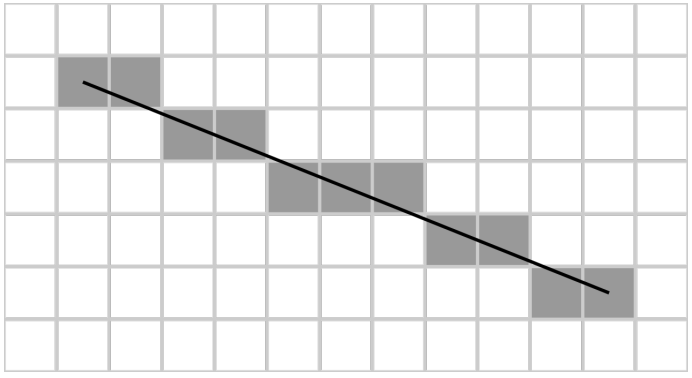
Algorithmes de ligne de vue

Begue Max, Broc Nina, Michaud Jules, Moutet Maxime, Seegmuller Raphaël

24 avril 2019



Contents

Remerciements :
Nous tenons particulièrement à remercier Mr. François BRUCKER, professeur à l'École Centrale Marseille, ainsi que notre encadrant de projet, pour nous avoir proposé ce sujet ainsi que pour nous avoir conseillé et guidé dans notre travail et être à notre écoute quand nous en avions besoin tout au long du semestre; et sans qui nous n'aurions pas pu arriver au bout du sujet.

1 Introduction

Très utilisée dans les jeux vidéos, la ligne de vue permet de déterminer ce qui est visible par un personnage et ce qui ne l'est pas dans un univers à deux ou trois dimensions. Dans un jeu vidéo, une ligne de vue est une droite reliant des points de l'univers vus par le personnage. La réunion de toutes les lignes de vues d'un personnage permet d'avoir le champ de vision du personnage.

Nous nous sommes donc demandé comment pouvait-on déterminer s'il existe une droite reliant 2 points sans obstacle, et comment représenter une telle droite dans un environnement en 2D composé uniquement de pixels. Pour calculer cette ligne de vue, il existe plusieurs algorithmes. Dans ce projet, nous vous présenterons ainsi les recherches que nous avons menées et la démarche scientifique que nous avons élaborée afin de pouvoir développer, comprendre et appliquer certains de ces algorithmes. Dans un premier temps, nous avons cherché à tracer une droite dans un univers 2D composé de pixels, au moyen de différents algorithmes. Une fois cela réalisé, nous avons vu comment l'on pouvait représenter cette droite de façon plus esthétique et dynamique. Enfin, nous nous sommes intéressés à comment représenter cette droite plus rapidement et comment étendre ce modèle à un univers 3D. Pour toutes ces droites, nous avons utilisé le langage de programmation Python 3.5.

2 Le codage de droites

Dans cette partie, nous nous intéresserons à comment coder une droite en python dans un univers 2D composé de pixels. Pour représenter cet univers, nous avons travaillé avec des matrices (listes de listes en python). Pour pouvoir travailler avec des coordonnées, nous avons dû définir des axes sur ces matrices: Un axe x partant du coin en haut à gauche de la matrice et allant vers la droite; et un axe y partant du coin en haut à gauche et allant vers le bas.

2.1 L'algorithme de Bresenham

Nous avons tout d'abord trouvé que l'algorithme original permettant de relier deux points par une droite est celui de Bresenham. Développé en 1962, il permet de déterminer les points qui doivent être tracés afin de former une approximation de droite. On peut voir un exemple d'une droite de Bresenham sur la figure 2, page 6. Nous admettrons que cet algorithme fonctionne, étant donné qu'il est utilisé dans de nombreux jeux-vidéo en 2D. D'après la définition d'une droite, celle-ci doit être la même du point A au point B et inversement du point B au point A : en algorithme, cela implique la présence d'une symétrie par rapport au milieu de la droite. De plus, on remarque que la droite passe par certains pixels qui ne sont pas coloriés en vert. En effet, nous verrons pourquoi expliquant le fonctionnement de l'algorithme ci-dessous.

Fonctionnement de l'algorithme Le principe est ici de calculer quels pixels sont les plus proches de la droite. Pour expliquer le fonctionnement, nous nous placerons dans le huitième octant (cf fig1, p.5).

Soient les points A(x_A, y_A) au centre de la matrice et B(x_B, y_B) dans le huitième octant formant la droite (AB). À chaque itération de l'algorithme, on allume un pixel sur la matrice. Le premier pixel a être allumé est donc celui avec les coordonnées de A.

$$M(x,y) \in (AB) \Leftrightarrow \vec{AM} = k \times \vec{AB}, k \in \mathbb{Z}$$

Ce qui revient à dire que:

$$x_{AB} \times y_{AM} = x_{AM} \times y_{AB} \Rightarrow (y - y_A) = \frac{(y_B - y_A) \times (x - x_A)}{(x_B - x_A)}$$

