruits de pas…).

Nous suivrons aussi les conseils d'un compositeur professionnel afin de réaliser une ambiance sonore plus crédible et adaptée.

1.7 Idées abandonnées

L'écriture du document de game design* nous a amenés à revoir certaines de nos idées voire à en abandonner. La raison est toujours la même, nous avions parfois conçu des mécanismes qui complexifiaient inutilement le gameplay*. C'est une expérience très fréquente lorsque l'on cherche à aller plus vite qu'il ne le faudrait, en essayant par exemple d'imaginer le level design* avant de terminer la réalisation des règles de gameplay.

Voici une liste non exhaustive des idées que nous avons abandonnées définitivement :

- Un cycle jour/nuit de périodes à durées égales : ceci aurait nui à la progression et au confort du joueur qui aurait dû parfois attendre inutilement le passage d'un cycle à l'autre.

- Tuer un monstre en se glissant dans son ombre en Shadow Form : cela semblait incohérent et nous avions opté pour une forme discrète, mais incapable de se défendre. Par ailleurs, cela aurait réduit l'intérêt de la Light Form.

- Un système monétaire permettant d'acheter équipements et potions : une idée qui avait été proposée lorsque le jeu avait une influence RPG*. Nous l'avons abandonné lorsque le choix du genre du jeu a été fixé, le RPG nous semblant alors inutilement lourd pour Light | Dark.

- L'obtention d'attaques spéciales pour la Light Form et la possibilité de contrôler des animaux spécifiques pour la Shadow Form à la fin d'un donjon : cette idée a été abandonnée, car elle comportait trop de fioritures qui rendaient les contrôles et la maniabilité trop complexe. Par ailleurs, elle montre que nous

avions cherché à imaginer le level design avant d'avoir terminé d'imaginer le gameplay.

Il y a toutefois une leçon à en tirer : à trop vouloir bien faire, on peut être amené à créer des éléments inutiles. Mieux vaut rester sur une base simple, claire et cohérente qui permettra une plus grande richesse par la suite.

2 Rapport technique

2.1 Technologies utilisées

Le langage Java est le langage qui nous est demandé par le cahier des charges. De plus ce langage, créé en 1995, est particulièrement portable ce qui est un atout dans la fabrication de produits multiplateformes. La programmation orientée objet est omniprésente sous Java, ce qui permet généralement de simplifier la relecture du code. Enfin, cette popularité dont Java bénéficie explique la documentation particulièrement riche en explications et en exemples.

La bibliothèque libGDX est une liste de fonctionnalités graphiques qui permettent de piloter la librairie OpenGL. L'avantage principal de libGDX réside dans la simplification de la programmation graphique et de méthodes de gestion d'acteurs liées au jeu vidéo. Tout comme le Java, libGDX est compatible sur plusieurs plates-formes. Bien qu'au début nous devions utiliser GAME, une surcouche de libGDX développé par le LIRMM, nous avons choisis libGDX pour sa diversité en fonctionnalité et sa légèreté et afin d'avoir une meilleure maitrise du code.

2.2 Conception de Light | Dark

2.2.1 Les collisions et interactions physiques

Dans le micro monde du jeu, il existe différents types de cases. En effet, le joueur ne peut passer au travers des obstacles ou ne pas subir les inconvénients d'une marre de boue. De plus, notre game design document* précise que le joueur doit pouvoir se déplacer dans une zone d'ombre s'il est en forme d'ombre.

- *Les cases bloquantes*

Ce type de case, comme son nom l'indique, empêche le joueur de passer au travers, elles représentent donc un obstacle.

Nous modélisons la gestion des obstacles dans le contrôleur du joueur, ainsi que dans les contrôleurs des différents acteurs tels que les monstres, les animaux et les projectiles. De cette façon le micro monde de Light | Dark peut imposer des contraintes sur ses différents acteurs en agissant directement sur leur comportement.

- *Les cases ombres*

Ces cases propres à la Shadow Form permettent au joueur de s'infiltrer dans le niveau tout en le protégeant. Les monstres ne peuvent effectivement pas détecter le joueur en forme d'ombre si celui-ci se trouve dans une case de type ombre. Contrairement aux monstres, les animaux peuvent se déplacer dans les cases ombres.

❖ *Les cases à friction*

Ces cases ont une propriété mitigée puisqu'elle se situe sur la frontière entre le refus absolu et la pleine acceptation. La plupart imposent un changement notable dans la vitesse de déplacement de l'acteur.

2.2.2 La Shadow Form

Cette forme représente la forme faible du joueur. Elle se caractérise par l'impossibilité de se trouver au contact de la lumière et par des moyens spéciaux de déplacement tel que le grappin. Une des spécificités de la Shadow Form est la possibilité de contrôler un animal, et ce même si ce dernier se déplace.

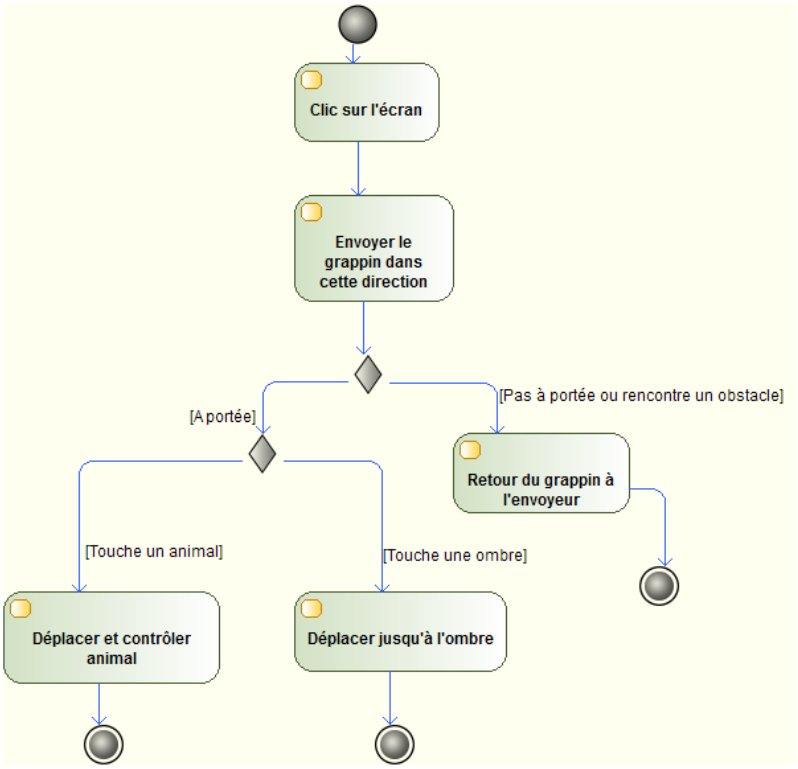


Figure 18 - Diagramme d'activité du grappin

Ce diagramme montre bien toutes les possibilités de cette forme. Comme indiqué, le joueur ne peut pas sélectionner une ombre ou un animal qui est hors de portée. Cependant, il est possible que le joueur se retrouve dans une situation où il est positionné dans la lumière et c'est à ce moment que le programme redirige le joueur dans une ombre. Cette fonction est importante, car les montres ne peuvent pas voir le joueur en forme d'ombre que lorsqu'il traverse une zone de lumière.

2.2.3 La Light Form

Cette forme est simple puisqu'elle contient les déplacements de la Shadow Form sans les limitations ce cette dernière. De plus, cette forme est dotée de capacités offensives

composées de deux types d'attaques différentes. En effet, l'épée permet une attaque au corps à corps contrairement à l'attaque à distance qui permet de toucher un monstre à une certaine portée.

