Algorithmique et Programmation

—

Projets 2011/2012

—

G1: monasse*(at)*imagine.enpc.fr G2: boulc-ha*(at)*imagine.enpc.fr
G3: yoann*(at)*le-bars.net G4: arnaud.carayol*(at)*univ-mlv.fr
G5: liuz*(at)*imagine.enpc.fr G6: vialette*(at)*univ-mlv.fr

1 Duel Tetris

1.1 Introduction

L'objet de ce projet est de présenter une extension au fameux jeu Tetris. cette extension est une intelligence artificielle qui joue au jeu en parallèle au joueur. Le but du jeu est :

- de comprendre les règles pour essayer de battre la machine,
- de construire une intelligence artificielle telle qu'il devienne extrêmement difficile de battre la machine,
- et enfin, en option, que la force de cette intelligence puisse être réglée (par exemple en utilisant un paramètre de 1 à 10).

Titre	**Année**	**Type de jeu**	**sous-type**
OXO	1952	IHM de jeu de Réflexion	Tic-tac-toe
PONG	1972	jeu de Sport	ping-pong
Space Invaders	1978	jeu de Tir	Shoot them up
Pac Man	1979	jeu de Labyrinthe	Plate-forme
Tetris	1984	jeu de Réflexion	Puzzle
Duel Tetris	2010	Jeu de Reflex	Puzzle

1.2 L'interface Homme-Machine

Dans cette partie, il s'agit de coder une interface pour jouer à Tetris. Un exemple peut être vu en figure 1.

1.2.1 Rappel des règles

- Le jeu contient 7 formes de bases (les tétrominos, c'est-à-dire les figures géométriques composées de quatre carrés) ayant chacune une couleur différente : *le carré (O), le bâton (I), le té (T), le èl (L), le èl inversé ou Gamma (J), le biais (Z) et le biais inversé (S)*.
- Ces formes descendent une par une dans un *puits* (*dix* cases de largeur et *vingt-deux* de haut).
- Le but du jeu étant de former des *lignes*.
- Toutes les *dix* lignes remplies, la vitesse de descente s'accélère d'un facteur ($1,1$ par exemple).

Le jeu se termine lorsqu'une pièce vient se poser sur une autre pièce qui a déjà atteint la hauteur maximum (c-à-d 22).

1.2.2 Codage de l'interface

Votre interface doit contenir une fenêtre graphique suffisamment grande pour contenir deux jeux de Tetris.

Chaque jeux de Tetris doit alors être composé d'une vue sur la pièce suivante (qui tombera dans le puits après celle déjà en chute (placée en haut par exemple).

Il doit être possible

- d'accélérer la vitesse de chute des pièces (d'un facteur deux par exemple)
- et de faire tourner les pièces d'un quart de tour.

1.3 L'Intelligence Artificielle

Dans cette partie, il s'agit de coder une interface pour jouer à Tetris.

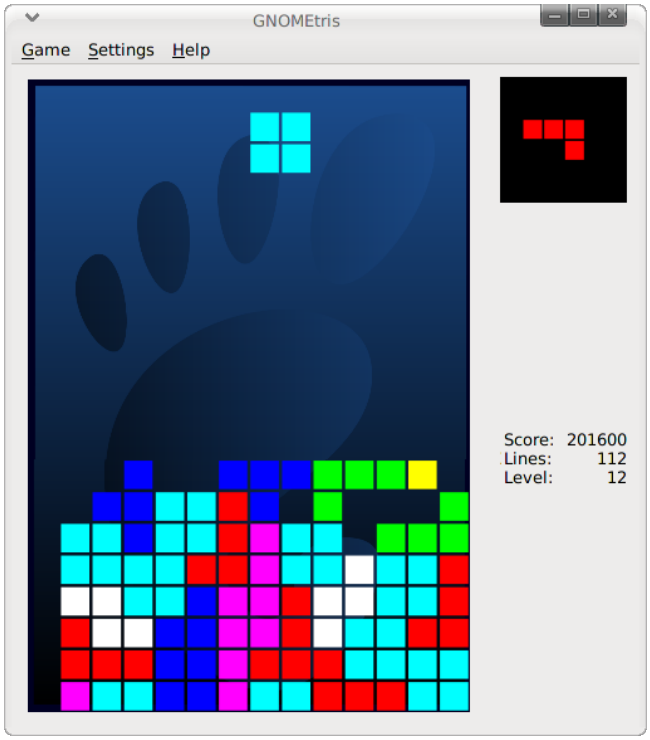


FIGURE 1 – Une interface de Tetris.

1.3.1 Ajout de règles

Les règles énoncées précédemment ne présentent pas de notion de score.

En effet, comme le Tetris qui va être codé ici va être un *défi* entre l'homme et la machine, il n'y a pas besoin de score.

Par contre, nous allons ajouter la règle suivante, lorsque l'un des deux concurrents (homme ou machine) réussi à faire une ligne, celle-ci est bien retirée de son puits, mais elle va alors s'ajouter à la base du puits de l'adversaire mais ensuite cinq carrés seront retirés aléatoirement (si on envoie 2 lignes en même temps seulement quatre carrés seront retirés par lignes, pour 3 lignes, ce sera trois carrés par ligne et pour un envoie de 4 lignes, seulement deux carrés seront retirés aléatoirement par ligne).

Le jeu devient alors plus un jeu de reflex que de réflexion car le but devient d'envoyer le plus rapidement des lignes (même une seule) chez l'adversaire.

1.3.2 Mise en place de stratégies de décision

Vous êtes libres quant aux choix de stratégies mise en place par la machine pour former des lignes. Néanmoins, une modélisation objet est recommandé pour les tétrominos, chaque tétromino possède 4 représentations graphiques issues de ces transformation par la rotation de $\frac{\pi}{2}$, une méthode permet d'effectuer le déplacement du tétromino, une méthode permet sa rotation.

À chaque tétromino sont associées des configurations de lignes pour lesquelles son placement est optimum, et d'autre pour lequel son placement est plutôt bon. Vous devez classer ses positionnements possibles et choisir le placement qui a le meilleur score pour la configuration actuelle des lignes dans le puits (l'idéal étant que pour le placement, la ligne de surface ait un impact sur le score mais que soit également pris en compte la configuration des 2 ou 3 lignes sous la surface pour ne pas 'boucher' des places éventuellement disponibles pour une autre pièce, d'autant plus si c'est la pièce qui arrive juste après...).

2 Construction de Panorama

2.1 Introduction

Si on a un ensemble d'images prises par une caméra tournante les images sont liées les unes aux autres par une transformation à peu de paramètres. On va donc pouvoir transformer chaque image vers l'espace d'une image de référence et on obtiendra ainsi une image unique offrant un très grand champ angulaire comme montré en Figure 2.

Lorsque deux caméras sont uniquement en rotation l'une par rapport à l'autre et sans translation (le centre optique ne bouge pas) alors la coordonnée d'un point x dans l'image 1 visualisant un point $3D$ P de la scène, se

FIGURE 2 – Exemples de reconstitution de panorama. En bas : reconstruction cylindrique d'une vue à 360°.

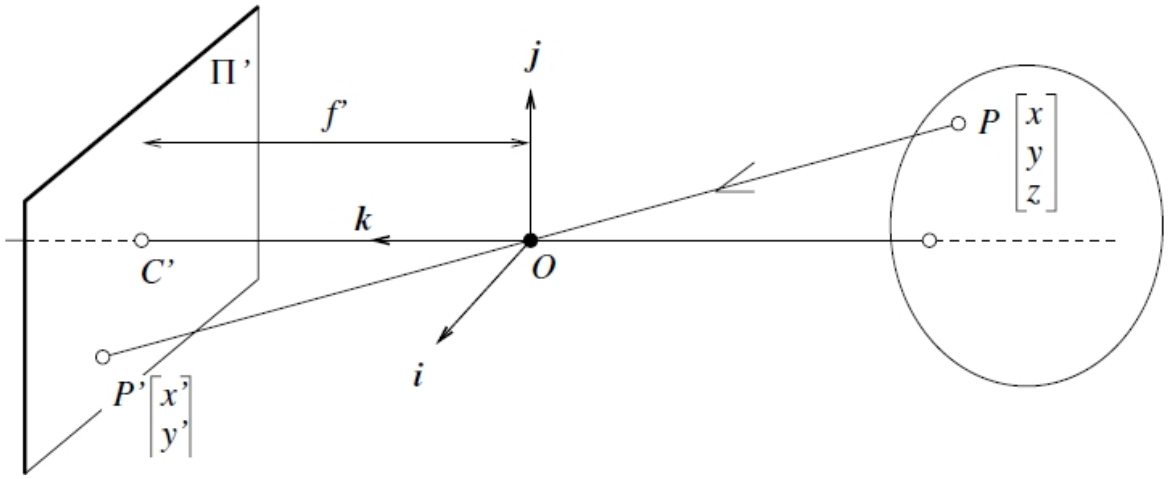


FIGURE 3 – Modéle de caméra pinhole

déduit du point x' dans l'image 2 correspondant au même point P. On a en fait (pour simplifier) $x' = Hx$. Ainsi pour une série d'image prise par une caméra tournante on en déduit à-partir des mises en correspondance de points des mises en correspondance d'images grâce aux homographies H. De cette façon on va pouvoir rectifier toutes les images pour les mettre en correspondance avec une image de référence choisie (a priori l'image centrale en terme de point de vue angulaire dans la série de photos) et ainsi reconstruire un panorama.