Avec $x_B - x_A \neq 0$
On pose

$$m = \frac{y_B - y_A}{x_B - x_A}$$

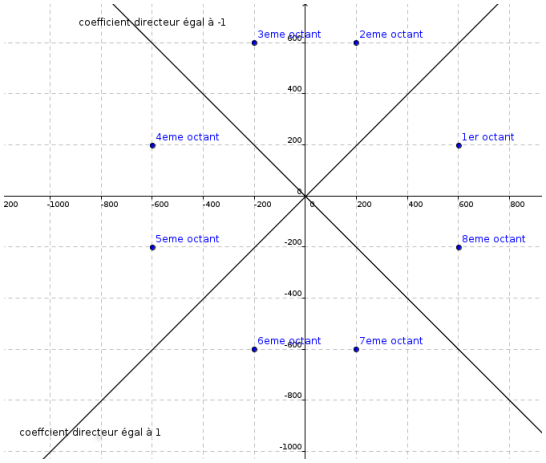


Figure 1: Decoupage du plan en 8 octants

m est le coefficient directeur de la droite (AB). Pour déterminer quels pixels doivent être allumés, on se déplace selon l'axe x en ajoutant la valeur de m à un compteur e (initialement e = 0) comme le montre l'équation suivante lorsqu'on ajoute 1 à x.

$$y - y_A = m \times (x + 1 - x_A) = m \times (x - x_A) + m \tag{1}$$

A chaque fois que e dépasse 0,5, cela veut dire que le pixel le plus proche de la droite est le pixel situé au dessous. On allume donc le pixel en ajoutant 1 en coordonnée y et on retranche 1 au compteur afin de pouvoir continuer, jusqu'à B.

Une fois que l'on a compris le fonctionnement pour tracer la ligne dans un des octants, il suffit de trouver des axes de symétrie afin de tracer les lignes dans les autres octants. On fait donc la symétrie selon l'axe y pour avoir la droite dans l'octant 5. La symétrie par rapport à l'axe x pour avoir la droite dans l'octant 1. La symétrie par rapport à la droite y = x pour avoir la droite dans l'octant 7 (cf fig.1 p.5 pour les octants).

2.2 La naissance d'un premier algorithme

L'algorithme que nous avons développé a découlé de plusieurs algorithmes plus ou moins performant, que nous allons brièvement vous présenter; disponibles dans le fichier fourni avec le rapport.
Dans toute cette partie, on considérera qu'une matrice (en python une liste de liste) est une bonne représentation d'un monde 2D, où chaque coefficient sera considéré comme un pixel. De plus, le code associé à cette partie est le fichier "Algo1.py" dans le dossier fourni.

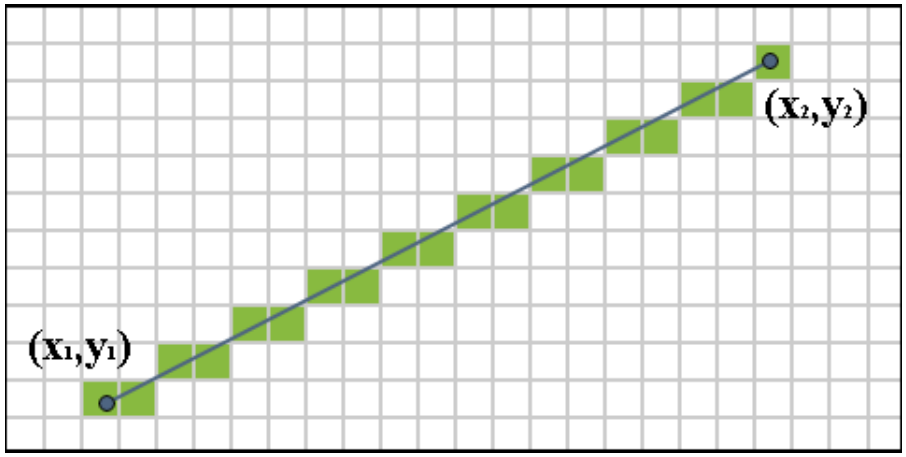


Figure 2: Droite reliant deux points avec l'algorithme de Bresenham - extrait de *https* : $//fr.wikipedia.org/wiki/Fichier$: $Bresenham\_line.png$

Version 1 Dans la première version, nous avons simplement cherché à représenter une droite en partant d'un coin d'une matrice vierge remplie de 0, grâce à la méthode de Bresenham, avec la fonction "line _ replace", qui met des 1 sur les pixels que l'on voit. Nous avons traité pour ce premier code, uniquement les octants 1 et 2 (en se plaçant le points de départ en bas à gauche de la matrice), ou les octants 7 et 8 (en se plaçant en haut à gauche) (cf fig.1 p.5).
On a alors 4 cas de droite:

1. un coefficient directeur supérieur à 1
2. un coefficient directeur compris entre 0 et 1
3. un coefficient directeur inférieur à -1
4. un coefficient directeur compris entre 0 et -1

Cet algorithme nous a simplement permis de comprendre le fonctionnement en python de la méthode de Bresenham, afin de pouvoir la manipuler correctement pour la suite, ainsi que l'affichage d'une matrice (avec un jeu de couleurs) grâce à la fonction "display _ board".

Version 2 Dans cette deuxième version, nous avons élargi l'algorithme précédent en divisant la matrice en 8 octants (cf fig.1 p.5). Pour cela, nous avons modifié la fonction "line _ replace" en intégrant dans chacun des 4 cas précédents un deuxième sous-programme qui permet de prolonger la droite tracée dans l'octant opposé, de manière symétrique.
Nous avons ainsi placé un personnage virtuel au centre de la matrice, et

avons amélioré notre premier algorithme pour faire partir un certain nombres de droites du centre dans les 8 cadrants avec des coefficients directeurs proportionnels (grâce à de simples relations trigonométriques entre angles et cosinus), avec une nouvelle fonction "field _ of _ view". Cela a donc permis d'élaborer un champ de vision (ensemble de tous les points vus autour d'un personnage).
Avec cette version, nous commençons déjà à percevoir à quoi peut ressembler la version finale de notre algorithme.
Cependant, nous apercevons nos premiers bugs : tout d'abord, ce programme ne traite que les matrices carrées et ne peut tracer qu'environ une trentaine de droites.