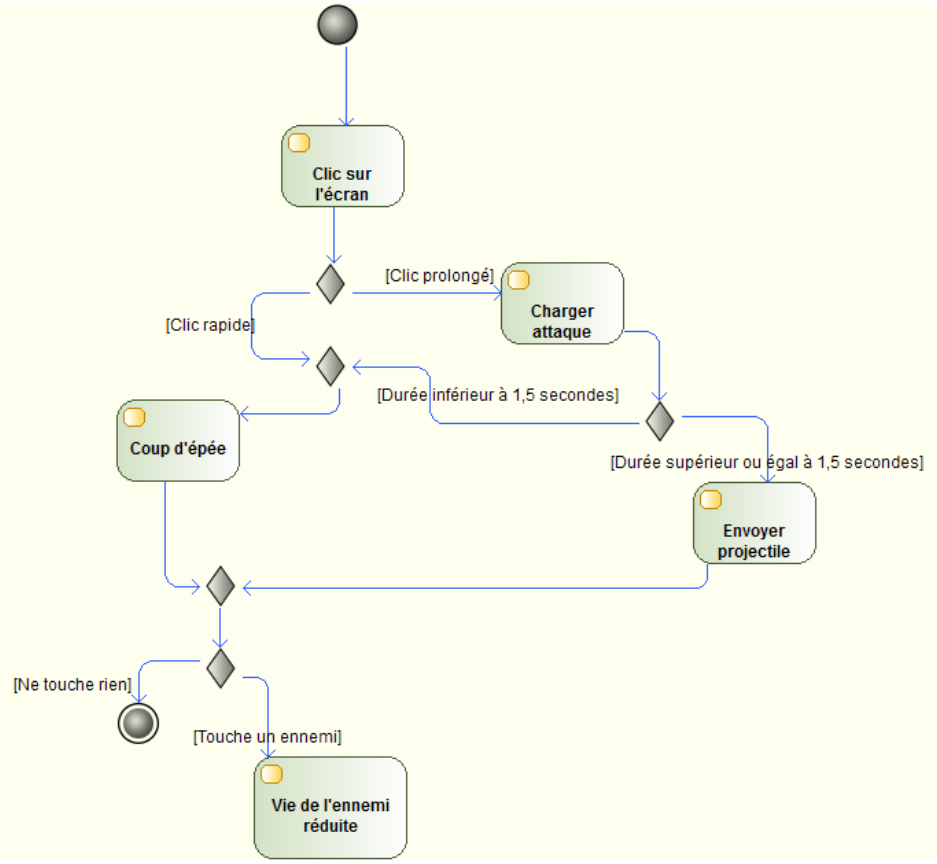


Figure 19 - Diagramme d'activité des attaques de la Light Form

Cette forme n'est pas la plus complexe du gameplay* puisqu'elle ne permet que de débloquer certains niveaux par la victoire du joueur sur les monstres.

2.2.4 Les monstres

Cette catégorie d'acteurs est la seule qui puisse interagir sur le nombre de points de vie du joueur. Les monstres sont particulièrement sensibles à l'obscurité et ils ne peuvent traverser les cases ombres, dans lesquelles ils ne peuvent pas voir. Ils peuvent être détruits par le joueur si celui-ci les tue avec ses moyens de défense en Light Form. Ce fonctionnement est visible sur l'annexe 6.

Les monstres restent fixes ou bien se déplacent de manière linéaire en suivant un parcours prédéfini qu'ils exécutent en boucle. Dans le cas où le joueur est détecté par ces derniers, ils peuvent sortir de leurs parcours pour attaquer le joueur. S'ils sont toujours en vie à la fin de l'altercation, ils rejoignent alors leurs parcours pour le continuer.

2.2.5 Les animaux

Les animaux sont complètement passifs par rapport au joueur qui peut les contrôler, mais pas les tuer. Les animaux suivent un parcours prédéfini qu'ils répètent en boucle.

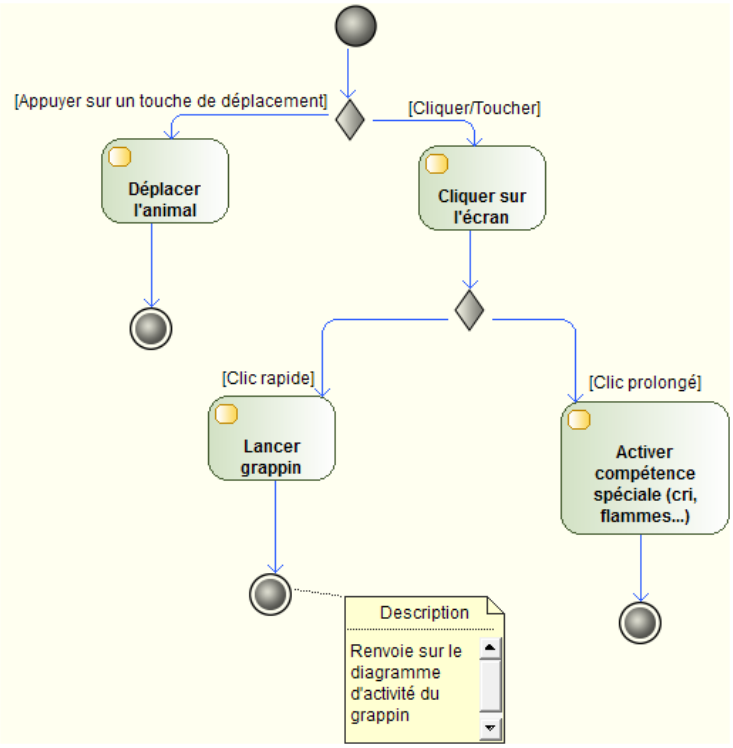


Figure 20 - Diagramme d'activité de contrôle d'animal

S'ils sortent de leur parcours, l'intelligence artificielle de l'animal leur permet de revenir sur leur parcours et de le continuer indéfiniment.

2.3 Les Algorithmes spécifiques

2.3.1 Le système de coordonnées de libGDX

LibGDX n'utilise pas un système de coordonnées par pixel bien que ce dernier est historiquement universel dans le monde des librairies graphiques et dans le monde des moteurs de jeu. Le système de coordonnées proposé par libGDX est assez générique. En effet, libGDX gère l'affichage par un système de flottant (float), ce genre de nombre permet le calcul des nombres rationnels sur une grande précision ce qui permet de proposer un système universel basé sur des unités virtuelles. Ainsi, le développeur décide sur combien d'unités il définit son écran, puis se positionne en fonction du système de coordonnées qu'il a défini. L'avantage indéniable que ce système apporte est son indépendance par rapport aux dimensions de l'écran.

De plus, un point se définit sous libGDX par un vecteur. La programmation graphique s'en trouve considérablement simplifiée notamment dans les calculs d'angles et de collisions.

Figure 21 - Représentation d'un vecteur symbolisant un déplacement vers le haut à droite

2.3.2 Les équations de conversion

❖ *Passage des pixels à la matrice de cases*

Les entrées étant toujours codées dans le système de coordonnées pixel, il nous faut une fonction de conversion pour passer d'un système à l'autre.

```
① float posX = (((this.monde.getNiveau().getLargeur() / (float) w) * (float) x));
② float posY = (this.monde.getNiveau().getHauteur() -((this.monde.getNiveau().getHauteur() / (float) h) * (float) y));
```

Fonction de conversion pour chaque dimension

Comme le montre la ligne ①, le système se résout grâce à une simple règle de trois. La conversion d'unité à l'autre est proportionnelle ce qui simplifie les calculs. La ligne ② est intéressante, car un élément supplémentaire vient s'ajouter : on inverse par rapport à l'écran la position Y calculée par une règle de trois comme dans la ligne précédente. Cette inversion vient du fait que libGDX ne possède pas l'origine au même endroit que celui de l'écran et permet de compenser cette différence.

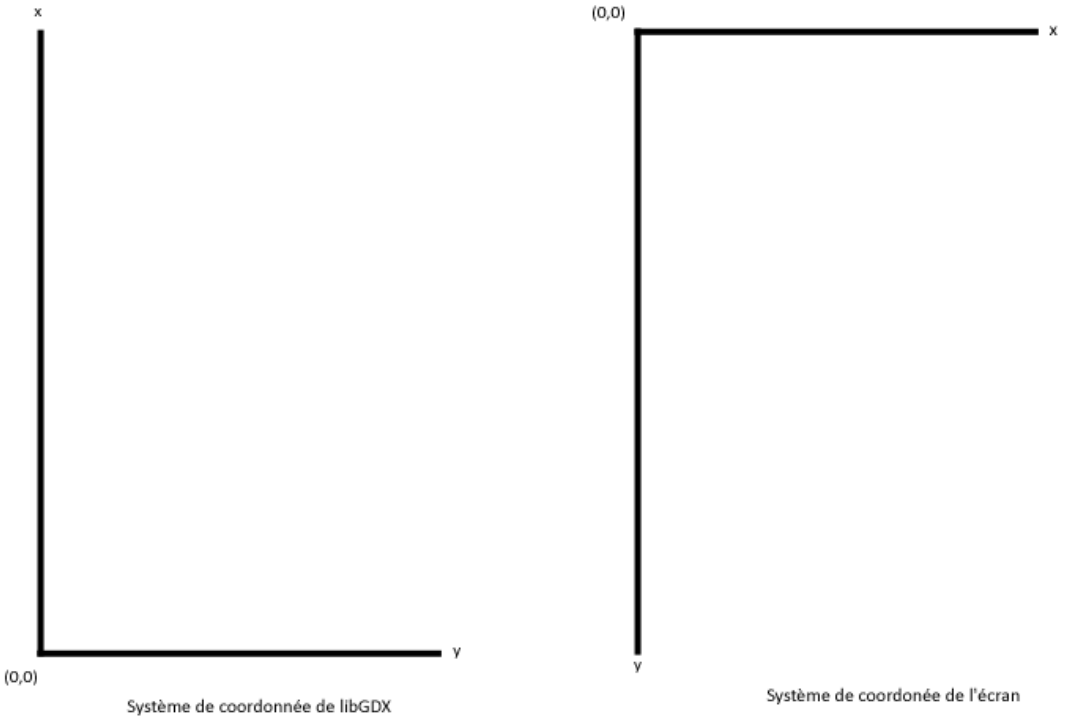


Figure 22 - Différences entre la méthode de positionnement de l'écran et de libGDX

❖ *Calcul de l'angle d'un vecteur*

Dans le gameplay*, nous avons défini la direction du tir du projectile en fonction de la direction imposée via la position du clic de souris par rapport à la position du personnage.

