Algorithmique et Programmation

—

Projets 2010/2011

—

G1: monasse*(at)*imagine.enpc.fr G2: david.ok*(at)*imagine.enpc.fr
G3: eric.bughin*(at)*gmail.com G4: renaud.marlet*(at)*enpc.fr
G5: drarenij*(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.

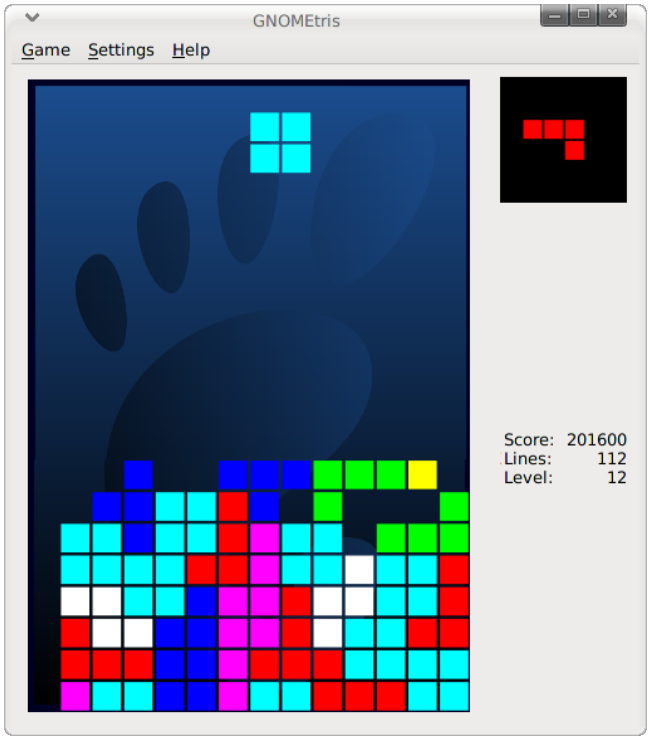


FIGURE 1 – Une interface de Tetris.

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°.

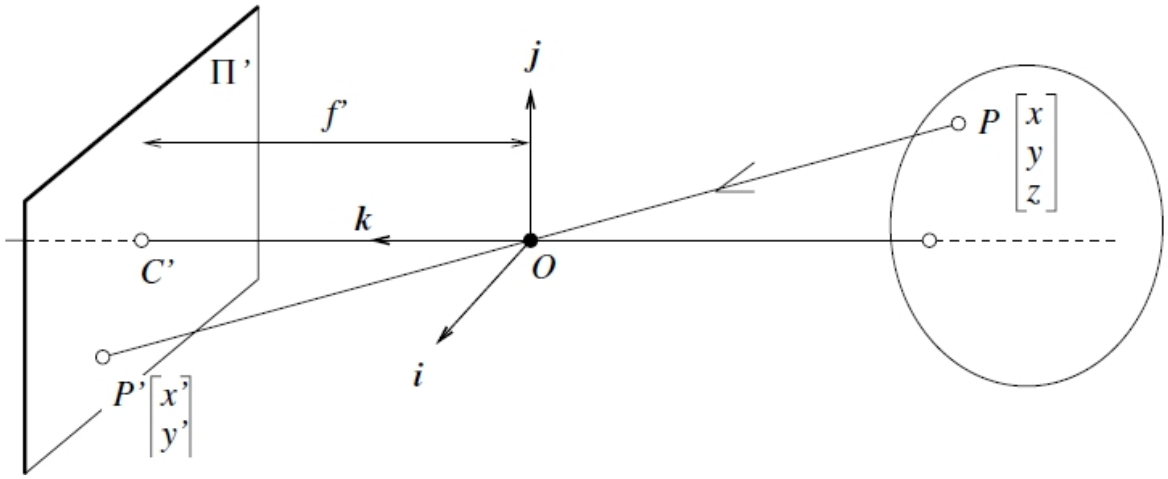


FIGURE 3 – Modéle de caméra pinhole

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 :

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

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 :

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

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'homographies. 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'intérets 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'intérets 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'intérets, 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 $treshold$ (en général $treshold = 1$ pixel) :

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

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 équatione vu 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'intérets, mise en correspondance. On utilisera la librairie 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

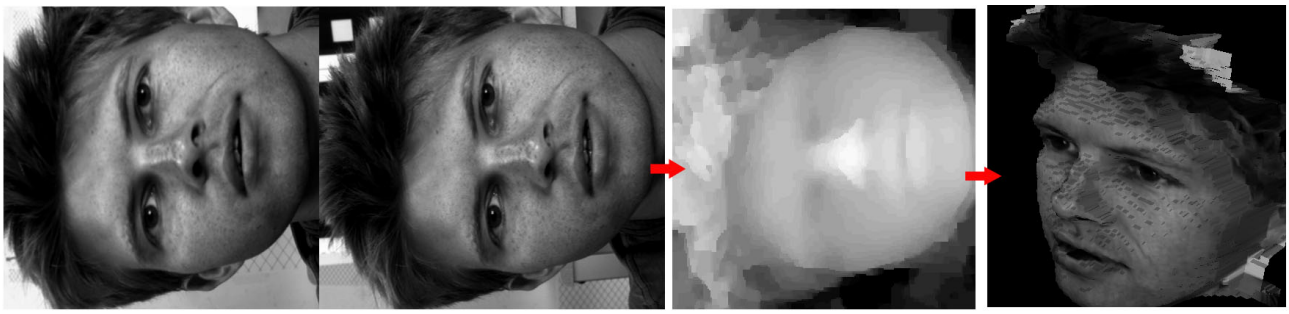


FIGURE 4 – Deux images. Carte de disparité. Reconstruction 3D

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.