Notez que comme souvent en informatique, nous importons le vocabulaire anglo-saxon et employons ici le terme "caméra" au sens appareil photographique.

2.2 Un peu de Théorie : caméras en rotation

Nous supposons que la caméra est un modèle parfait *pinhole*, [1] dans ce cas dans le repère attaché à la caméra (x et y dans le sens des pixels de l'image, z orthogonal au plan image) on a une transformation simple qui est représentée dans la Figure 3.

En utilisant le théorème de Thalès on obtient :

$$\begin{cases} x' &= \frac{f'x}{z} \\ y' &= \frac{f'y}{z} \\ z' &= -f' \end{cases}$$

1. la traduction française est "sténopée", mais est rarement employée

Si l'origine dans le plan image est dans le coin en haut à gauche de l'image comme c'est souvent le cas on obtient alors :

$$\begin{cases} x' &= \frac{f'x}{z} + u_0 \\ y' &= \frac{f'y}{z} + v_0 \end{cases},$$

où (u_0, v_0) sont les coordonnées pixel du point C'.

Il suffit de multiplier ces équations par z pour obtenir un système linéaire :

$$\begin{pmatrix} u \\ v \\ w \end{pmatrix} := \begin{pmatrix} x'z \\ y'z \\ z \end{pmatrix} = \begin{pmatrix} f' & 0 & u_0 \\ 0 & f' & v_0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = K \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

Notez que $x' = u/w$ et $y' = v/w$.

Maintenant si le repère est toujours centré au centre optique de la caméra mais avec une rotation on obtient :

$$\begin{pmatrix} u \\ v \\ w \end{pmatrix} = KR \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

et pour la seconde caméra qui à le même centre optique mais une autre rotation on a :

$$\begin{pmatrix} u' \\ v' \\ w' \end{pmatrix} = K'R' \begin{pmatrix} x \\ y \\ z \end{pmatrix}.$$

Eliminons les inconnues x, y et z de ces 6 équations ; les matrices K et R étant inversibles on en déduit :

$$\begin{pmatrix} u' \\ v' \\ w' \end{pmatrix} = K'R'R^{-1}K^{-1} \begin{pmatrix} u \\ v \\ w \end{pmatrix} = H \begin{pmatrix} u \\ v \\ w \end{pmatrix}.$$

Pour repasser des coordonnées projectives aux coordonnées euclidiennes il suffit de diviser par la troisième coordonnée w' soit $x' = \frac{u'}{w'}$ et $y' = \frac{v'}{w'}$. Nous n'utilisons pas le fait que $K = K'$ (même focale), donc en toute généralité, la matrice H a 9 coefficients. Notons que H peut être multipliée par un réel λ quelconque et on obtient encore les mêmes u' et v'. Donc on a 8 coefficient indépendants, et on peut supposer $h_{33} = 1$. On en déduit la relation homographique en développant la relation matricielle ci-dessus et en divisant par la troisième coordonnée : $x' = \frac{a_1x+a_2y+a_3}{a_7x+a_8y+1}$ et $y = \frac{a_4x+a_5y+a_6}{a_7x+a_8y+1}$, où $a_1 \dots a_8$ sont les coefficients de H. Ainsi avec les coordonnées d'un point vu dans les deux images, trouvez la relation du type :

$$A(x, y, x', y') \underbrace{\begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \\ a_8 \end{pmatrix}}_{h} = b_1(x, y, x', y')$$

Ce système est composé de 2 ou 3 équations linéaires (suivant votre manière de calculer), mais seulement 2 équations indépendantes.

En déduire qu'avec 4 points on a la relation

$$\underbrace{A}_{8\times 8} h = b$$

Ainsi on peut simplement en déduire les 8 coefficients de l'homographie. Une fois cette homographie connue on peut facilement en déduire les coordonnées théoriques dans l'image 1 et les comparer aux coordonnées réelles pour d'autres correspondances que celles utilisées pour calculer la relation homographique :

$$\begin{cases} x'_{theory} &= \frac{a_1x+a_2y+a_3}{a_7x+a_8y+1} \\ y'_{theory} &= \frac{a_4x+a_5y+a_6}{a_7x+a_8y+1} \end{cases}$$

2.3 Calcul de l'homographie par RANSAC

La première étape consiste à extraire des points d'intéret dans les images et à les mettre en correspondance entre 2 images. Etant donné un point d'intéret dans une image on cherche tous les descripteurs des points d'intéret dans l'autre image et on conserve le plus proche ; s'il est suffisemment proche, on le considère comme une correspondance. Le programme extrayant les points d'intéret, les descripteurs et la mise en correspondance vous est fourni avec un fichier *test.cpp* pour avoir un exemple d'utilisation.

Dans cet ensemble E de correspondances on a des correspondances effectivement correctes *inliers* et des correspondances fausses dites *outliers*. On cherche a déterminer les inliers et les outliers grâce à un algorithme, le RANSAC, que vous allez implémenter.

Le RANSAC (RANdom SAmple Consensus) consiste à itérer un grand nombre de fois N des hypothèses d'inliers et à les vérifier. A chaque itération on prend s correspondances au hasard nécessaires à l'estimation du modèle, parmi l'ensemble de correspondances E. Ici $s = 4$ et le modèle est l'homographie h. Ensuite on teste le nombre de points en adéquation avec le modèle, soit le nombre d'inlier n_i. Ici un point est un inlier si l'erreur de reprojection est sous un seuil s (en général $s = 1$ pixel) :

$$\begin{cases} x_{1calcul} = \frac{a_1 x_2 + a_2 y_2 + a_3}{a_7 x_2 + a_8 y_2 + 1} \\ y_{1calcul} = \frac{a_4 x_2 + a_5 y_2 + a_6}{a_7 x_2 + a_8 y_2 + 1} \\ \text{si } (x_1 - x_{1calcul})^2 + (y_1 - y_{1calcul})^2 < s^2 \text{ alors c'est un inlier.} \end{cases}$$

A la fin de chaque itération on teste si on a plus d'inliers que pour les itérations précédentes si c'est le cas on enregistre et met à jour le plus grand nombre d'inlier et la meilleure homographie, n_{ibest} et h_{best} ainsi que un tableau d'indice des inliers $Ind_{inliers}$. Au bout des N itérations on a le modèle h_{best} qui a permis de retrouver le plus grand nombre d'inliers ainsi que les indices des inliers $Ind_{inliers}$.