Version 3 Cette nouvelle version est une version de transition qui permet simplement de créer une limite au champs de vision, choisie arbitrairement: $limite = \frac{2}{5} \times min(longueur, largeur)$, grâce à l'ajout d'un compteur dans la fonction principale "line _ replace".
On introduit également la fonction "put _ walls" qui permet de mettre des murs aléatoirement sur la matrice, mais on ne s'occupe pas encore de la collision. Cependant, même si cette version permet de résoudre le problème du nombre maximal de droites, nous aimerions avoir une limite de vue circulaire.

Version 4 On peut observer une version presque finale de l'algorithme dans cette partie. En effet, cette version traite la collision des droites et des murs. Pour cela, on rajoute une condition simple dans la fonction "line _ replace", si la droite rencontre un pixel représentant un mur (soit dans cette version un 2) la boucle se casse, et l'on passa à la représentation graphique de la droite suivante.
On définit également un nombre de droites à tracer en fonction de la taille de la matrice, afin de pouvoir recouvrir entièrement la surface étudiée, que l'on fixe avec l'expérience à 3/2 du minimum entre la longueur et la largeur de la matrice, avec la fonction "line _ number".
Cependant, certains bugs comme l'apparition problématique d'un mur à coté du personnage, subsistent.

Version 5 (Algo1) Dans cette dernière version, nous avons cherché à améliorer au maximum la version précédente de notre algorithme. Nous avons pour cela résolu notre problème de limite carrée à l'aide tout simplement d'une équation de cercle, et avons rajouté la possibilité de mettre des "vitres" sur nos matrices, dont on peut regarder au travers. Ces dernières nous seront utiles dans les parties suivantes.
Nous avons également proposé une fonction permettant de placer nous même des murs aux endroits souhaités, soit la fonction "put _ wall", qui permet

de placer des murs opaques ainsi que des vitres.
Enfin, nous avons simplifié la fonction principale "line _ replace" pour la rendre plus lisible.

2.3 L'élaboration d'un algorithme plus performant

L'algorithme précédent était donc un test afin de nous familiariser avec le problème. Toutefois, de nombreux problèmes et bugs sont apparus et malgré la volonté de les résoudre, certains sont restés non élucidés (problème du mur collé au personnage, matrices carrées uniquement).
Nous avons ainsi cherché un algorithme plus avancé permettant de résoudre tous les problèmes rencontrés avec le précédent.
Après quelques recherches, nous avons trouvé un algorithme. Il utilise la même méthode mais en un temps moindre (et en moins de lignes, ce qui n'est pas négligeable). En effet, le fait de travailler avec des entiers plutôt qu'avec des flottants accélère les calculs sur python.
Nous avons de plus rajouter des tests permettant de vérifier que pour deux points A et B, les droites (AB) et (BA) sont bien les mêmes.
Vous pourrez trouver ce programme dans le code "Algo2+image+luminosité.py", dans la partie *"Tracer les lignes de vues avec l'algorithme de Bresenham"*. Il utilise cependant des fonctions et notions que nous expliquons dans la partie 3 p.9.

Fonctionnement de l'algorithme Le principe repose aussi sur l'algorithme de Bresenham. Toutefois,les structures conditionnelles ne sont pas les mêmes afin de se ramener à des calculs avec des entiers (cf A.1 p.17).
De plus, les symétries par rapports aux axes se font simplement en échangeant les coordonnées des points A et B selon l'octant dans lequel on veut se placer. Ainsi, on peut traiter tous les octants en ne faisant qu'une seule boucle dans la fonction.
Cet algorithme ne modifie plus une matrice, il retourne une liste contenant les coordonnées de tous les points vus. La taille de cette liste est $max(|x_B - x_A|, |y_B - y_A|)$ (cf A.1 p.17).

Ainsi, nous sommes maintenant arrivés à un algorithme permettant de savoir "tout ce que le personnage voit". Notre problème suivant sera d'afficher ce résultat en image. On verra de plus, plus tard que cet algorithme n'est pas le plus performant et que l'on peut trouver des alternatives, plus rapides.

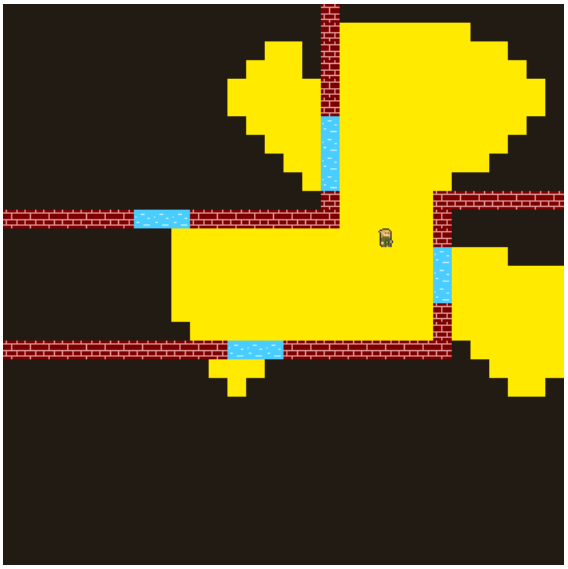


Figure 3: Representation en image

3 Représenter une image

Une fois notre algorithme de ligne de vue mis en place, il était donc temps à présent de pouvoir représenter notre matrice en tant qu'image avec des pixels. Après avoir ainsi résolu le problème de coder des droites, nous voulions afficher le résultat de façon plus agréable. En effet, l'affichage en matrices que nous utilisions pouvait être amélioré. Pour cela nous avons trouvé un module python: Pillow.