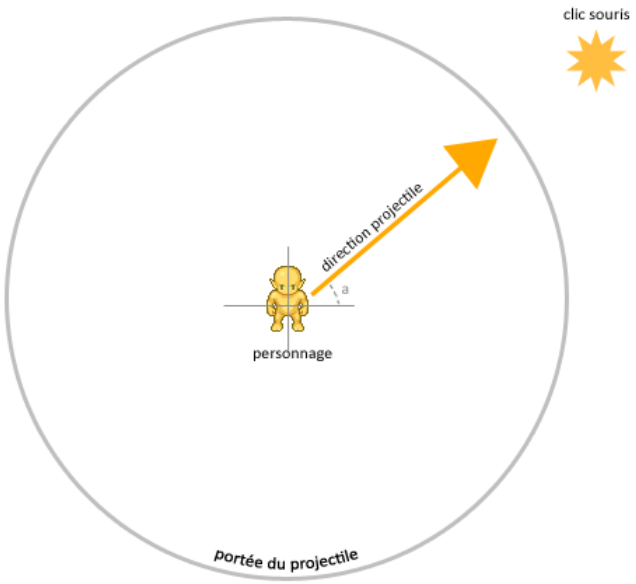


Figure 23 - Tir du personnage par rapport à la position de la souris

Aussi, libGDX possède une fonctionnalité qui permet le calcul de l'angle d'un vecteur. Le code ci-dessous montre l'utilisation de cette méthode qui retourne dans la variable angle la valeur de l'angle « a ».

```
// v est un vecteur
float angle = v.angle();
```

Cependant, nous pouvons réécrire cette méthode à des fins de compréhension et de recherche. Ainsi, pour donner une direction au projectile nous devons tout d'abord calculer l'angle « a » du clic par rapport à l'horizontale du personnage. En utilisant les propriétés trigonométriques du vecteur, on peut en déduire l'angle par la fonction inverse de tangente sur $]\frac{-\pi}{2} ; \frac{\pi}{2}[$ (①). Cependant, nous avons l'angle dans le repère polaire que l'on doit donc traduire pour l'adapter à un repère orthonormé afin de pouvoir l'exploiter par le vecteur. Pour les convertir, nous utilisons les propriétés trigonométriques des coordonnées polaires (②). Nous pouvons donc construire un vecteur de direction correspondant à la trajectoire voulue.

```
// v est le vecteur de la position du personnage à la position de la souris
// d est le vecteur représentant la direction du projectile
float angle = (float) Math.atan2(v.y, v.x); ①
d.x =(float)Math.cos(angle); ②
d.y =(float)Math.sin(angle);
```

Le calcul de la portée du projectile se fait par rapport à la distance qui sépare le projectile de son point de lancement. De plus, deux méthodes fournissent le même résultat, mais pas à la même rapidité.

```
// v est un vecteur
float dist = v.dst( x, y) ;
// v est un vecteur
float dist2 = v.dst2(x, y) ;
```

En effet, la méthode *dst()* utilise un calcul de racine carrée ce qui n'est pas le cas de dst2() qui est de ce fait plus rapide.

Nous fournissons ci-dessous une implémentation possible de *dst()*. La distance d'un calcul peut être calculée grâce au théorème du triangle rectangle de Pythagore.

$$x^2 + y^2 = distance^2$$

En calculant la racine carrée, on obtient la distance. Notons toutefois que la racine carrée est un calcul gourmand pour le processeur, mais négligeable par rapport à la puissance actuelle des ordinateurs et de la demande de ressources de la part de notre jeu. Voici une implémentation par rétro-ingénierie de la fonction *dst()*.

```
// vtemp est un vecteur vide
// posInitial le vecteur de départ du projectile
// position le vecteur de la position actuelle du projectile
vtemp.x = (Math.abs(position.x - posInitial.x)); // ici la soustraction est détaillée
vtemp.y = (Math.abs(position.y - posInitial.y));

// calcul de la distance par le théorème de Pythagore
float dist = (float) Math.sqrt(Math.pow((double)vtemp.x, 2.0)+Math.pow((double)vtemp.y, 2.0));

if (dist > DISTANCE_MAX){
```

```
		// détruire le projectile
}
```

2.3.3 La détection de collision

Le micro monde impose au joueur certaines règles qui l'empêche d'avoir une liberté totale de déplacement. Ainsi comme détaillé précédemment, les cases bloquantes ou à friction influent le comportement du joueur pendant son déplacement.

Il est donc primordial de se pencher sur la manière de contrôler ces interactions. La librairie libGDX possède une fonction *overlaps()* sur l'objet rectangle qui retourne vrai lorsque celui-ci entre collision avec un mur ou une case spéciale.

```
perso.getRapidite().scl(delta); ①// on travaille au ralenti

Rectangle persoRect = rectPool.obtain();②
persoRect.set(perso.getCadre());

this.chargerCollision();③
persoRect.x += perso.getRapidite().x; ④
persoRect.y += perso.getRapidite().y;

int i = 0;
boolean ok = true;
while (i< collision.size && ok){ ⑤
		if (collision.get(i) != null && persoRect.overlaps(collision.get(i))){ ⑥
				perso.getRapidite().x = 0;
				perso.getRapidite().y = 0;
				ok = false;
		}
i++;
}
perso.getRapidite().scl(1/delta); ⑦ // on restaure la vitesse
```

Tout d'abord, on adapte la vitesse du déplacement du personnage afin de travailler sur des valeurs convenables et compatibles avec le déplacement. Il est par exemple dangereux que ces valeurs soient trop grandes puisqu'à un certain seuil elles perdraient leur sens ①.

On crée ensuite un rectangle invisible qui simulera la présence du personnage après avoir avancé ②. On appelle ce rectangle un rectangle de travail.

Ensuite, nous chargeons dans un *array* la totalité des cases bloquantes dans le cas d'une détection de collision entre le joueur et des murs③. Cet *array* de collision se créé par rapport à l'état du joueur.

Puis nous mettons à jour les coordonnées du cadre du joueur qui sont modélisées par un rectangle ④. En effet, nous utilisons des rectangles pour optimiser la rapidité des fonctions de contrôle ou de tests sur les éléments.

Enfin, pour chaque élément de l'*array* ⑤, on vérifie que le rectangle de travail n'est pas entré en collision avec un élément bloquant ⑥. Dans le cas où le rectangle de travail entre en collision, nous remettons à zéro la vitesse de déplacement du personnage pour l'empêcher

de continuer. Finalement, nous pouvons restaurer la vitesse du joueur et actualiser la position avec l'avancement ou non du joueur.

2.3.4 Le système de matrice et le système array

Le système de matrice se définit par une matrice bidimensionnelle, dont chaque élément d'indice x, y représente une case sur la carte par rapport à sa position. L'*array* quant à lui permet de lister des cases. Les deux systèmes sont tout aussi opérationnels, mais chacun d'eux a leurs avantages. Aussi, nous utilisons les deux, car nous combinons leurs avantages pour améliorer la célérité du jeu.

En effet, les deux méthodes sont des approches différentes de travail sur les cases. Cependant si l'on traite la liste complète des cases bloquantes, par exemple, il est plus utile de parcourir une liste, plutôt que de tester par un parcours total d'une matrice bidimensionnelle si une case est, ou n'est pas une case bloquante. De même, il est plus simple pour vérifier les cases autour du joueur de parcourir les cases sur une matrice dont la position des cases est donnée par un indice plutôt que de parcourir une liste et de tester les positions de toutes les cases.

2.3.5 La modélisation des projectiles

Nous modélisons les projectiles comme des objets indépendants qui respectent les contraintes de déplacement et qui sont détruits à la collision avec une case bloquante ou au dépassement de la limite de la portée.