Ce modèle h_{best} a été obtenu à partir de seulement 4 correspondances et n'est pas ajusté au mieux à l'ensemble des inliers. A partir de là on estime la meilleure solution avec l'*ensemble* des inliers. Notre équation vue pour estimer le modèle à partir de 8 points devient :

$$\underbrace{A}_{(2*ninliers)\times 8} \quad h = \quad \underbrace{b}_{(2*ninliers)}$$

(plus d'équations que d'inconnues). La solution approchée est :

$$\underbrace{A}_{(2*ninliers)\times 8} \quad h_{final} = A.\text{PseudoInverse} \quad \underbrace{b}_{(2*ninliers)}$$

Pour estimer le modèle final avec les $ninliers$ points il est préférable de donner une égale importance à chaque ligne de l'équation $Ah = b$. Pour cela on peut simplement diviser chaque ligne de A par sa norme et diviser les coefficients de b en adéquation. Il est en fait préférable d'utiliser une normalisation plus adaptée au problème qui consiste à travailler sur des points normalisés tels que les inliers aient pour moyenne 0 et une déviation standard de 1. On à alors les équations matricielles utilisant les coordonnées projectives (u, v, w) (on note avec un $\hat{}$ les données dans l'espace normalisé) :

$$\begin{pmatrix} \hat{u}_{inlier1} \\ \hat{v}_{inlier1} \\ \hat{w}_{inlier1} \end{pmatrix} = \underbrace{Norm_1}_{3\times 3} \begin{pmatrix} u_{inlier1} \\ v_{inlier1} \\ w_{inlier1} \end{pmatrix}$$

$$\begin{pmatrix} \hat{u}_{inlier2} \\ \hat{v}_{inlier2} \\ \hat{w}_{inlier2} \end{pmatrix} = \underbrace{Norm_2}_{3\times 3} \begin{pmatrix} u_{inlier2} \\ v_{inlier2} \\ w_{inlier2} \end{pmatrix}$$

$$\begin{pmatrix} \hat{u}_{inlier1} \\ \hat{v}_{inlier1} \\ \hat{w}_{inlier1} \end{pmatrix} = \hat{H} \begin{pmatrix} \hat{u}_{inlier2} \\ \hat{v}_{inlier2} \\ \hat{w}_{inlier2} \end{pmatrix}$$

En déduire la fonction de dénormalisation permettant de passer de $\hat{H}$ à H, en fonction de $Norm_1$ et $Norm_2$.

2.4 Démarche générale et Implémentation

On calculera les homograhies de proche en proche entre 2 images consécutives. Puis on calculera les composition d'homographies qui permettent de passer d'une image à l'image centrale de référence. Pour chaque image on calculera la *bouding box* en fonction des coordonnées des 4 sommets d'une image et de leur transformations homographiques dans l'image de référence. Ainsi avec toutes les images on obtiendra la taille minimale de la fenêtre qu'il nous faut.

Du point de vue de l'implémentation, un programme de test utilisant les librairies nécessaires et leur utilisation vous est fourni. Principalement : calcul matriciel, extraction de points d'intéret, mise en correspondance. On utilisera le module LinAlg d'Imagine++ pour les matrices et la résolution de systèmes linéaires.

Remarque : la bonne façon de calculer l'image transformée par une homographie n'est pas de *pousser* les pixels originaux vers l'image résultat, mais au contraire de partir des pixels de l'image finale et de voir d'où ils viennent dans l'image initiale (donc de se servir de l'inverse de H).

Avant même de programmer le RANSAC, faites le test avec deux images seulement et des correspondances cliquées par l'utilisateur.

Vous pouvez commencer par utiliser les images fournies avec le projet puis chercher sur le web d'autres images permettant de reconstituer des panoramas.

2.5 Travail supplémentaire : recollage d'images

Lorsque vous recollez les images, vous pouvez voir nettement les frontières de recollement, car les images A et B ont des différences d'intensité lumineuse. Si il vous reste du temps, Il est donc plus élégant de faire une frontière progressive. Le problème est de recoller deux images A et B en une image C sans que cela soit flagrant. On part de deux méthodes.

L'idée la plus naïve consiste à construire $C(x) = (1 - \lambda)A(x) + \lambda B(x)$ avec λ qui passe progressivement de 0 à 1.

S'il vous reste d'avantage de temps vous pouvez essayer la méthode suivante qui est plus efficace : le principe du mélange multi-échelle est de faire varier λ de 0 à 1 sur une bande de largeur dépendant de la fréquence. Plus exactement, on décompose les images en fréquence. On mélange leur spectre sur une largeur différente pour chaque fréquence. Puis on revient à l'espace initial pour obtenir l'image finale. Cette méthode est décrite ici :
http://web.mit.edu/persci/people/adelson/pub_pdfs/spline83.pdf.

3 Stéréo, Carte de Disparité et Reconstruction 3D

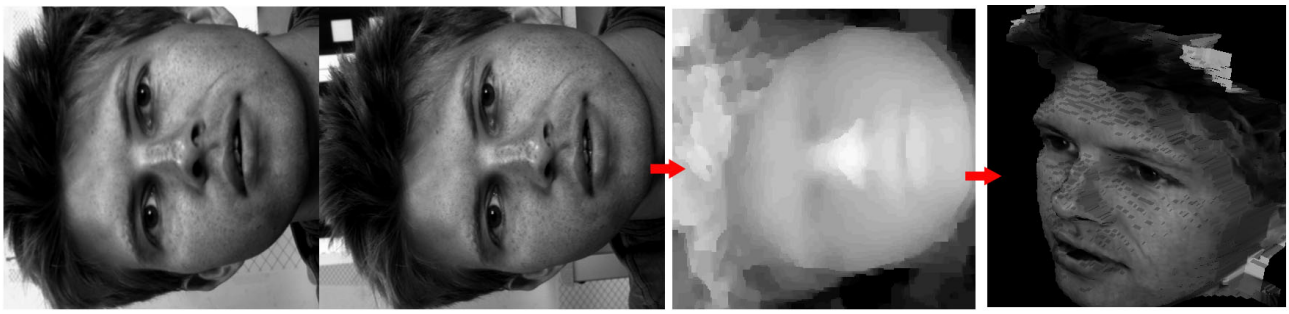


FIGURE 4 – Deux images. Carte de disparité. Reconstruction 3D

3.1 Introduction

Lorsqu'on dispose de deux images d'un même objet, on parle de paire stéréo et il existe des techniques pour retrouver la troisième dimension et reconstruire un modèle 3D de l'objet. L'objectif de ce projet est de réaliser un programme capable de reconstruire automatiquement un objet 3D (un visage par exemple) à partir d'une paire stéréo. Nous allons réaliser ce projet par étapes successives, en ordre inverse par rapport à la chaîne de traitement complète.

3.2 Mise en correspondance

Nous supposons données deux images I_1 et I_2 telles que les points qui se correspondent sont sur les mêmes horizontales (on peut toujours se ramener a ce cas si la lentille ne déforme pas trop les images). On parle d'images rectifiées. Il faut retrouver la disparité $d(u, v)$ telle que le point $m_1 = (u_1, v_1)$ de I_1 corresponde au point $m_2 = (u_2, v_2)$ de I_2 avec $u_2 = u_1 + d(u1, v1)$ et $v_2 = v_1$.

On essaie de trouver le correspondant dans la deuxième image par corrélation. Pour un pixel (x_1, y_1) de I_1, on estime la "ressemblance" du pixel (x_2, y_2) de I_2. Pour cela, on compare les valeurs des pixels autour de (x_1, y_1) et de (x_2, y_2). Plus précisément, on mesure la corrélation entre deux "fenêtres" autour de ces points. On définit par étapes la corrélation C_{12} :

– Moyenne :

$$\bar{I}_i(x_i, y_i) = \frac{1}{(2N+1)^2} \sum_{u=-N}^{N} \sum_{v=-N}^{N} I_i(x_i + u, y_i + v)$$

– Produit scalaire :

$$\langle I_i, I_j \rangle(x_i, y_i, x_j, y_j) = \frac{1}{(2N+1)^2} \sum_{u=-N}^{N} \sum_{v=-N}^{N} (I_i(x_i + u, y_i + v) - \bar{I}_i(x_i, y_i)) \\ (I_j(x_j + u, y_j + v) - \bar{I}_j(x_j, y_j))$$

– Norme :

$$|I_i|(x_i, y_i) = \sqrt{\langle I_i, I_i \rangle(x_i, y_i, x_i, y_i)}$$

– Corrélation-croisée normalisée :

$$C_{12}(x_1, y_1, x_2, y_2) = \frac{\langle I_1, I_2 \rangle(x_1, y_1, x_2, y_2)}{|I_1|(x_1, y_1)|I_2|(x_2, y_2)}$$

qui vérifie $-1 \leq C_{12} \leq 1$ (ce qui est un bon test de correction du programme !)

Pour un point (x_1, y_1), on pourrait choisir alors comme point correspondant le point (x_2, y_2) maximisant la corrélation $C_{12}(x_1, y_1, x_2, y_2)$. Dans notre cas la recherche se réduit le long d'une ligne et on cherche donc à maximiser $C_{12}(x_1, y, x_2, y)$ où y est fixé par le pixel de I_1.