3.1 Un affichage grâce au module PIL

La bibliothèque Pillow (ou PIL pour Python Imagin Library) est un module sur Python permettant le traitement et la manipulation d'images telle que la manipulation des pixels, l'affichage d'une image ou encore la réalisation de différents effets que l'on verra par la suite.

Tout d'abord, l'affichage de l'image fut ici notre premier objectif. Pour cela, nous avons commencé par récupérer ce que l'on appelle des "sprites" c'est-à-dire les images représentant respectivement le personnage, les murs, les vitres et la ligne de vue. On doit toutes les créer sur une même image (sprites _ proj) pour accéder à chaque pixel le plus rapidement et aisément possible.

Sur python, On parcourt la matrice d'affichage réalisée à partir des algorithmes vus dans la partie I. En effet, à chaque point de la matrice d'affichage est associée une valeur: "m" pour mur, "@" pour le personnage, etc.. Ces

valeurs ont été associées à la matrice grâce aux différentes fonctions que nous avons réalisées permettant de tracer des droites, des murs et des vitres comme vu précédemment. Ainsi, à chaque valeur, on associe un sprite qui lui correspond que l'on copie sur une nouvelle image (vide au départ, aux mêmes coordonnées que celles de la matrice). Pour cela, nous avons donc redimensionné les images des sprites par rapport à la taille d'un pixel pour avoir une cohérence dans la représentation de celle-ci. Il a donc été nécessaire d'utiliser les fonctions provenant de Pillow pour ces différentes actions telles que ouvrir une image, redimensionner celle-ci ou en créer une nouvelle à partir d'une autre. C'est ainsi que nous avons obtenu le résultat observable fig.3 p.9.

L'algorithme était jusque là suffisant pour représenter notre matrice. Mais nous avons donc cherché à affiner ce dernier afin d'observer plus de détails et se rapprocher un peu plus de la représentation en 2D d'un jeu vidéo, ou du moins d'une image tirée de ce dernier.

3.2 La baisse de la luminosité

Ainsi, notre second objectif fut d'ajouter des détails à cette image pour qu'elle soit plus réaliste/ressemblante à un jeu vidéo. En effet, sur notre première version, seule l'image sans détail précis s'affichait. On a donc décidé d'effectuer des changements de luminosité sur la ligne de vue du personnage: "les pixels jaunes".

La première étape fut de comprendre le fonctionnement et les différents outils de Pillow concernant la modification des pixels. Nous avons notamment commencé par baisser la luminosité globale de l'image. Pour cela, nous avons parcouru chaque pixel (qui sont composés de trois valeurs compris entre 0 et 255 pour le rouge, le vert et le bleu) et baissé leurs valeurs grâce différentes fonctions de Pillow telles que getpixel pour récupérer la valeur d'un pixel et putpixel pour changer le pixel en question. Ensuite, pour baisser la luminosité seulement sur la ligne de vue, la tâche fut plus complexe. Pour cela, on a tout d'abord récupéré la distance d'un point à partir du personnage avec la fonction comput _ fog. Le principe étant de réaliser un brouillard volumétrique: à partir du personnage, on associe à chaque point de la ligne de vue un coefficient compris entre 0 à 1 grâce une fonction exponentiellement décroissante (expfog). Ainsi, chaque point appartenant à la ligne de vue dans la matrice d'affichage aura une valeur différente, nécessaire pour l'étape d'après.

De plus, nous avons modifié l'algorithme de ligne de vue lors de la rencontre avec une vitre. En effet, la luminosité doit être diminuée d'avantage lors de la rencontre avec une vitre et on doit donc diminuer le coefficient associé. On ajoute donc un compteur pour le nombre de vitre rencontrée. A celui-ci, on associe la fonction précédente calculant le coefficient du "brouillard" et on enlève à cela le nombre de vitres rencontrées avec un facteur à

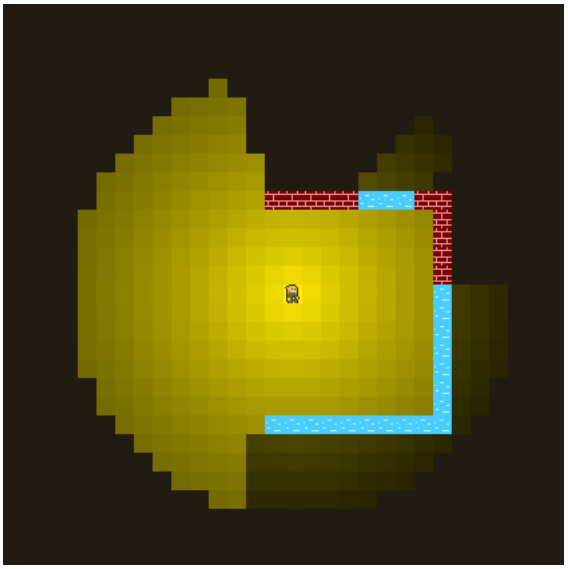


Figure 4: Image avec baisse de la luminosité

0,3 qui semblait correct dans l'affichage.