2.3.6 Gestion de l'orbe

La gestion de l'orbe se fait en multithreading. En effet, le passage d'un état à l'autre s'opère par une transformation qui ensuite est annulée par un « timer ».

```
public void orbPressed(){
        //Récupération des coordonnées du joueur
        int posX =(int)perso.getPosition().x;
        int posY = (int)perso.getPosition().y;

        //S'il est sur une ombre et que l'orbe est activé
        if (monde.getNiveau().get(posX, posY).getTypeCase()=="OMBRE" && orbEnabled){
                final Vector2 savePos = new Vector2(perso.getPosition());
                //Sauve la position du joueur
                perso.switchForm();
                orbEnabled = false; //désactive l'Orbe

                Timer.Task transform = new Timer.Task() ①
                {
                        @Override
                        public void run() {
                                System.out.println("[DEBUG] Transformation terminé.");
                                Vector2 back = new Vector2(savePos);
                                perso.setPosition(savePos);
                                perso.switchForm();
                        }
```

```
                };
                Timer.Task cooldown = new Timer.Task()
                {
                        @Override
                        public void run() {
                                System.out.println("[DEBUG] Orbe réactivé.");
                                orbEnabled = true; //activation de l'orbe
                        }
                };
                System.out.println("[DEBUG] Orbe utilisé.");
                Timer.schedule(transform, transformTime);
                Timer.schedule(cooldown, cooldownTime);
}
                /*Si la transformation est impossible, dire pourquoi
                 * Plus tard, on pourra implémenter un feedback*/
        else if(!orbEnabled)
                System.out.println("[DEBUG] L'orbe doit se recharger !");
        else System.out.println("[DEBUG] Tu n'es pas sur une ombre !");
}
```

L'objet « *transform* » est un processus qui va déclencher la méthode *run()* redéfinie de la classe mère, au moment où le temps est écoulé. Notons par ailleurs que les processus n'entrent pas en conflit.

2.3.7 L'intelligence artificielle

Dans le cadre de la gestion des animaux et monstres, nous sommes amenés à implémenter une intelligence artificielle. On peut séparer deux comportements distincts : les déplacements et les réactions à un évènement.

❖ *Déplacements*

Les déplacements se définissent par un parcours en boucle d'un chemin linéaire et prédéfini.

```
Vector2 v = a.getPath().get(a.getPathStep()); ①
Vector2 p = a.getPosition(); ②

corrigeDirection(a); ③
float aprox = 0.1f;
if ( Math.abs(p.x - v.x )<aprox && Math.abs(p.y - v.y )<aprox){ ④
        this.nextStep(a); ⑤
}
```

① On récupère dans le vecteur v les coordonnés de la case visée par l'élément animé. ② On affecte la position de l'élément animé au vecteur. ③ Ici la direction est corrigée si on n'est pas dans la bonne direction et si on entre en collision avec la case visée ④ le code sélectionne la case suivante ⑤.

❖ *Réaction à un évènement*

Les monstres ont en plus des déplacements la possibilité de réagir à un ou plusieurs évènements. Par exemple lorsqu'un joueur se téléporte d'une case ombre à l'autre le monstre peut l'intercepter et le tuer.

2.3.8 Intégration graphique

Le lien entre graphisme et programmation se fait au moyen de « sprites ». Les sprites se définissent par une texture ou en ensemble de textures qui seront affichées à une position donnée. Ils servent principalement à la création du rendu sur l'écran tout en simplifiant la programmation puisque le développeur travaille sur des petits morceaux d'écran plutôt que sur l'écran entier.

```
TextureRegion imgLightForm = new TextureRegion(new Texture(Gdx.files.internal("images/light.png")));
spriteBatch.draw(imgLightForm, X, Y, largeur, hauteur);
```

Les « atlas » sont des ensembles de textures qui sont réunis sur une image afin d'accélérer le traitement des images par la carte graphique, de simplifier le code et l'arborescence des fichiers.

2.4 Architecture du logiciel

2.4.1 Programmation MVC

Le modèle MVC se décompose en trois différents éléments :

- Modèle : qui ne contient que la définition des éléments
- Vue : qui ne contient que la manière d'afficher les modèles
- Contrôleur : qui détermine le comportement des modèles

L'arborescence des fichiers est visible en annexe 2.

2.4.2 Hiérarchie des classes

Notre jeu est conçu selon une certaine arborescence. Lors de l'affichage, par exemple, la classe GameScreen.java appelle la méthode update d'AfficheMonde.java, qui appelle les méthodes updates de tous les modèles.

Comme indiqué sur la figure en annexe 5, la classe GameScreen.java est au centre de l'affichage puisqu'elle exécute indirectement toutes les méthodes de tous les objets affichables du jeu.

2.4.3 Développement multiplateforme

LibGDX propose par défaut différentes plateformes telles que Windows, Mac, Linux, Android, iOS et HTLM 5 pour une interface web.

Notre projet se décompose en trois sous-projets que sont le projet desktop, le projet Android et le moteur du jeu. Aussi, nous avons conçu deux interfaces différentes pour la version bureau et pour la version mobile.

2.5 Résultat

2.5.1 Présentation du jeu

Le jeu se définit par un personnage principal qui, pour avancer dans l'aventure, doit franchir des niveaux. Face au joueur s'élèvent des retardataires tels que des murs, des cases spéciales, des monstres… Mais également des aides pour le joueur, comme les animaux qui ont des caractéristiques spéciales.

2.5.2 Fonctionnalités du cahier des charges

Pour concevoir ce gameplay* original, nous devions réaliser un système de jeu sur deux cœurs différents. Pour cela, nous avons implémenté deux formes : la Shadow Form et la Light Form.

- Shadow Form
  - Déplacement dans les huit directions à condition d'être dans l'ombre.
  - Téléportation d'une zone d'ombre à une autre en utilisant un grappin qui doit être lancé.
  - Possibilité de contrôler différents animaux.
  - Changement de forme.
  - Impossibilité de traverser les murs.
  - Invisibilité dans l'ombre pour les animés.
- Light Form
  - Déplacement dans les huit directions.
  - Possibilité d'attaquer un monstre.
  - Impossibilité de traverser les murs

Parallèlement, les animés se décomposent en deux sous-catégories : les animaux et les monstres. Les animaux sont passifs et limités, contrairement aux monstres qui interagissent avec le personnage en l'attaquant.

- Animaux
  - Possibilité de déplacement linéaire prédéfini et parcouru en boucle.
  - Possibilité d'être contrôlé par le joueur.
  - Capacité spéciale.
  - Impossibilité de traverser les murs.
- Monstres
  - Déplacement linéaire prédéfini et parcouru en boucle.
  - Capacité de détecter le joueur s'il n'est pas dans une ombre.

- Attaquer le joueur dès que possible.
- Impossibilité de traverser le noir.
- Possibilité de mourir sous les projectiles du joueur.
- Impossibilité de traverser les murs.

3 Manuel d'utilisation et d'installation

3.1 Manuel d'installation

3.1.1 Environnement Java (JRE)

Une machine virtuelle Java est nécessaire pour lancer le projet. L'installateur est disponible sur le site officiel www.java.com/fr/. Lancez et complétez l'installation pour disposer de la machine virtuelle Java qui permettra d'exécuter le jeu.

3.1.2 Récupération du jeu

Si vous souhaitez uniquement récupérer le code source, veuillez télécharger l'archive à cette adresse : http://projets-lightdark.fr/wiki/index.php?n=Telecharger.Source ou via ce lien court http://bit.ly/1aa7zww.

La version java du projet est disponible en téléchargement à l'adresse http://projets-lightdark.fr/wiki/index.php?n=Telecharger.Executable ou utilisez ce lien court : http://bit.ly/1mb8srd. Téléchargez le fichier .jar et double-cliquez dessus. Le jeu va maintenant se lancer.

Si vous utilisez une console sous Linux, placez-vous dans le répertoire du projet *cd chemin/du/projet* puis exécutez le fichier avec la commande *java jar monfichier.jar*

3.2 Manuel d'utilisation

3.2.1 Se déplacer

Le personnage peut se déplacer à l'aide des touches Z, Q, S, et D du clavier. Il est ainsi possible de se déplacer dans les quatre directions cardinales et en diagonale.

3.2.2 Changer de forme

En cliquant ou en appuyant sur l'orbe, le joueur change temporairement de forme. Si l'orbe est grisé alors il n'est pas possible de changer de forme pour le moment.

L'orbe est utilisable uniquement si le joueur est placé sur une ombre. De plus, certaines zones peuvent empêcher l'orbe de fonctionner.