Commencez par programmer les deux fonctions suivantes :

1. La corrélation $C_{12}(x_1, y, x_2, y)$
2. La correspondance qui pour un pixel p_1 dans I_1 renvoie son pixel correspondant p_2 ainsi que la valeur de corrélation.

Des images rectifiées sont fournies avec le projet.

3.3 Carte de disparités

L'objectif est à présent de calculer la disparité et la pofondeur pour tous les pixels de l'image. Calculer la disparité des pixels dans l'ordre ne permet pas d'obtenir des resultats convenables. Il faut d'abord commencer par les points pour lesquels la disparité est fiable et propager le calcul aux voisins en utilisant les disparités déjà connues.

On appelle graines les points qui ont le meilleur taux de corrélation et donc pour lesquels la valeur de la disparité sera d'autant plus fiable. Ces points sont ceux dont la corrélation dépasse un certain seuil que vous déterminerez par expérimentation.

On procède de la manière suivante :

1. Pour chaque graine $g_i = p_1^i$ dans I_1, on calcule le pixel p_2^i correspondant dans I_2, la disparité est notée $d_i = p_2^i - p_1^i$. On note $\mathcal{V}(g_i)$ l'ensemble des pixels voisins de g_i.
2. Pour chaque pixel $p_1^k \in \mathcal{V}(g_i)$, le correspondant p_2^k dans I_2 est tel que

   $$p_2^k(x) = p_2^i(x) + (p_1^k(x) - p_1^i(x)) + l$$

   où l est choisi pour maximiser la corrélation entre p_1^k et p_2^k, avec $|l| <= v$. En d'autre termes la variation de disparité est bornée : $|d_i - d_k| <= v$ où d_k est la disparité du pixel p_k (prendre $v = 1$ pixel).
3. Ajouter p_1^k dans la liste des graines (en queue !)
4. Itérer tant que la liste des graines n'est pas vide

Il est judicieux de traiter les graines dans l'ordre de leur taux de corrélation.

3.4 Visualisation 3D

La disparité est inversement proportionnelle à la profondeur ; représenter le visage fourni en 3D. Imagine++ vous permet d'entrer un modèle 3D et de laisser l'utilisateur tourner autour.

3.5 Votre visage en 3D !

Pour se ramener à la situation rectifiée (mouvement apparent des points horizontal), il faut déterminer un certain nombre de correspondances et transformer les images par une homographie adéquate. Ce processus est complexe et hors sujet. Pour néanmoins pouvoir traiter vos propres photos, vous pouvez utiliser l'URL suivante :

http://www.ipol.im/pub/demo/m_quasi_euclidean_epipolar_rectification/

Vous pouvez reconstruire votre visage en 3D en prenant deux photos devant un coin de mur bien texturé. Ne changez pas de zoom entre les deux photos, gardez les mêmes réglages, et bien sûr gardez la pose entre les photos. Changez le point de vue mais pas trop.

FIGURE 5 – 3D L-System

4 Synthèse d'images par L-systèmes

4.1 Introduction

Un L-System (ou système de Lindenmayer en bon français) est une grammaire formelle, permettant un procédé algorithmique, inventé en 1968 par le biologiste hongrois Aristid Lindenmayer qui consiste à modéliser le processus de développement et de prolifération de plantes ou de bactéries.

On pourra consulter l'entrée Wikipedia [2] pour une présentation succinte (page d'où est en fait tiré l'essentiel de cette présentation). La référence la plus complère est le livre "*The Algorithmic beauty of plants*" [3] de Przemyslaw Prusinkiewicz et Aristid Lindenmayer, 1990 où vous pourrez retrouver toutes ces informations et bien d'autres (les couleurs ne sont ici par exemple pas du tout traitées).

Plus précisément, un L-System est une grammaire formelle qui comprend :

- Un *alphabet* V. C'est l'ensemble des variables du L-System. V^* est l'ensemble des mots que l'on peut construire avec les symboles de V et V^+ l'ensemble des mots contenant au moins un symbole. (les notations V^* et V^+ sont standard en théorie des langages).
- Un ensemble de *constantes* S. Certains de ces symboles sont communs à tous les L-System (voir plus bas la Turtle interpretation).
- Un *axiome* ω choisi parmi V^+. Il s'agit de l'*état intial* choisi.
- Un ensemble de *règles* (on parle encore de *productions* en compilation), noté P, de réécriture de symboles de V.

Un L-System est alors noté (V, S, ω, P).

Illustrons ces définitions avec l'algue de Lindenmayer (un système simple permettant décrire le développement d'une algue) :

- Alphabet : $V = \{A, B\}$.
- Constantes : $S = \emptyset$.
- Axiome de départ : $\omega = A$.
- Règles : $\{(A \to AB), (B \to A)\}$.

La description informatique de ce système est donc :

```
Algue_de_Lindenmayer
{
Axiom A
A=AB
B=A
}
```

où `Algue_de_Lindenmayer` est le nom du L-System. En premier nous trouvons l'axiome ω, puis chaque règle de P (`A=AB` s'interprète comme tout symbole `A` devient un «*mot*» `AB` à la génération suivante. Voici le résultat sur six générations :

2. `http://fr.wikipedia.org/wiki/L-System`
3. `http://algorithmicbotany.org/papers/abop/abop.pdf`

- $n = 0$, `A`
- $n = 1$, `AB`
- $n = 2$, `ABA`
- $n = 3$, `ABAAB`
- $n = 4$, `ABAABABA`
- $n = 5$, `ABAABABAABAAB`
- $n = 6$, `ABAABABAABAABABAABABA`

Turtle interpretation

Cette chaîne de caractère est un mot sans signification en soi, mais qui peut fort bien se prêter à une interprétation graphique, en deux ou trois dimensions. Pour illustrer la manière de construire une plante avec un L-System, reprenons Wikipedia : *imaginons que nous avons un crayon à la main et qu'elle se balade sur la feuille sous nos ordres : "monte d'un cran, puis tourne de 20° à gauche, déplace toi deux fois de un cran, mémorise ta position et avance encore d'un cran, lève-toi puis repose-toi sur la position mémorisée" et ainsi de suite... Il a donc fallu inventer des symboles variants* $\in V$*, ou constants* $\in S$*, pour permettre de guider cette main. Plusieurs d'entre eux ont été normalisés, ils font partie de ce qu'on appelle la "Turtle interpretation". Ce nom vient de la "tortue" du langage de programmation Logo qui fonctionne sur le même principe. En fait c'est cette tortue qui est votre main qui tient le crayon.*

Voici les signes couramment utilisés :
- `'F'` : Se déplacer d'un pas unitaire ($\in V$).
- `'+'` : Tourner à gauche d'angle α ($\in S$).
- `'-'` : Tourner à droite d'un angle α ($\in S$).
- `'&'` : Pivoter vers le bas d'un angle α ($\in S$).
- `'^'` : Pivoter vers le haut d'un angle α ($\in S$).
- `'<'` : Roulez vers la gauche d'un angle α ($\in S$).
- `'>'` : Roulez vers la droite d'un angle α ($\in S$).
- `'|'` : Tourner sur soi-même de $180°$ ($\in S$).
- `'['` : Sauvegarder la position courante ainsi que le contexte graphique($\in S$)
- `']'` : Restaurer la dernière position sauvée ainsi que le contexte graphique ($\in S$).

Illustrons cela avec la courbe de Koch :
- Alphabet : $V = \{F\}$.
- Constantes : $S = \{+, -\}$.
- Axiome de départ : $\omega = F$.
- Règles : $\{(F \to F + F - F - F + F)\}$.

Une description informatique de ce système est donc :

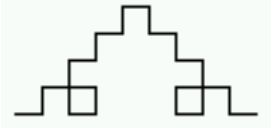
```
La_Courbe_de_Koch
{
angle 90
Axiom F
F=F+F-F-F+F
}
```

où `angle` détermine que l'on tourne de $90°$ avec les symboles + et −.