Enfin, dans notre fonction permettant l'affichage de notre image, nous nous sommes servis de la fonction ImageEnhance du module Pillow. En effet, on associe tout d'abord le sprite "jaune" lors de la rencontre avec le visible pour ensuite baisser la luminosité de ce pixel à partir du coefficient associé à celui-ci dans la matrice d'affichage, calculé précédemment. La luminosité va donc être modifiée au fur et à mesure grâce à ces différents coefficients. Il est important de noter que si nous ne sommes pas dans le cas "visible", la luminosité n'est pas changée et on colle sur l'image vide nos différents sprites.

Ainsi, notre algorithme d'affichage a été très profondément modifié suite à la baisse de la luminosité: on crée un dictionnaire pour les valeurs de la matrice d'affichage pour augmenter la rapidité et la performance. Comme la ligne de vue étant le seul "flottant", on baisse la luminosité seulement lors de la rencontre de ce-dernier, sinon, on colle le sprite. On peut observer un exemple d'application sur la figure 4 p.11.

Ainsi, nous sommes ici arrivés à notre objectif d'afficher une image et nous avons pu ajouter des détails. Notre dernier objectif est donc de rechercher comment améliorer la détermination du champ de vision, afin d'obtenir un résultat plus rapidement qui pourrait être implémenté dans un jeu vidéo.

Figure 5: Polygone de ligne de vue - extrait de *https* : *//ncase.me/sight* − *and* − *light/*

4 Peut on voir un point

En repartant à la problématique générale du projet, nous avons envisagé une autre méthode pour trouver et afficher ce qu'un personnage de jeu vidéo en 2D voyait. Cette nouvelle méthode est intéressante car elle pourrait avoir une complexité moins élevée et donc être plus rapide que notre premier algorithme. En effet, pour déterminer "tout ce qu'on voit" avec la méthode de la partie 2, p.4, on doit tracer beaucoup de droites et donc une complexité trop élevée (cf B.2 p.18). Cette nouvelle méthode repose sur la détermination de ce qu'on appelle un polygone de ligne de vue (Figure 5 p.12 en rouge) qui correspond à l'ensemble des pixels (l'espace étant défini en pixel) que le personnage voit.

4.1 Le polygone de ligne de vue

Pour construire géométriquement ce polygone, nous allons d'abord déterminer quels sont les objets, en l'occurrence des murs ou des vitres, présents dans le champ de vision du personnage. Nous considérerons que les objets placés sont des polygones et que le personnage a un champ de vision s'assimilant à un carré et non à un cercle (on étudie un cas simple).

On peut trouver quels sont ces objets présent dans le champ de vision du personnage en utilisant un algorithme de tri comme le AABB-tree(cf 4.3 p.15 pour son fonctionnement). Une fois que les polygones potentiellement vu par le personnage sont trouvés, on regarde s'il y a une intersection entre la ligne de vue du personnage et les côtés de ces polygones. Pour simplifier cette étape on effectue une triangulation (expliquée ci-dessous) de chaque

polygone c'est à dire qu'on les divise en triangles. On étudie alors les intersections entre chacun des côtés du triangle et la ligne de vue du personnage. Cette étude est détaillée dans les parties 4.2 p.13 et 4.3 p.14. Une fois que l'on a déterminé quelles sont les faces des triangles visibles par le personnage, on trace les segments passant par les extrémités des faces visibles qui vont du personnage jusqu'à un point assimilé à la limite de vision ou, dans certains cas, jusqu'à un autre objet (dans le cas ou un autre objet se situerais dans le prolongement du segment). On trace également les segments entre le personnage et les quatre coins du champ de vision. Si un objet se trouve entre le personnage et un coin du champ de vision, on trace le segment jusqu'à l'objet mais pas plus loin. De cette façon, la totalité de l'espace vu par le personnage est divisée en triangles dont on peut connaître les coordonnées des sommets dans la matrice. En prenant les coordonnées de chaque sommet de chacun des triangles, excepté les coordonnées du personnage et en faisant attention à ne pas prendre deux fois les mêmes points (un sommet d'un triangle peut être un sommet d'un autre triangle), on peut obtenir les coordonnées de tous les sommets de notre polygone de ligne de vue. Nous pouvons donc enfin le tracer ce qui nous donne une image approximativement similaire à la figure 5 p.12. Cette méthode serait plus rapide que celle de la partie 2 p.4 ou on doit tracer énormément de droites.

Triangulation d'un polygone Il s'agit d'une méthode permettant de diviser un polygone simple en plusieurs triangles. Pour l'utiliser, nous comptons prendre un algorithme déjà existant. Il existe notamment l'algorithme qui utilise la "méthode des oreilles". Cet algorithme consiste à soustraire petit à petit au polygone, des triangles, qui possèdent deux côtés communs avec ceux du polygone. Cependant, cet algorithme à une complexité en $O(n^2)$ donc il ne permettrait pas d'améliorer la complexité de notre algorithme principal. Un autre algorithme qui semble plus pertinent, puisque sa complexité est en $O(nlog(n))$, est celui utilisant la méthode de la "décomposition en chaînes monotones". Malheureusement, nous éprouvons des difficultés à la compréhension de ce dernier.