3.2.3 Attaquer

Quand il est sous sa forme de lumière, le joueur peut attaquer en cliquant ou en appuyant dans la direction voulue. Le personnage porte alors un coup d'épée dans la direction pointée. Il peut ainsi combattre les monstres qui se dressent contre lui.

3.2.4 Se glisser dans une ombre

Sous forme d'ombre, le joueur peut se déplacer d'ombre en ombre. Pour ce faire, il dispose d'un grappin qui lui permet de se déplacer sur l'ombre pointée si elle est bien à portée de son grappin. Sinon, le grappin revient à son lanceur.

3.2.5 Contrôler un animal

Il est également possible de prendre le contrôle d'animaux en se glissant dans leur ombre. Ainsi, on peut contrôler les mouvements de l'animal : se déplacer et utiliser son action spéciale. Cette action spéciale est propre à chaque animal et se déclenche en gardant le clic enfoncé lorsque l'on contrôle l'animal.

4 Rapport d'activité

4.1 Méthode de gestion de projet

Pour le développement de notre projet, nous avons adopté, à l'initiative de notre tuteur, M. Abdelkader Gouaich, la méthode agile Scrum.

4.1.1 Méthode Scrum : définition

La méthode Scrum est une méthode agile pour gérer les projets. Dans cette méthode, le travail à réaliser est découpé en différentes parties, nommées sprints, qui couvrent chacune une courte période de temps (dans notre cas, d'une à trois semaines).

Au début de chaque sprint, nous nous retrouvions avec notre tuteur pour faire le point sur le sprint précédent et fixer les objectifs pour le sprint courant. Nous nous organisions alors pour nous répartir librement les différentes tâches entre nous, selon le principe d'auto-organisation de cette méthode.

Enfin, nous nous réunissions fréquemment durant les sprints au cours de réunions de travail ou de réflexion, que nous avons nommées « séances de brainstorming ».

4.1.2 Bilan d'utilisation

Nous avons rapidement réussi à mettre en place cette technique. L'utilisation de la méthode agile Scrum nous a permis, rétrospectivement, de structurer et d'organiser l'avancement de notre projet, combiné à l'utilisation de Trello. Nous avons cependant aménagé et modifié quelque peu cette méthode pour l'adapter avec nos autres contraintes scolaires.

Nous avons aussi suivi cette méthode pour les réunions avec M. Gouaich, mais nous l'avons, concernant la tenue de nos réunions (ou brainstorming), quelque peu modifié pour l'adapter à la tenue de réunions plus longues.

4.2 Planification

4.2.1 Planning prévu

Comme demandé par les consignes, nous avons réalisé un planning initial, sous forme de diagramme de Gantt, couvrant l'intégralité de la durée du projet, en évaluant à priori les tâches à réaliser et leur durée. Celui-ci se trouve en annexe 1.

Ce planning global a au final été peu suivi, mais nous a cependant poussés, et ce dès le début du projet, à établir une liste de tâches à réaliser, que nous avons souvent réutilisée par la suite pour savoir où nous en étions dans le projet ou bien savoir quelles tâches globales nous devions alors réaliser.

4.2.2 Planning réel

❖ *Des plannings pour chaque sprint*

Nous avons, en plus du planning global, réalisé pour chaque sprint des plannings des tâches à l'aide de diagrammes de Gantt. Ces plannings, plus en adéquation avec la réalité de l'avancement de notre projet, ont été plus réalistes et mieux suivis.

❖ *Les contraintes*

Ces plannings ont cependant été soumis à quelques contraintes, du fait de notre planning chargé, ou bien de nos indisponibilités individuelles.

❖ *Conclusion sur les plannings*

La réalisation de ces plannings nous aura permis, dans le prolongement de la méthode Scrum, de lister nos tâches à effectuer et tenter de s'organiser pour que ces tâches soient réalisées durant le temps imparti, et ce à chaque sprint. C'est donc un outil puissant qui nous a permis de mieux structurer la progression de notre projet.

4.3 Méthode et outils de travail

4.3.1 Section modélisation

❖ *Modélisation UML*

Pour la modélisation nous avons choisi d'utiliser surtout le logiciel Modelio, mais aussi parfois le logiciel StarUML, qui sont des logiciels gratuits de modélisation UML. Nous avons toutefois porté notre préférence sur Modelio simplement pour l'esthétique de ses diagrammes.

❖ *Autres diagrammes*

Pour réaliser quelques diagrammes explicatifs sous forme de cartes mentales, à destination du wiki et du rapport de projet, nous avons également utilisé occasionnellement le logiciel Xmind.

Enfin nous avons réalisé quelques schémas à la main, que nous avons ensuite scannés.

4.3.2 Section codage

Pour le codage, nous avons utilisé le langage de programmation Java, la plateforme de développement Eclipse, et libGDX, avec SVN sur un serveur Google.

❖ *Java*

Nous avons dû utiliser, dans le cadre de notre projet, libGDX qui est une librairie écrite en Java. Par ailleurs, le langage de programmation Java ayant la particularité d'être portable sur tous les supports, il était parfaitement adapté à notre projet et c'est tout naturellement que nous l'avons choisi.

❖ *LibGDX*

LibGDX est une librairie Java, axée sur la création et le développement de jeux vidéo, conseillé pour ce projet. Bien que nous ne la connaissions pas, elle s'est révélée être d'une grande aide pour le développement du jeu vidéo, en nous offrant de multiples méthodes déjà réalisées et en nous fournissant un squelette de code type. Nous n'avons toutefois pas utilisé la surcouche GAME, développée par le LIRMM, car nous souhaitions avoir une plus grande liberté et maitrise sur le code.

❖ *SVN*

SVN est un outil de gestion de version, qui permet de gérer le développement d'un projet à plusieurs membres. Nous avons choisi d'utiliser SVN plutôt que GIT, son concurrent, car c'est un outil sur lequel nous maitrisions mieux grâce à notre formation. Malgré quelques erreurs de version sur certains fichiers, cet outil s'est révélé particulièrement utile pour travailler en commun sur une ou parfois plusieurs copies du programme.

Nous avons choisi de stocker notre projet sur un serveur Google, en raison sa simplicité d'utilisation et sa gratuité.

❖ *Utilisation coordonnée*

Une séance usuelle de travail sur le code du jeu suivait les étapes suivantes :

- Définition des objectifs de la séance sur Trello
- Récupération de la dernière version du dépôt SVN
- Développement de la fonctionnalité fixée plus tôt
- Mise à jour de la copie de travail puis enregistrement des modifications sur le dépôt

4.3.3 Section organisation.

❖ *Wiki*

Nous avons réalisé un wiki, pour permettre aux observateurs de suivre l'évolution de notre projet, comme demandé dans les consignes du projet.

Nous avons préféré l'utilisation d'un wiki à celle d'un blog, pour plus de simplicité : nous sommes en effets plus habituées à utiliser un wiki qu'un blog. De plus, l'utilisation d'un

wiki nous a semblé plus cohérente avec notre projet qui possède une documentation dense et qui nécessite rapidement une organisation structurée.

Ce wiki a régulièrement été mis à jour, d'autant plus qu'il s'est rapidement révélé utile pour savoir entre nous l'avancement global du projet, de suivre l'évolution du projet et de stocker les comptes rendus des réunions et de connaitre l'avancement général du projet.

❖ *Trello*

Trello est un outil web gratuit permettant d'organiser et de gérer des projets. Les tâches, qui peuvent être ajoutés par chacun des membres, peuvent être ajoutés dans trois colonnes : à faire, en cours, fait, et sont déplaçable de l'une à l'autre. Elles peuvent par ailleurs se voir attribuer des membres.

Trello nous est vite apparu comme un outil permettant d'étendre l'utilisation de la méthode Scrum, notamment concernant la distribution et le choix des tâches à réaliser, et s'est rapidement imposé comme un outil indispensable de cette méthode.

Bien utilisé, Trello nous a permis de répondre aux questions : « Que faut-il faire ? », « Qui fait quoi ? », « Ou en est-on? », et nous a permis de nous organiser facilement au niveau de chaque tâche.

Cet outil, que nous avons tenté d'associer à chacune de nos séances de travail, en particulier lors du codage où les petites tâches se sont démultipliées, a pu être d'une grande aide.

❖ *Communication*

Au niveau du travail, nous avons travaillé en groupe et individuellement, les réunions avec les professionnels se déroulant avec une partie ou la totalité du groupe.

Concernant le travail de groupe, nous avons privilégié autant que possible la rencontre de visu, de préférence dans les locaux de l'IUT de Montpellier. Lors des séances de télétravail, nous avons utilisé Teamspeak 3 comme logiciel de communication.