Voici le résultat sur deux générations :
- $n = 0$, `F`

- $n = 1$, `F+F-F-F+F`

- $n = 2$, `F+F-F-F+F+F+F-F-F+F-F+F-F-F+F-F+F-F-F+F+F+F-F-F+F`



SOL-System

Le système qui nous avons évoqué est déterministe, *i.e.*, il n'offre qu'une seule évolution possible depuis l'axiome à la génération n. Un SOL-System fait appel aux probabilités, il est aussi appelé système non-déterministe. Il est alors possible de déterminer plusieurs transformations pour un symbole. Chaque possibilité sera pondérée pour pouvoir donner priorité à certaines transformations par rapport à d'autres.

Par exemple, pour une plante stochastique :

```
Une_Plante_Stochastique
{
angle 20
axiom X
X=(0.2)F[++X]F[-X]+X
X=(0.8)F[+X]F[-X]+X
F=(1.0)FF
}
```

où `(0.2)`, `(0.8)` et `(1.0)` représentent les poids de chaque transformation possible de `X` et de `F`.

Voici un résultat possible sur deux générations :

- $n = 0$, `X`
- $n = 1$, `F[+X]F[-X]+X`
- $n = 2$, `F[+F[++X]F[-X]+X]F[-F[++X]F[-X]+X]+F[++X]F[-X]+X`

COntext-sensitive

Les systèmes précédents ne peuvent pas simuler l'interaction de parties d'une plante car ils sont context-free, *i.e.*, chaque partie se développe indépendamment des autres parties. Un L-System context-sensitive résout ce problème en prenant en compte ce qui précéde ou succède à une partie, c'est-à-dire un symbole. Un tel système est appelé (k, l)-System, le contexte de gauche est un "mot" de longueur k et celui de droite un "mot" de longueur l. Pour expliquer la manière dont se lisent les regles voici deux exemples :

Un signal *acropète* :

- Variable : $V = \{A, B\}$.
- Constantes : $S = \{+, -, [,], <\}$.
- Axiome : $\omega = B[+A]A[-A]A[+A]A$
- Règles : $\{(B < A \rightarrow B)\}$.

La règle se comprend ainsi : si un symbole `A` est **précédé** d'un symbole `B`, alors ce `A` devient un `B` à la génération suivante.

Un signal *basipète* :

- Variable : $V = \{A, B\}$.
- Constantes : $S = \{+, -, [,], <\}$.
- Axiome : $\omega = A[+A]A[-A]A[+A]B$
- Règles : $\{(B > A \rightarrow B)\}$.

La règle se comprend ainsi : si un symbole `A` est **suivi** d'un symbole `B`, alors ce `A` devient un `B` à la génération suivante.

Extension à la 3D

Cette "*turtle interpretation*" peut être exploitée en trois dimensions grâce aux idées de Harold Abeson et Andrea diSessa dans leur ouvrage commun, "Turtle geometry : the computer as a medium for exploring mathematic". L'orientation est représentée par trois vecteurs $\vec{H}$, $\vec{L}$ et $\vec{U}$ avec $\vec{H} \times \vec{L} = \vec{U}$.

- $\vec{H}$ pour *turtle heading*. Il s'agit du regard de la tortue.
- $\vec{U}$ pour *up*. Il s'agit de la direction vers laquelle se dirige la tortue.
- $\vec{L}$ pour *left*. Il s'agit de la gauche de cette tortue.

La rotation de la tortue se note alors $[\vec{H}'\vec{L}'\vec{U}'] = [\vec{H}\vec{L}\vec{U}]R$, où R est une matrice 3×3. Les rotations d'un angle α autour des axes $\vec{U}$, $\vec{L}$ ou $\vec{H}$ sont représentées par les matrices :

$$RU(\alpha) = \begin{pmatrix} \cos(\alpha) & \sin(\alpha) & 0 \\ -\sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$RL(\alpha) = \begin{pmatrix} \cos(\alpha) & 0 & -\sin(\alpha) \\ 0 & 1 & 0 \\ \sin(\alpha) & 0 & \cos(\alpha) \end{pmatrix}$$

$$RH(\alpha) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) \\ 0 & \sin(\alpha) & \cos(\alpha) \end{pmatrix}$$

Les symboles prennent maintenant la signification suivante :

- '+' : Tourner à gauche d'un angle α, en utilisant la matrice de rotation $RU(\alpha)$.

– '−' : Tourner à droite d'un angle α, en utilisant la matrice de rotation $RU(-\alpha)$.
– '&' : Pivoter vers le bas d'un angle α, en utilisant la matrice de rotation $RL(\alpha)$.
– '^' : Pivoter vers le haut d'un angle α, en utilisant la matrice de rotation $RL(-\alpha)$.
– '\' : Rouler à droite d'un angle α, en utilisant la matrice de rotation $RH(\alpha)$.
– '/' : Rouler à gauche d'un angle α, en utilisant la matrice de rotation $RH(-\alpha)$.
– '|' : Tourner autour de lui-même de $180°$, en utilisant la matrice de rotation $RU(180°)$.

Travail demandé

Le travail minimum demandé est :
– Implémentation et rendu graphique des systèmes de Lindenmayer déterministes.
– Implémentation et rendu graphique des systèmes de Lindenmayer stochastiques.
– Implémentation et rendu graphique des systèmes de Lindenmayer context-sensitive.

Noter qu'il vous sera nécessaire de développer un module pour lire des descriptions informatiques de Lindenmayer.

Pour les plus avancés d'entre vous.
– Gestion des couleurs.
– Gestion de la 3D et rendu graphique 2D.
– Sauvegarde d'images.

5 Fast Marching

5.1 Introduction

Le *fast marching* est une méthode de calcul de chemins minimaux étant donné une carte de potentiel. Il généralise l'algorithme de *Dijkstra* (cf. http://fr.wikipedia.org/wiki/Algorithme_de_Dijkstra.) pour des métriques plus complexes que la distance euclidienne et distance ℓ_1. Cette technique est assez largement utilisée pour la segmentation d'images médicales, extraction de structure tubulaires (angiographie, extraction de routes, ...). On peut y trouver d'autres applications comme la planification de chemin en robotique.

En théorie des graphes, le problème est modélisé à chercher un chemin entre deux nœuds d'une graphe G(E, V), telle que la somme des poids d'arête que le chemin a parcouru est minimisée.

5.2 L'approche

Dans ce projet, nous demandons de trouver le plus court chemin sur une image à partir d'un point de départ et un point d'arrivé illustré à la fig. 5.2. Il est naturel de considérer l'image comme une graphe où les pixels sont des nœuds et les arêtes sont des liens entre les voisins.

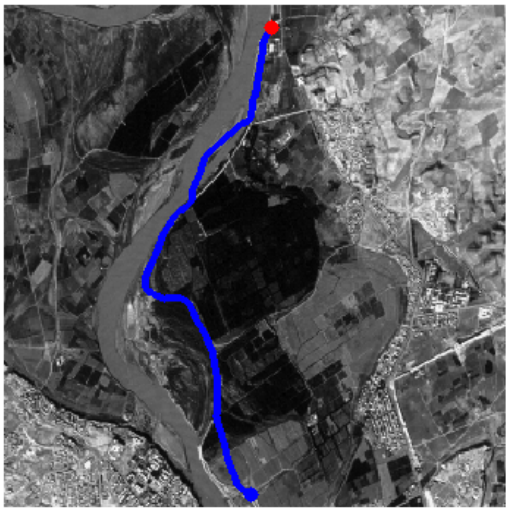


FIGURE 6 – Le plus court chemin sur une photo satellite.

En 2D, étant donné un métrique *W(x, y)*,un point de départ *(x0,y0)* et un point d'arrivé *(x1,y1)* le cout total d'une courbe *r(t)* à minimiser est :

$$\|\mathcal{L}(r)\| = \int_0^1 (W(r(t))||r'(t)||)dt \tag{1}$$

où *r(0)=(x0,y0)* et *r(1)=(x1,y1)*.

Le cout de déplacement vers (x, y) qu'on défini dans un premier temps est le suivant :

$$\|W(x,y)\| = \epsilon + |f(x0,y0) - f(x,y)| \tag{2}$$

où ϵ est une constant de lissage,f(x,y) est le niveau de gris du voisin avec lequel on construit le lien.

5.3 La carte de distance

Chercher la solution en itérant sur tous les chemins possible est peu efficace. En fait, la recherche du chemin peut se faire a l'aide de la carte de distance.