4.2 Le calcul d'intersections

Enfin, notre objectif ici est de savoir si un personnage peut voir un point de l'espace situé par ses coordonnées dans le plan. D'un point de vue pratique, il est impossible de voir un point s'il y a un objet, un obstacle entre le personnage et le point en question. Nous avons donc construit un algorithme qui permet de vérifier l'intersection entre deux segments.

Dans cette partie tous nos objets seront représentés par des triangles (dont nous expliquerons la raison ci-dessous). Posons A le point correspondant au personnage et B le point que l'on veut voir. Connaissant les coor-

données de ces points, on peut aisément déterminer $\vec{AB}$. Nous définissons nos objets et obstacles par une liste de sommets, ce qui nous permet de connaître le vecteur correspondants à chaque côté des triangles. Grâce à un prétraitement des coordonnées des sommets des triangles, on construit un dictionnaire associant à chaque sommet du triangle le vecteur permettant d'aller au point suivant.

Dans cette partie, nous considéreront un triangle EFG.

Pour savoir si le personnage voit le point nous allons simplement vérifier s'il existe une intersection entre $[AB]$ et un des côtés du triangle. Pour cela on utilise les équations paramétriques des droites supportées par $\vec{AB}$, $\vec{EF}$, $\vec{FG}$ et $\vec{GH}$. pour la droite (EF) on a par exemple :

$$(EF) : \begin{pmatrix} x_{\vec{EF}} \\ y_{\vec{EF}} \end{pmatrix} \times t + \begin{pmatrix} x_E \\ y_E \end{pmatrix}, t \in \mathbb{R}$$

$$(AB) : \begin{pmatrix} x_{\vec{AB}} \\ y_{\vec{AB}} \end{pmatrix} \times t' + \begin{pmatrix} x_A \\ y_A \end{pmatrix}, t' \in \mathbb{R}$$

Après avoir vérifié que les deux droites ne sont pas parallèles, on peut déterminer l'intersection entre (AB) et (EF). En décomposant les droites selon les axes (Ox) et (Oy) on peut exprimer t en fonction de t' et calculer t' grâce à l'autre équation. On remplace finalement dans la première équation et on détermine t. Si $0 < t < 1$ et $t' > 0$ on a bien intersection entre $[AB]$ et $[EF]$. En effet si on a $0 < t < 1$ alors l'intersection se situe entre E et F. La condition $t' > 0$ permet de s'assurer que l'intersection a bien lieu sur $[AB)$

En réalisant cette vérification pour tous les côtés des triangles on peut savoir si le personnage voit le point ou non. Cette méthode est pratique de part sa simplicité mais accuse une complexité en $O(n)$ ou n représente le nombre de triangles, ce qui est trop élevé pour pouvoir être utilisé dans un jeu.

4.3 Comment améliorer la recherche d'intersection ?

Dans cette partie on va supposer que le nombre d'objets autour de notre personnage est très important: une ville ou une forêt. On va donc s'intéresser à deux méthodes pour ranger les objets de manière à ne regarder que ceux qui nous intéressent, c'est à dire ceux qui sont proches de notre personnage.

Les Quad-Trees Les Quad-trees sont tout d'abord une structure de donnée en arbre consistant à diviser l'espace en 4 parties égales, puis chaque sous parties encore en quatre et ainsi de suite jusqu'à n'avoir plus que quelques objets par sous parties. De cette manière on peut ranger les objets en fonction de leur position dans l'espace. Cette structure de donnée peut être

représentée par un arbre : à chaque subdivision de l'espace on crée un noeud avec quatre branches qui nous permettent de ranger les objets. De cette manière, tout en bas de l'arbre il ne reste que des petites subdivisions de l'espace avec quelques objets dedans. L'avantage de cette méthode est qu'une fois la création de l'arbre réalisée, on peut connaître les objets à proximité avec une complexité en $O(ln(n))$. Cette complexité aillant l'air formidable, reste à discuter puisqu'elle est dite "en moyenne" (c'est à dire que la complexité dépend de l'organisation des objets dans l'espace et le $O(ln(n))$ n'est atteint que si les objets sont répartis assez uniformément dans l'espace, ce qui est en moyenne le cas). Pour pallier à ce problème, les AABB-trees ont donc été mis au point.

les AABB-trees Ensuite, les AABB-trees sont un type de structure de donnée qui a aussi pour but de ranger les objets selon leur position dans l'espace et peut aussi être représenté par un arbre. La différence avec le quad-tree est que l'espace n'est plus découpé équitablement en 4 mais en deux parties de manière à avoir cinquante pour cent des objets de l'espace d'un côté et cinquante pour cent dans l'autre. Pour connaître où se situe la médiane, il existe un algorithme en complexité linéaire ($O(n)$) qui permet de la calculer. De cette manière, on peut créer un arbre en séparant toujours le nombre d'objet en 2. Cela permet d'avoir une complexité effective en $O(ln(n))$ quel que soit la disposition des objets.