❖ *Rapport*

Enfin, pour l'écriture du présent rapport, nous avons choisi d'utiliser Microsoft Word 2013, pour nous simplifier la mise en page et l'écriture du rapport en utilisant une plateforme moderne et identique pour chacun des membres du groupe.

Conclusion

Réaliser un jeu vidéo est une expérience assez différente de celle que l'on peut rencontrer dans d'autres projets, mais surtout de ce que l'on pouvait en penser avant de commencer Light | Dark. L'une des premières difficultés vient du game design document*, qu'il faut savoir réaliser sans penser au scénario ou au level design* afin de ne pas dénaturer le gameplay*, qui est au final la composante la plus importante. Le gameplay à deux cœurs de Light | Dark permet une grande richesse en termes de combinaisons et énigmes possibles, tout en conservant cet aspect dynamique que nous recherchions pour ce prototype. Par ailleurs, si le temps imparti ne nous a pas permis d'en réaliser la totalité et de l'adapter au projet, les dessins conceptuels et l'environnement sonore actuel nous ont permis de définir l'orientation artistique du jeu et donc d'avoir un avant-goût de ce que pourrait devenir le jeu.

Nous souhaitons d'ailleurs continuer ce projet en collaboration avec notre tuteur de projet afin d'obtenir un jeu complet qui pourra être distribué sur différentes plateformes de téléchargement tels que Google Play. À cette fin, nous développerons une version totalement compatible avec Android et nous continuerons aussi de travailler avec les professionnels qui nous ont aidés à concevoir l'esthétique du jeu afin d'aboutir à un jeu soigné ayant sa propre identité.

Light | Dark nous a notamment permis d'approfondir nos connaissances en matière de développement Java, de programmation événementielle et graphique ; mais aussi de connaitre certaines techniques propres au jeu vidéo grâce à la librairie libGDX. Il nous a aussi permis de nous familiariser avec la méthode agile Scrum et d'acquérir une certaine rigueur dans le développement, tout en apprenant à communiquer avec d'autres corps de métiers afin de collaborer sur un même projet.

Médiagraphie

Campbell, J. (1949). *The Hero with a Thousand Faces.* Pantheon Books.

Nintendo. (1986, Février). The Legend Of Zelda. Japon.

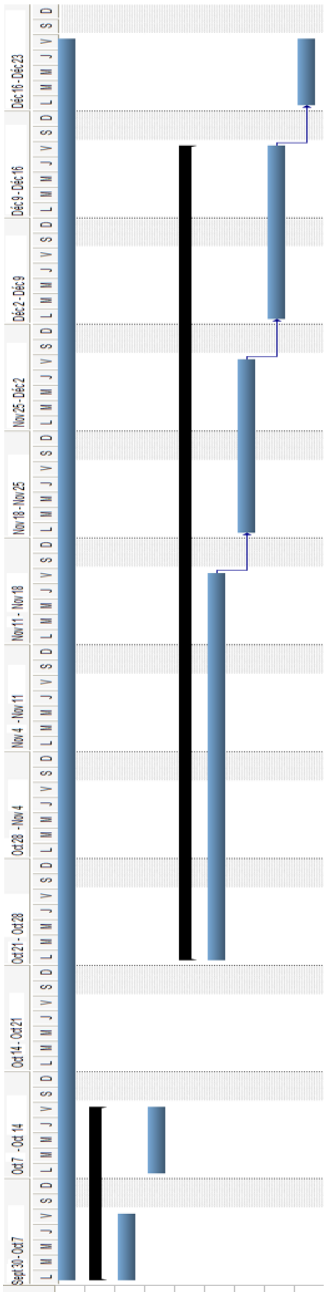
Swirsky, J., & Pajot, L. (Réalisateurs). (2012). *Indie Game : The Movie* [Film documentaire].

Tamas, J. (2012, Février 28). *Getting Started in Android Game Development with libgdx*. Récupéré sur against the grain : http://obviam.net

Ubisoft. (2003). Prince of Persia : The Sands of Time. Montréal, Canada.

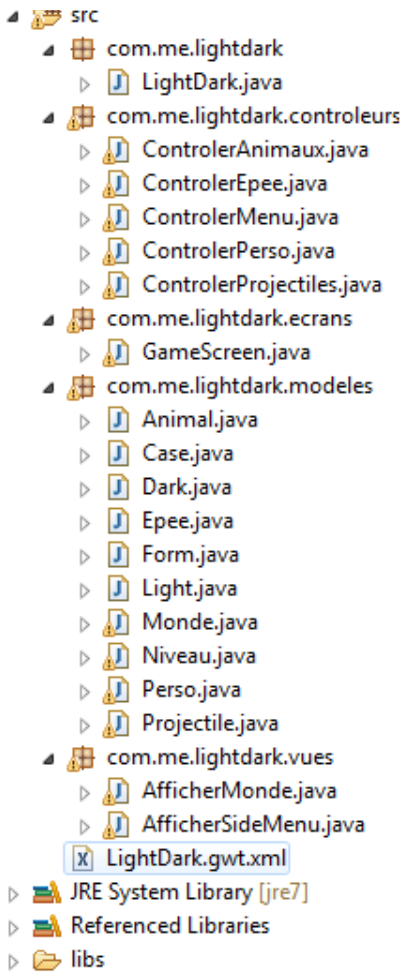
Annexes

ANNEXE 1 - Planning global initial



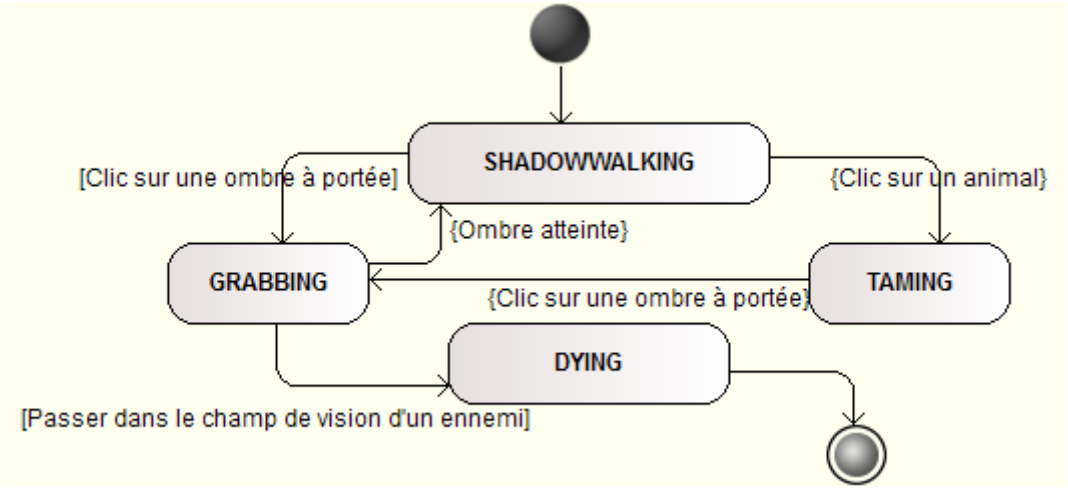
Nom	Durée	Début	Fin
Projet jeu JAVA 2D	3m?	14/10/2013	03/01/2014
Préparation et analyse	**0.5m?**	**14/10/2013**	**25/10/2013**
Installation de l'environnement de travail	1s?	14/10/2013	18/10/2013
Creation d'un document de Game Design	1s	21/10/2013	25/10/2013
Developpement	**2m?**	**04/11/2013**	**27/12/2013**
Codage	1m?	04/11/2013	29/11/2013
Graphisme	2s?	02/12/2013	13/12/2013
Son	2s	16/12/2013	27/12/2013
Tests finaux	1s	30/12/2013	03/01/2014