A partir du point de départ d, on calcule petit à petit le cout de chemin d'abord pour les voisins de d, puis pour les voisins des voisins, etc... Si un point p est déjà calculé mais on rencontre à nouveau ce point dans l'itération, on garde la valeur la plus petite. **Attention** : si p est mis à jour à nouveau, les voisins de p le doivent aussi (si nécessaire), et les voisins des voisins etc...(Conseil : on utilise la notion de points actifs). L'itération de calcul de la carte de distance se termine au moment où la frontière touche le point d'arrivée.

Le premier travail consiste donc à réaliser et illustrer la processus pour calculer la carte de distance à partir d'une image donnée 7.

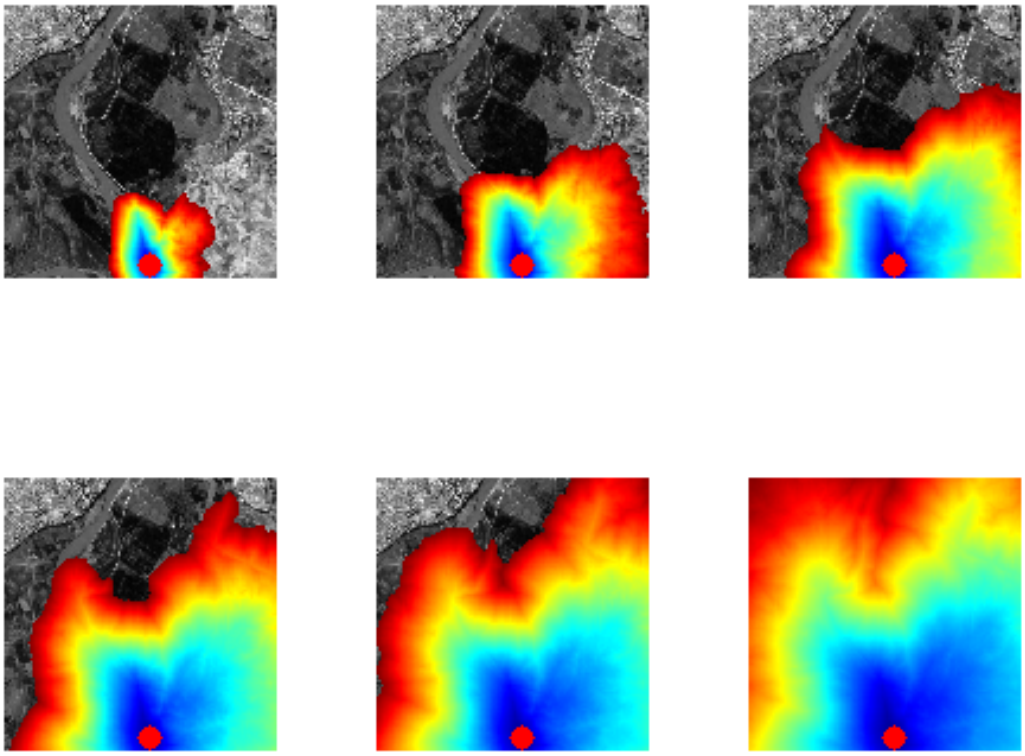


FIGURE 7 – calcule de la carte de distance

5.4 La descente de gradient

Une fois la carte de distance réalisée, il suffit de calculer les gradients des points sur la carte de distance, et effectuer une descente de gradient à partir du point d'arrivée jusqu'au point de départ.

Donc le deuxième travail est de

- calculer et montrer l'image des gradients.
- montrer le chemin construit sur l'image des gradients, la carte de distance et l'image initiale 8.

5.5 Variation de ϵ

Dans cette partie, vous devez essayer de varier la valeur ϵ pour voir les résultats diffèrer. Quand ϵ tend vers l'infini et quand ϵ tend vers 0, que deviendra la courbe ?

5.6 Variation de la fonction *W(x, y)*

Maintenant, vous devez aussi varier la fonction W pour voir la performance. Si W est quadratique ? Si au lieu de comparer avec (x0, y0), le cout de déplacement W(x,y) ne dépend que du contraste avec le voisin (xv,yv) ? Rapporter vos résultats.

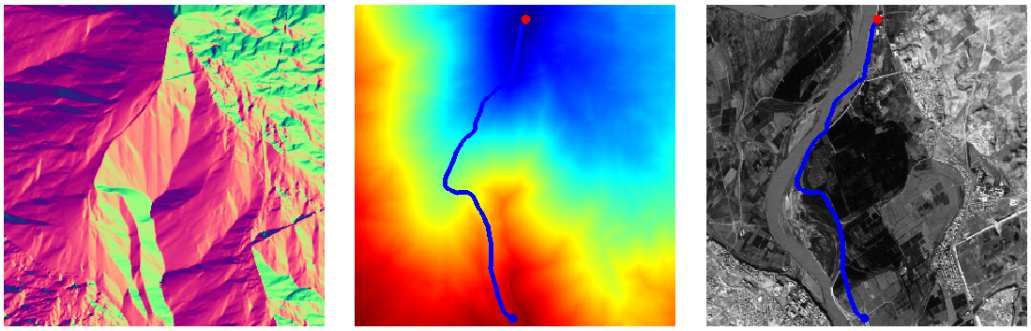


FIGURE 8 – l'image des gradients, la carte de distance et l'image initiale avec la solution

5.7 Tester sur d'autres images

Vous pouvez maintenant tester le programme sur d'autres images avec des paramètres appropriés. Par exemple, lancez le programme sur les images de labyrinthes et les images de carte de parc pour trouver les plus courts chemins.9

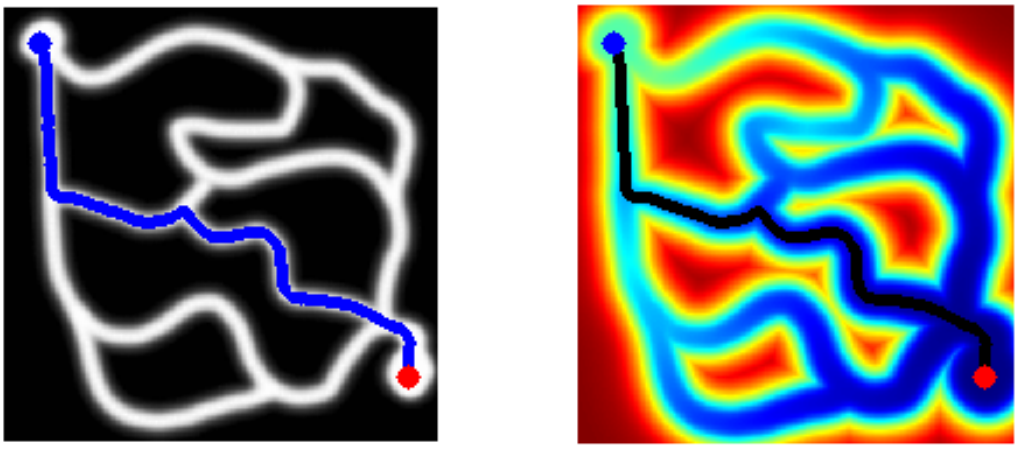


FIGURE 9 – l'image de labyrinthe et la carte de distance

5.8 Application à la détection de contours

Le plus court chemin peut aussi appliquer à la détection de contours, qui consiste à trouver le chemin passant par les points ayant un grand contraste. Ici, le poids peut être défini comme :

$$\mathcal{W}(x,y) = \frac{1}{\epsilon + G\sigma * G(x,y)} \tag{3}$$

où $G(x, y)$=| |∇f| |, * est une convolution

Il vous faut :

1. Programmer d'abord l'image de gradient, IG
2. Ensuite, lisser l'image IG par une gaussienne G de variance σ pour obtenir l'image I* qui représente $G\sigma$*$G(x,y)$
3. Programmer aussi l'image 10 qui représente $\frac{1}{\epsilon+G\sigma*G(x,y)}$.
4. Choisir deux points sur le contour, et programmer la carte de distance pour les deux.
5. Sélectionner les point à équidistance des deux points de départ.
6. Trouver deux points sur la ligne d'équidistance, qui ont les valeurs de distance les plus petites
7. Trouver les plus courts chemins entre les points pour compléter le contour
8. Faire la même chose sur d'autres images 11 pour tester.