On peut ainsi se demander, quelle est la réelle efficacité de ce traitement? Même si construire les arbres peut être long, il n'est nécessaire de créer l'arbre qu'une seule fois et de le garder en mémoire. Cela permet de savoir en temps logarithmique quels objets sont situés à côté de notre personnage, la où il faudrait un temps linéaire pour vérifier l'intersection avec tous les triangles. Par exemple, avec un AABB-tree, s'il faut 2048 secondes pour vérifier l'intersection avec tous les triangles, il n'en faut pas plus de 10 pour savoir quels sont les objets à proximité de notre personnage.

5 Conclusion

Enfin, tout au long de notre projet, nous avons pu apprendre et construire différentes méthodes pour savoir si dans un monde virtuel un personnage voit un objet ou non.

Notre travail à donc débuté par la compréhension puis la mise en place de l'algorithme établi par Bresenham. Nous avons pu construire notre propre algorithme en se basant sur cette méthode. Néanmoins, de nombreux bugs sont apparus au fur et à mesure qui ont pu légèrement nous freiner et nous faire perdre du temps mais qui ont en revanche servi à nous pousser d'avantage sur le perfectionnement de nos algorithmes. Différentes étapes se sont suivies: la création de la matrice, la mise en place de mur et de vitres, etc...

Lorsque notre matrice fut terminée, nous avions donc décidé de la représenter de manière plus réaliste, à la manière d'un réel jeu en 2D. Ce nouvel objectif nous a permet d'en apprendre énormément sur la manipulation d'images et de pixels sous python qui peut s'avérer très utile lorsque l'on travaillera prochainement sur d'autres images.

Enfin, pour ne pas seulement se limiter au 2D, nous avons effectué de nombreuses recherches pour comprendre comment fonctionne de manière rapide et efficace les algorithmes de recherches d'un objet dans un monde non pas seulement en 2D, mais également en 3D. Les algorithmes étant longs et très complexes à mettre en place, nous avons effectué de nombreuses recherches pour comprendre leur fonctionnement et cela nous a permis de nous donner un avant-goût pour l'année prochaine avec les "arbres" par exemple.

Ainsi, ce projet à su nous apprendre à travailler en groupe, de pouvoir se répartir les tâches efficacement et de s'écouter les uns et les autres afin d'avancer le plus efficacement possible. Il nous a également permis de renforcer nos méthodes de travail, le sérieux et la rigueur au vu des consignes exigées. Nous avons pu en apprendre énormément d'un point de vue scientifique en informatique principalement mais aussi en mathématiques. D'un point de vue professionnel, ce projet nous a permis pour chacun d'entre nous de se donner une nouvelle perspective concernant le travail dans le domaine de l'informatique et de pouvoir affiner nos choix pour le futur.

A L'élaboration d'un algorithme plus performant

A.1 Des calculs avec des entiers

On a les conditions suivantes:

- $e_0 = 0$
- à chaque x += 1, e += m
- des que $e > 0,5$, y += 1 et e-= 1

On pose $e' = e - 0.5$

- $e'_0 = -0.5$
- à chaque x += 1, e' += m
- des que $e > 0,5$ devient $e' > 0$, y += 1 et e' -= 1

On continue le raisonnement en posant $e'' = -2 \times e' \times dx$ avec $dx = x_B - x_A$

- $e''_0 = e'_0 \times -2 \times dx = -0.5 \times -2 \times dx = dx$
- à chaque x += 1, $e'' += m \times -2dx = -2dy$
- des que $e'' = -2e'dx < 0$ (car $-2dx < 0$ dans l'octant considéré), y += 1 et e += 2dx

Ainsi, On a maintenant des calculs utilisant des entiers relatifs et non des nombres flottants ce qui accélère les calculs sur python.

Taille d'une droite Avec l'algorithme de Bresenham, on allume un seul pixel à chaque itération. Si $|x_B - x_A| > |y_B - y_A|$, la boucle du programme fait $|x_B - x_A|$ itérations. Sinon, la boucle fait $|y_B - y_A|$ itérations. Ainsi, la taille de la liste est $max(|x_B - x_A|, |y_B - y_A|)$.

B Les calculs de complexité

Nous détaillerons ici les calculs des complexités de chaque fonction de nos algorithmes.

B.1 Algo1

create_ matrix $O(m)$ avec m le nombre de colonnes de la matrices

display_ board $O(n) \times O(m) = O(nm)$ avec n le nombre de lignes et m le nombre de colonnes de la matrice

put_ wall $O(l)$ avec l la taille du mur.

line_ replace (1,2,3,4) $O(n-x)+0(n-x) = O(n-x)$ avec n le nombre de lignes et x la position initiale du personnage selon l'axe des x.

line_ replace (5,6) $O(min(x,y)) + O(min(x,y)) = O(min(x,y))$ avec x et y les positions initiales du personnages respectivement selon l'axe des x et selon l'axe des y.

final_ replace $O(n-x) \times 4 + O(min(x,y)) \times 2 = O(n-x+min(x,y))$ avec n le nombre de lignes, x et y les positions initiales du personnages respectivement selon l'axe des x et selon l'axe des y. On ne peut pas simplifier cette complexité car il n'y en a pas une qui soit toujours plus grande que l'autre, cela dépend des cas.

line _ number $O(1)$, on effectue un simple calcul.

field_ of_ view $O(k) \times (O(n-x+min(x,y)) = O(k(n-x+min(x,y)))$ avec k le nombre de droites à tracer, n le nombre de lignes, x et y les positions initiales du personnages respectivement selon l'axe des x et selon l'axe des y.