ANNEXE 2 - Arborescence des fichiers du projet

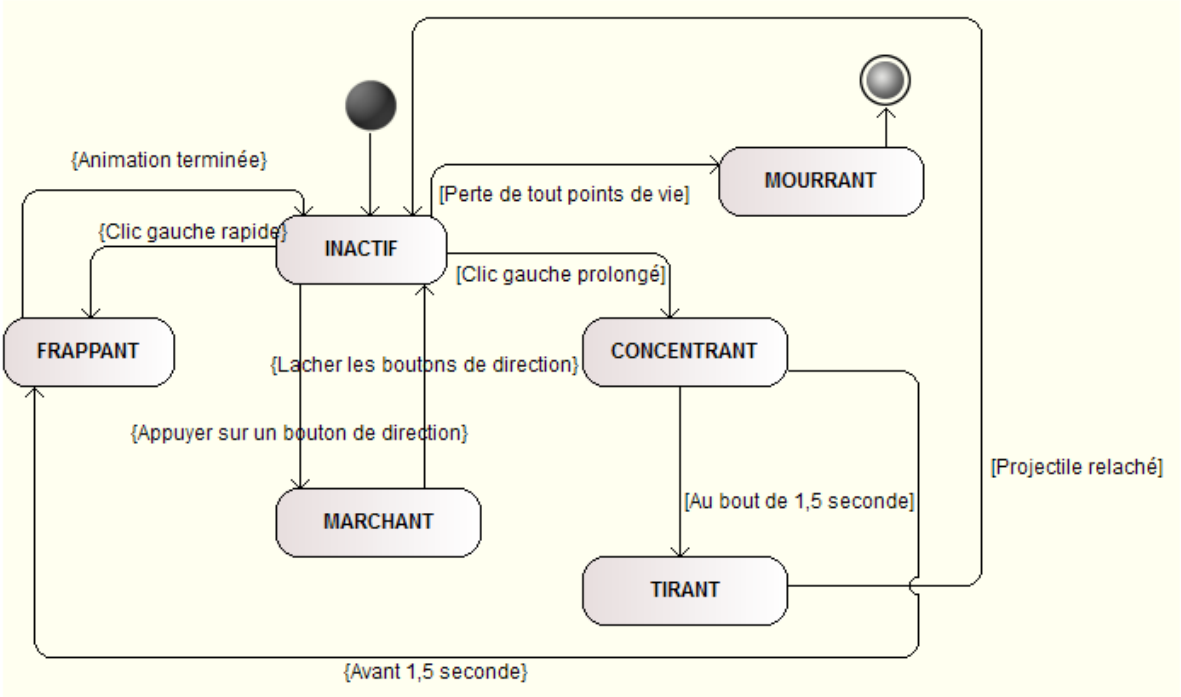


ANNEXE 3 - Diagramme d'états/transitions des formes

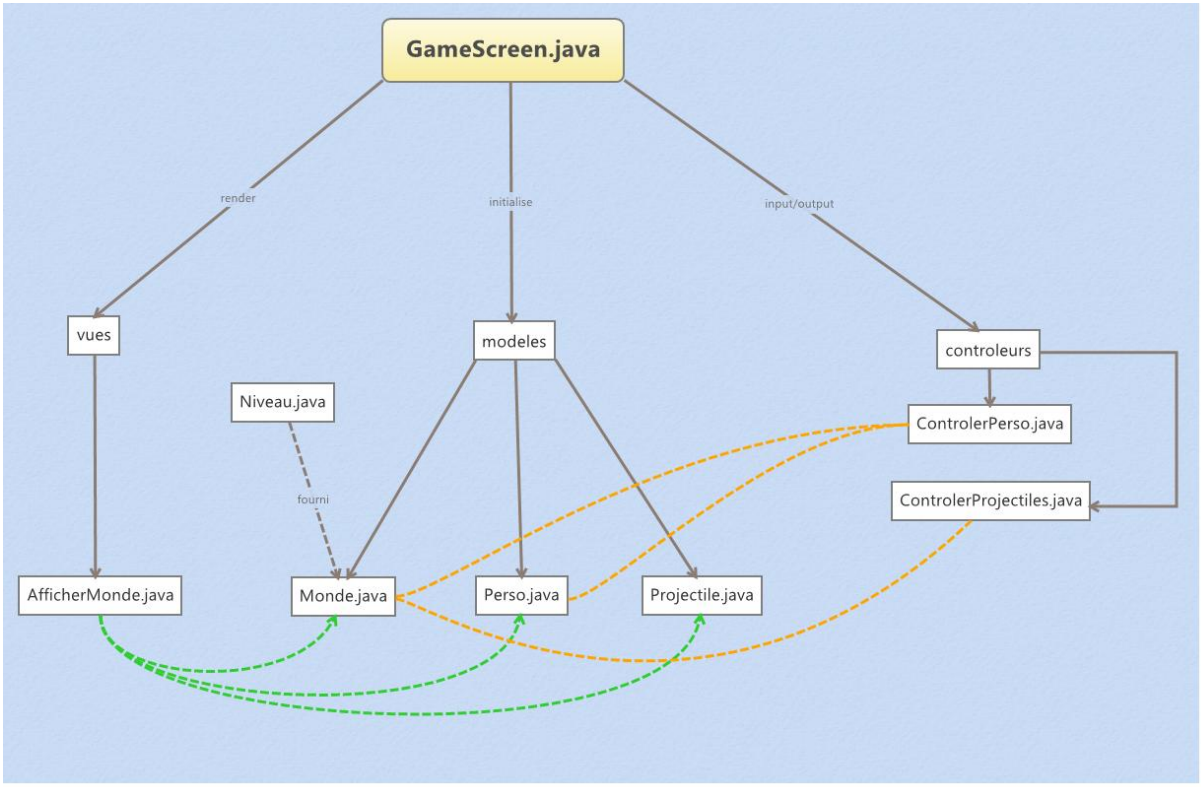
ANNEXE 4 - Diagramme d'états/transitions de la Shadow Form



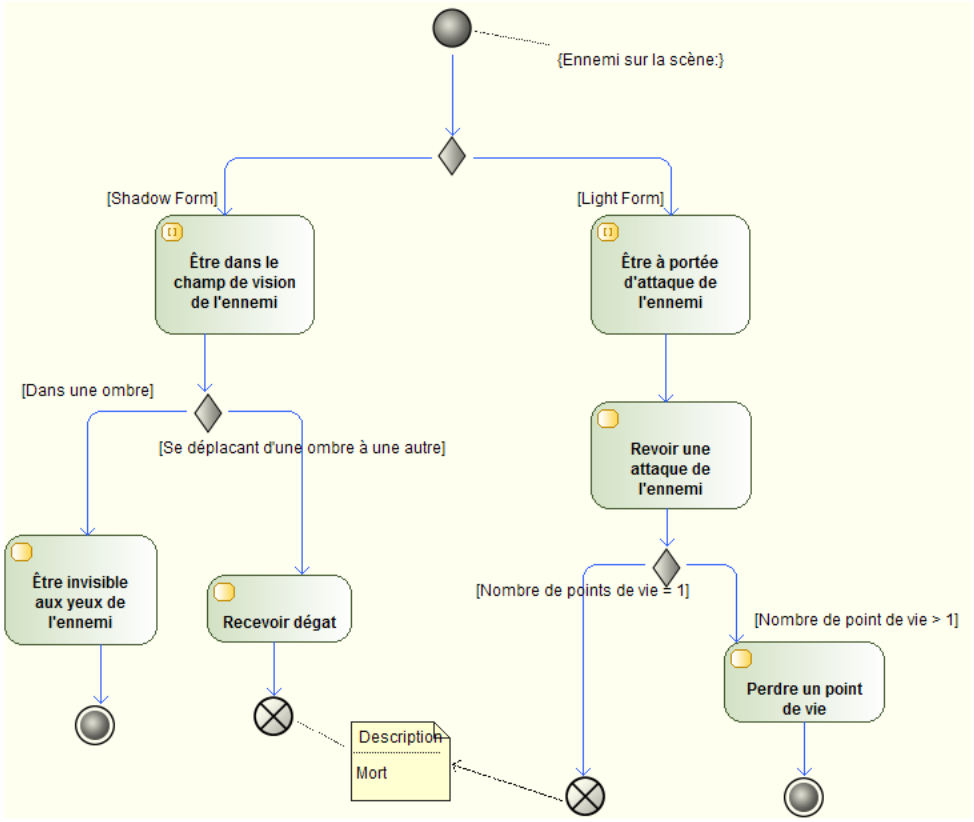
ANNEXE 5 - Diagramme d'états/transition de la Light Form