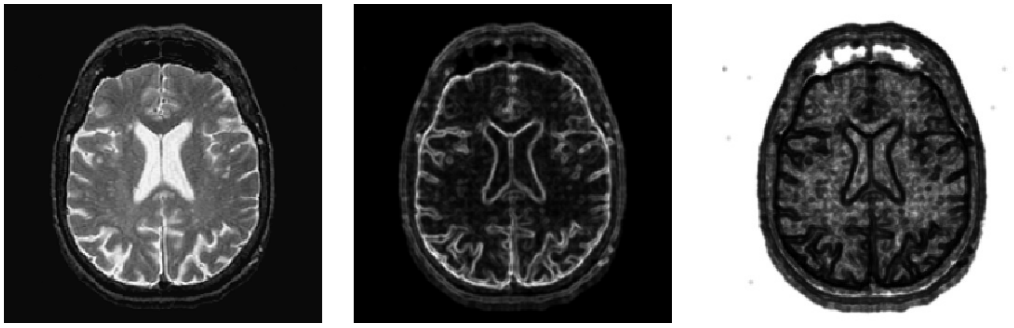


FIGURE 10 – l'image initiale, l'image $G\sigma * G(x,y)$ et l'image W(x,y)

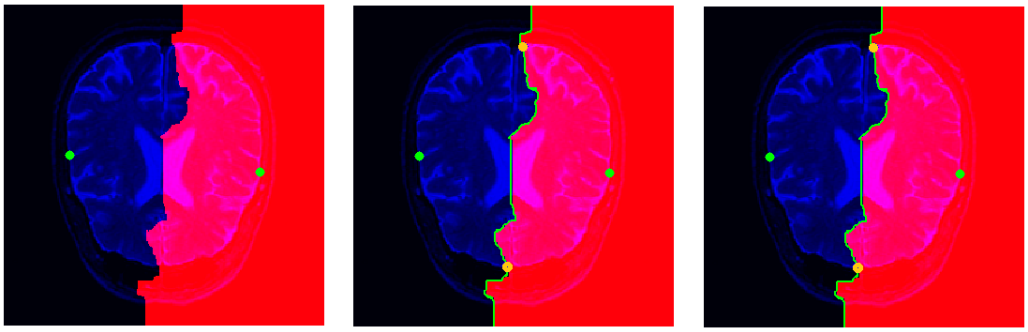


FIGURE 11 – les étapes de la segmentation

5.9 Segmentation d'image

En fait, on peut avoir un ensemble de points de départ au lieu d'avoir un seul, et encore plus, avoir plusieurs groupes d'ensembles de points. Donc on peut segmenter l'image en posant seulement quelques taches pour indique vaguement les parties différentes.

Vous devez dans cette partie,

- Proposer et implémenter cette méthode de segmentation.
- Expliquer vos algorithmes et les résultats
- Améliorer le système

5.10 Travail à fournir

1. Implémenter et vérifier la technique de la carte de distance.
2. Réaliser une interface graphique simple avec Imagine++ permettant de poser des points de départ et les point d'arrivée, de charger les fichier et d'illustrer les résultats. Il faut donc être capable de :
   - Charger une image.
   - Choisir l'application qu'on veut utiliser (chercher le plus court chemin, détection de contour ou segmentation)
   - modifier les paramètres s'il y en a.
   - afficher la solution et la carte de distance
   - Annuler ou recommencer une action (pour la pose de point)
   - Sauvegarder l'image de la carte de distance et la solution obtenue.
3. Analyser vaguement le temps d'exécution.

6 Space Invaders

6.1 Introduction

Sortie en 1978 sur bornes d'arcades, Space Invaders propose au joueur de piloter un vaisseau et de lutter contre l'invasion d'extraterrestres pixelisés. Il s'agit du premier succès pour le genre *shoot-them-up* (tuez les tous), c'est-à-dire qu'il propose au joueur de pratiquer l'éradication de tout ce qui se trouve à l'écran. Vous pouvez tester une version en ligne de ce jeu à cette adresse : http://www.freespaceinvaders.org/

Le but de ce projet est de réaliser une version de ce jeu utilisant Imagine++.

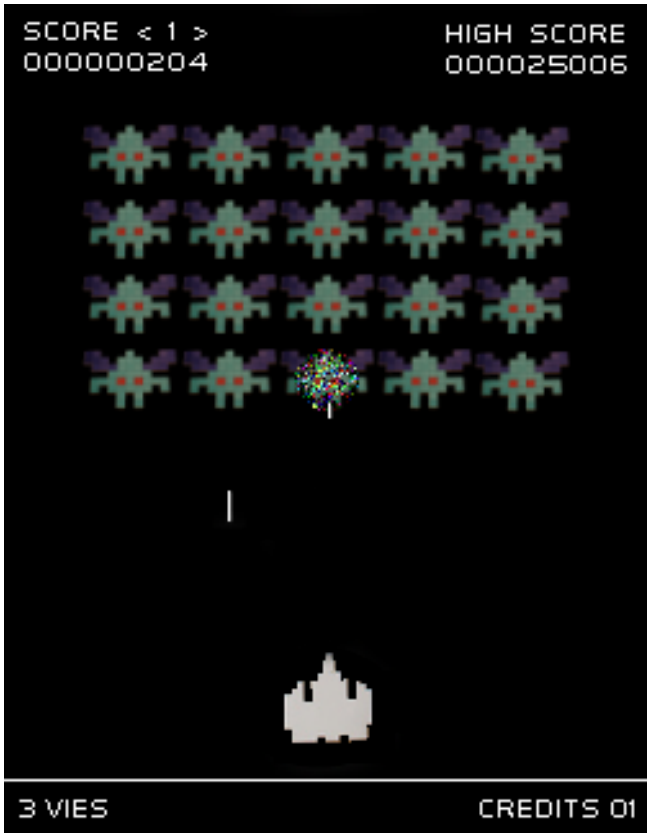


FIGURE 12 – Space Invaders dans sa version la plus simple.

6.2 La règle du jeu

Dans la version la plus simple du jeu, le joueur contrôle un vaisseau qui peut uniquement se déplacer horizontalement, vers la gauche ou vers la droite (en appuyant sur les flèches) et qui est cantonné au bas de l'écran. Ce vaisseau peut également tirer des coups de canon laser (par exemple en utilisant la touche espace), lequel canon peut tirer un nombre infini de fois.

Dans la partie supérieure de l'écran se trouvent les envahisseurs. Ils sont en un nombre déterminé : dans l'exemple de la figure 12 il y en a 20, répartis en quatre lignes de cinq individus. Les envahisseurs se déplacent non seulement de gauche à droite et de droite à gauche, mais ils peuvent également descendre. Leur déplacement est toujours le même : ils se déplacent cran par cran vers le bord droit de l'écran, lorsque la colonne la plus à droite butte contre le bord de l'écran, ils descendent tous d'un cran, puis se dirigent vers la gauche cran par cran. Lorsque la colonne de gauche butte contre le bord de l'écran, ils descendent de nouveau d'un cran et recommencent leur déplacement. Si l'un des envahisseurs touche le vaisseau, l'un et l'autre son détruits. La destruction du vaisseau ou d'un ennemi donne lieu à l'apparition d'un effet d'explosion.

Lorsqu'un tir du vaisseau atteint l'un des envahisseurs, celui-ci est détruit et le joueur marque un certain nombre de points, par exemple cinq. Régulièrement, les ennemis accélèrent. Lorsque tous les ennemis ont été détruits, le joueur passe au niveau suivant. Les envahisseurs tirent également, à des intervalles de temps aléatoires. Si le vaisseau du joueur est atteint par un tir, il est détruit.

Lorsque l'on passe au niveau suivant, les envahisseurs commencent plus proche du vaisseau du joueur et se déplace plus rapidement qu'au début du niveau précédent, jusqu'à une certaine limite qu'ils ne peuvent dépasser au début du jeu. Le joueur dispose de trois essais (trois vies) pour réaliser le score le plus important possible. À chaque passage de niveau, le joueur gagne une vie supplémentaire, mais il ne peut pas en avoir plus de cinq. Une fois toutes vies épuisées, le score du joueur est comparé aux dix meilleurs scores réalisés. Si le joueur a réalisé un score plus important qu'au moins l'un des autres scores déjà réalisés, son score et son nom sont alors sauvegardés.

6.3 Objectifs

Pour réaliser ce projet, il vous faudra déterminer la structure de données la plus appropriée pour représenter un nombre arbitraire d'envahisseurs, déterminer comment leur faire réaliser leur déplacement à une vitesse donnée et les faire tirer à intervalles aléatoires, avec une augmentation de la cadence à mesure que l'on monte dans les niveaux.