Versions précédentes Pour les autres versions, les complexités des fonctions sont exactement les mêmes, hormis pour la toute première version où la complexité de "line_ number" est en $O(n) \times O(m) = O(nm)$ avec n le nombre de lignes et m le nombre de colonnes de la matrice.

B.2 Algo2+image+luminosité

line_ first_ octant Pour une droite (AB), $O(n)$ avec $n = x_B - x_A$

line_ of_ sight Ici, la complexité dépend de l'octant auquel la droite appartient. En effet, la complexité est

$$O(nombre\_de\_points) + O(nombre\_de\_points) = O(nombre\_de\_points)$$

avec $nombre\_de\_points = max(|x_B - x_A|, |y_B - y_A|)$. En effet, on a parcourt une première boucle nombre_ de_ points fois. On parcourt une seconde boucle parmit les éléments de la liste de points donc de longueur nombre_ de_ points. Si la droite (AB) se trouve dans les octants 2, 3, 6 ou 7, $nombre\_de\_points = |y_B - y_A|$, la complexité est $O(|y_B - y_A|)$.
Sinon, $nombre\_de\_points = |x_B - x_A|$ et la complexité est $O(|x_B - x_A|)$.

champ_ de_ vision Ici, on parcourt une première boucle n fois avec n le nombre de lignes de la matrice de terrain, puis une seconde boucle m fois avec m le nombre de colonnes de la matrice. Dans chaque boucle, on exécute la fonction line_ of_ sight de complexité $O(nombre\_de\_points)$.
Cela nous donne une complexité totale de

$$O(n \times nombre\_de\_points) + O(m \times nombre\_de\_points)$$

On a $nb\_de\_points = max(|x_B - x_A|, |y_B - y_A|)$ pour chaque droite tracée. On peut de plus ajouter que la taille maximale d'une droite, en partant du centre et en supposant que la matrice de terrain est carrée est $\frac{n}{2}$. On à donc une complexité maximale de $O(\frac{n}{2}) = O(n)$ dans ces conditions, avec n la dimension de la matrice.

superpose On parcourt une boucle for sur la taille de la liste contenant les points vus. On a donc une complexité en $O(nb\_de\_points)$. avec $nb\_de\_points = max(|x_B - x_A|, |y_B - y_A|)$.

superpose_ champ_ de_ vision On parcourt une première boucle sur le nombre de droites tracées soit $n+m$ pour une matrice de dimension $n \times m$. On parcourt une deuxième boucle sur le nombre de points sur la droite soit $nb\_de\_points = max(|x_B - x_A|, |y_B - y_A|)$. Cela donne une complexité totale de

$$O(n+m) \times O(nb\_de\_points) = 0((n+m) \times nb\_de\_points)$$

Pour une matrice carrée avec le personnage au centre, on a donc $nb\_de\_points = \frac{n}{2}$ dans le cas le pire. Ce qui donne une complexité maximale de $O(2n \times \frac{n}{2}) = 0(n^2)$

Étant donné que toutes ces fonctions utilisent presque uniquement des boucles for, on n'a pas de problèmes pour ce qui est des finitudes des fonctions.

Pour ce qui est des fonctions utilisant la bibliothèque pillow, ne connaissant pas les complexités des fonctions prédéfinies, nous ne pouvons pas calculer les complexités des fonctions.

Ainsi, si l'on veut établir un champ de vision autour d'un personnage, On doit utiliser successivement les fonctions *create_matrix*, *champ_de_vision* et *superpose_champ_de_vision*. Pour une matrice carrée de dimension n en plaçant le personnage au milieu de la matrice on trouve en additionnant les complexités:

$$O(n) + O(n) + O(n^2) = O(n^2)$$

Cette complexité est malheureusement trop élevée pour être utilisé dans un jeu vidéo ou l'on est proche des 60 images par secondes.

List of Figures

References

[1] ***SIGHT & LIGHT how to create 2D visibility/shadow effects for your game***
mis à jour le 22/06/2015,
Disponible sur : https://ncase.me/sight-and-light/

[2] https://github.com/ncase/sight-and-light, lien github lié au site précédent, juin 2015 (dernière modification en date)

[3] Red Blob Games, ***2d visibility***
mis à jour 03/2018,
Disponible sur: https://www.redblobgames.com/articles/visibility/

[4] ***Algorithme de tracé de Bresenham***
mis à jour 03/2019,
Disponible sur : https://fr.wikipedia.org/wiki/Algorithme_de_tracé_de_segment_de_Bresenham

[5] ***Bresenham's Line Algorithme***
mis à jour 06/2018,
Disponible sur: http://www.roguebasin.com/index.php?title=Bresenham%27s_Line_Algorithm

[6] ***Pillow - l'informatique c'est fantastique!***
mis à jour 12/2017,
Disponible sur: http://info.blaisepascal.fr/pillow

[7]

Durant ce projet informatique, nous avons tout d'abord cherché comment coder une ligne de vue, en python et de manière efficace. Nous avons ensuite voulu représenter une image, pour se rapprocher de la réalité des jeux vidéos. Enfin, nous avons voulu améliorer nos techniques de détermination de lignes de vues et essayer de comprendre les méthodes actuelles qui permettent de faire ce travail le plus efficacement possible.

Mots-clés de projet: **Bresenham - Pixels - Droite - Image - Intersections**