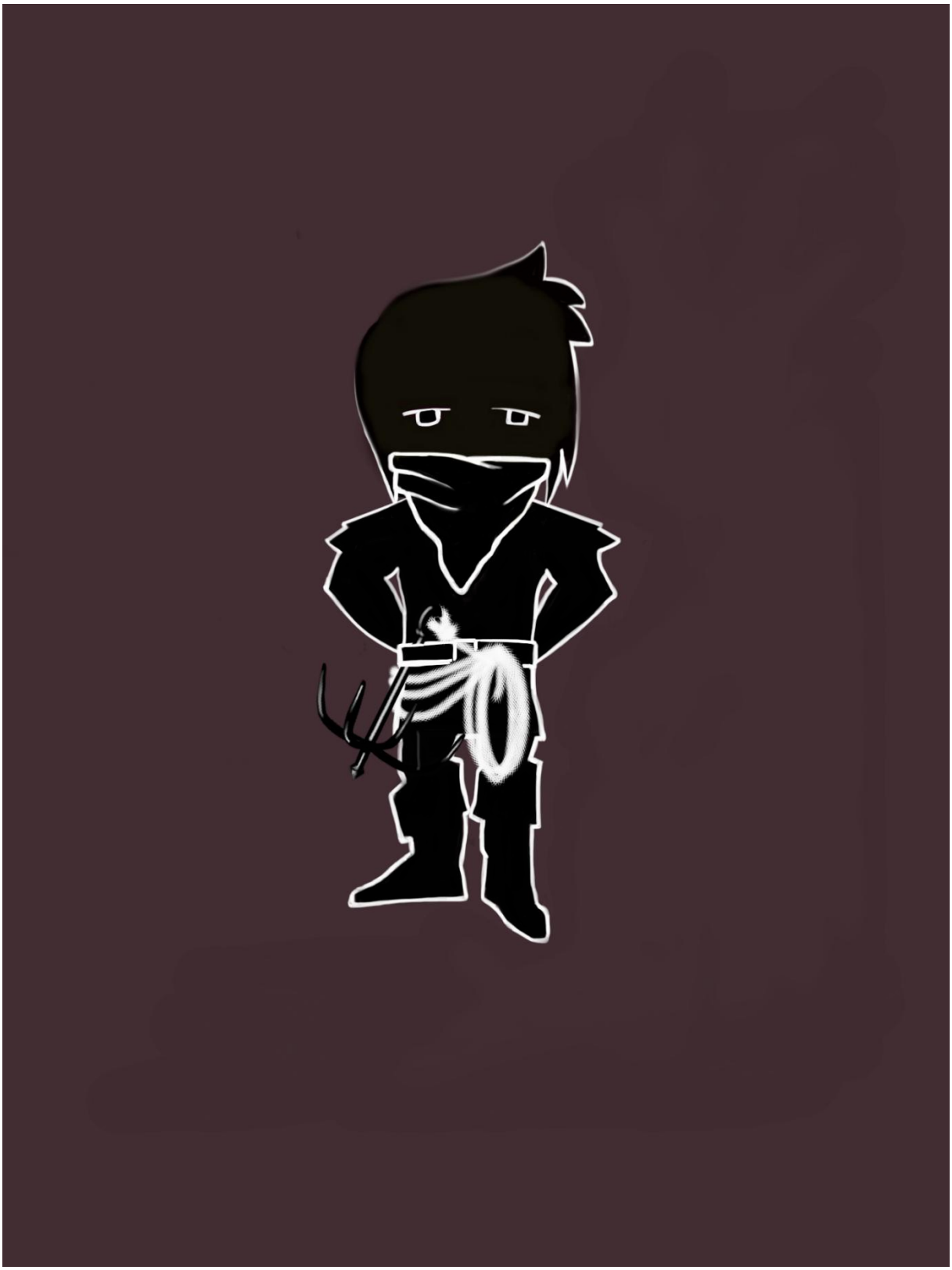
ANNEXE 6 - Schéma de fonctionnement des classes



ANNEXE 7 - Diagramme d'activité de l'interaction monstres/personnage



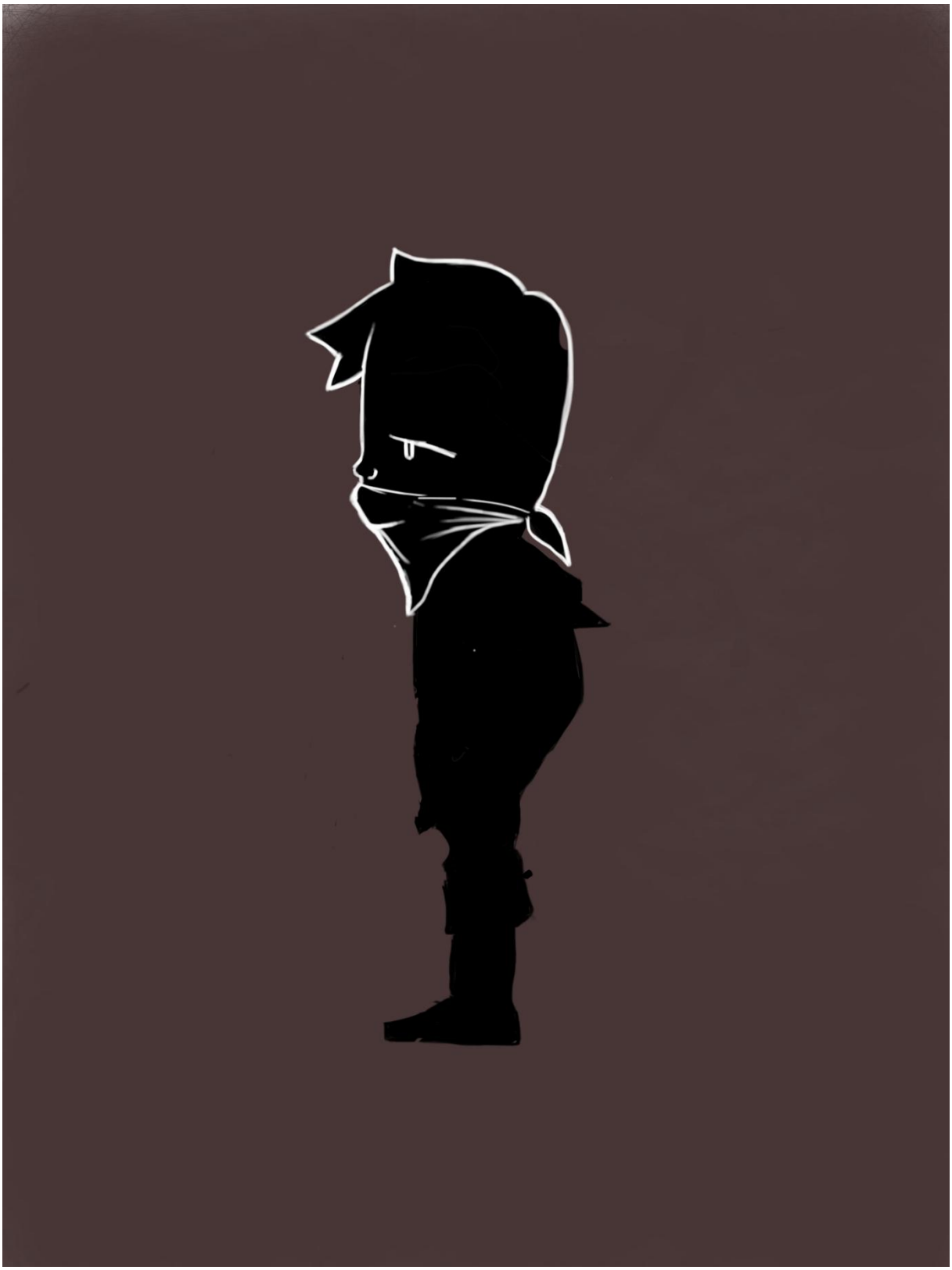
ANNEXE 9 - Artwork Shadow Form



ANNEXE 10 - Ébauche de profil droit de la Light Form

ANNEXE 11 - Ébauche de profil gauche de la Light Form

ANNEXE 12 - Ébauche de profil gauche de la Shadow Form



Résumé

Le projet Light | Dark vise à créer un jeu vidéo en deux dimensions en passant par toutes les étapes de création d'un jeu, du document de game design jusqu'à la réalisation d'un modèle pleinement fonctionnel pouvant être présenté puis joué par des joueurs réguliers.

Il illustre les différentes contraintes liées à la réalisation d'un jeu vidéo orienté puzzle/aventure : analyse, ergonomie, mécanismes, environnement visuel et sonore, mais surtout la relation particulière entre le scénario et le gameplay.

Ce jeu a été développé en utilisant le langage de programmation Java et la librairie libGDX.

Mots clés

Jeu vidéo, 2D, game design document, gameplay, scénario, boucle OCR, Java, libGDX.

Summary

The Light | Dark project aims to create a bi-dimensional video game by going through all the steps of the video game development to the realization of a fully functional model which can be presented then played by regular players.

It also illustrates the constraints related to the realization of a puzzle/adventure oriented video game: analysis, ergonomics, mechanisms, visual and sound environment, and most of all the particular relationship between scenario and gameplay.

This game has been developed using the programming language Java and the library libGDX.

Keywords

Video game, 2D, game design document, gameplay, scenario, OCR loop, Java, libGDX.