Il vous faudra également gérer les évènements au clavier de manière efficace afin de permettre au joueur

de déplacer le vaisseau. Il vous faudra aussi trouver un algorithme efficace pour déterminer si le vaisseau ou un envahisseur est touché par un tir. Enfin, il vous faudra gérer le score du joueur, représenter un classement des meilleures performances et le stocker de manière persistante, c'est-à-dire de telle sorte qu'on le retrouve identique après avoir relancé le jeu.

6.4 Améliorations du jeu

Rapidement, de nouvelles versions de Space Invaders sont apparues, qui enrichissaient le jeu et relançaient son intérêt. Vous êtes vous aussi encouragés à ajouter des fonctionnalités au jeu.

La première variante apparue consistait à ajouter des plots entre le vaisseau et les envahisseurs, réalisant une barrière partielle. Lorsqu'un plot subit un tir, il est partiellement détruit (plusieurs tirs sur le même plot sont nécessaires pour le faire totalement disparaître). Si un envahisseur touche un plot, il explose, endommageant au passage le plot. Ces plots permettent au vaisseau de s'abriter des tirs, mais il s'agit aussi d'une gêne pour atteindre les envahisseurs. Si les envahisseurs commencent tellement bas qu'ils devraient être dans les plots, les plots n'apparaissent pas. Également, on peut créer différent types d'envahisseur, le nombre de points gagné par le joueur devenant fonction du type d'ennemi détruit.

Pour ajouter un peu d'attractivité, les envahisseurs peuvent être animés, bougeant leurs ailes, tandis que le vaisseau peut s'incliner en fonction de la direction qu'il prend.

Également, certaines variantes faisaient apparaître aléatoirement des soucoupes volantes au sommet de l'écran. Si le joueur parvenait à les toucher d'un tir de laser, il avait alors une bonification de points. De plus, il est possible de limiter le nombre total de tirs disponible au joueur. Pour pouvoir en gagner plus, il lui faut alors toucher certaines des soucoupes volantes bonus – on notera au passage qu'il devient alors envisageable que le joueur ne soit tout simplement plus en mesure de tirer, on veillera donc à ce que le nombre de tirs disponible soit remis au maximum à chaque nouvelle vie.

D'autres variantes encore convertissaient le temps de jeu en points. N'hésitez pas à exprimer votre créativité en inventant vos propres options.

7 Wolfenstein 2D

7.1 Introduction

Wolfenstein 3D (Wolf 3D) est l'ancètre du jeu Doom. Lancé en 1992 par les futurs créateurs de Doom, Wolf 3D est le premier jeu de tir subjectif (vue à la première personne) de l'histoire. L'histoire se déroule lors de la seconde guerre mondiale, le héros est un soldat allié devant s'échapper d'un château tenu par les nazis.

Une vidéo du jeu est disponible au lien suivant.
http ://www.youtube.com/watch ?v=C00n4rDUMNo

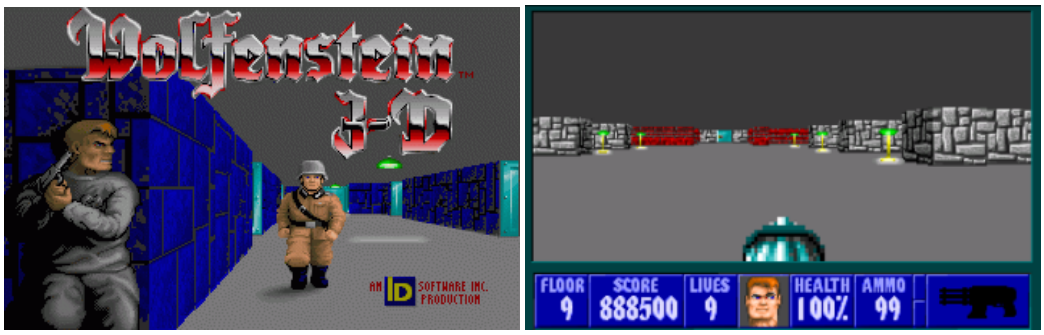


FIGURE 13 – Captures d'écran du jeu Wolfenstein 3D.

Dans chaque niveau, le but est de trouver la porte de sortie. Certains niveaux peuvent s'apparenter à des labyrinthes.

7.2 De la fausse 3D

Le jeu bien que s'apppelant Wolf 3D aurait mieux porté le nom de Wolf 2.5D. En effet, tout dans le jeu est géré de façon 2D.

Les cartes de niveaux sont des cartes planes. La figure 14 illustre que toute la carte est gérée en 2D. La position des personnages (héros comme ennemis) sont des points 2D sur cette carte.

De même dans la vue subjective, il n'y a pas de visée décorrelée de la position *ie* la position de l'arme est fixe sur l'écran, c'est uniquement l'orientation du personnage qui détermine la direction de l'arme.

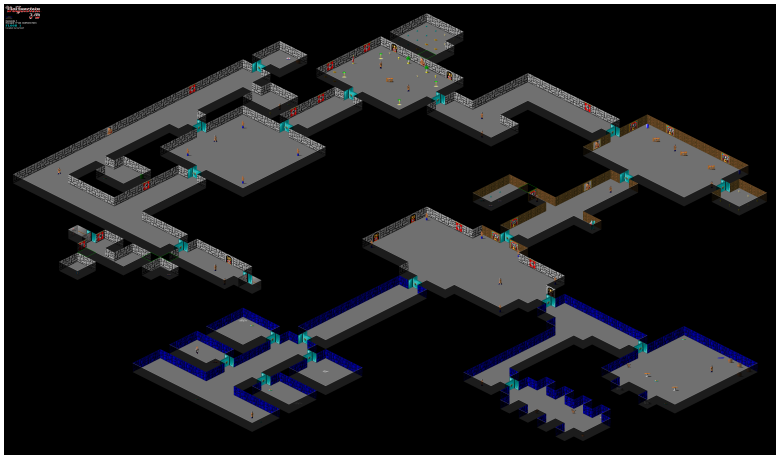


FIGURE 14 – Exemple de carte de niveau, la carte est en deux dimensions.

7.3 Les personnages

Le personnage principal se déplace de la manière suivante : il ne fait qu'avancer et reculer ou tourner sur lui-même. Les ennemis ont un parcours fixe dans le niveau (connu au chargement du niveau). Tous les personnages ont un niveau de vie variant entre 100% (pleine forme) et 0% (mort).

7.4 Implémentation

7.4.1 Format de données

Pour le format des niveaux, on pourra par exemple utiliser une image contenant les informations statiques (point d'entrée et de sortie du niveau, murs, portes ...) et un fichier annexe dans lequel on stockera le nombre d'ennemis, et leur parcours sous forme de suite de points ou de vecteurs.

Minimum

Le but est de concevoir un jeu 2D (vue de dessus) reprenant tout ou partie des concepts du jeu original Wolfenstein 3D. Le travail minimum demandé est repris par les points suivants :

- Plusieurs cartes (il faudra trouver un format simple de cartes).
- Des ennemis au parcours fixe (le parcours est lié à la carte), et à champ de vision circulaire.
- Entrée et sortie du niveau, chargement du niveau suivant.
- Gestion du niveau de vie et du nombre de vies restant.

Dans une première version, l'entrée du héros dans le camp de vision de l'ennemi tue le héros. Le rendu graphique devra montrer les zones de la carte vues par les ennemis (attention aux angles morts causés par les murs), les positions, des données vitales du héros, son orientation ...

Vers un jeu plus complet

Celui qui aura joué à Wolf 3D aura remarqué le jeu ne se limite pas à la version développée précédemment, voici en vrac quelques idées plus ou moins faciles à mettre en oeuvres pour rendre plus interessant le jeu.

- Champ de vision des ennemis conique
- Gestion des portes.
- Possibilité de tuer les ennemis, ne pas mourir instantanément dans le champ de vision de l'ennemi
- Les ennemis doivent se rapprocher du héros pour tirer
- Gestion des munitions
- Items (guérison, armes, munitions ...)
- Sons
- 3D ?
- autre ?

8 Autre

Vous pouvez proposer des sujets (à faire valider par le responsable du cours !) mais ne le faîtes que si vous vous sentez à l'aise en programmation et si vous êtes autonome.