4 Reconstruction 3D par Stéréophotométrie

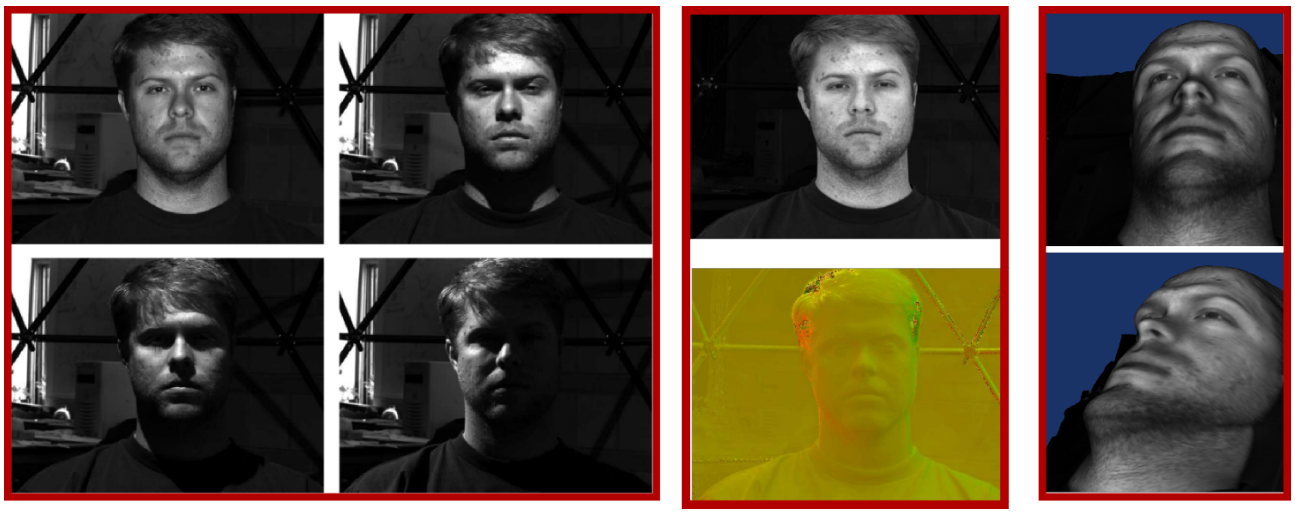


FIGURE 5 – Images avec éclairage variable. Carte de normales et albedo. Reconstruction 3D

4.1 Introduction

La stéréo photométrique (SP) est une technique de vision par ordinateur qui permet de reconstruire en 3D des objets fixes à l'aide de sources lumineuses contrôlées.

L'avantage de cette technique par rapport à la stéréo conventionnelle (voir REF), réside dans l'affranchissement du problème de correspondance. De plus, la SP fournit un modèle 3D dont la précision n'est limité que par la résolution de la caméra. En d'autres termes, quasiment tous les pixels de la caméra sont assurés d'être reconstruit en 3D.

Ceci n'est possible que sous certaines hypothèses qui définissent les limites et inconvénients de la SP dont les plus contraignants sont :

Surface lambertienne
L'applicabilité de la SF est limitée aux surfaces dites *lambertiennes* (platre, matériau mat), ou quasi-lambertienne (visage humain). Par opposition aux surfaces spéculaires (métal , miroir...).

Calibrage de la lumière
Pour produire un modèle 3D, il faut prendre en photo le sujet de reconstruction , éclairé à partir de sources de lumières différentes. La direction de ces sources de lumière doit être connue.

Portée de la reconstruction
Le volume de reconstruction est limité par la puissance de l'éclairage.

Les détails de chacune de ces contraintes seront étayés dans les prochaines sections.

4.2 Modèle de réflexion lambertien

Le modèle lambertien est un modèle de réflexion simplifié qui stipule que la lumière qui pénètre une surface est réfléchie de façon uniforme dans toutes les directions, indépendamment de la lumière incidente. En conséquence, la luminance de la surface I_i au point X_i reste la même peu importe de quel point de vue on lobserve et ne dépend que de sa normale, $\vec{N_i} = (p_i, q_i, 1)^T$, de la surface et de la direction d'éclairage $\vec{L} = (l_x, l_y, 1)^T$:

$$I_i = \rho(\vec{N_i} \cdot \vec{L}) \tag{1}$$

Ici, ρ represente la vraie couleur de la surface (ou le ton de gris) qu'on appelle *albedo* Le produit scalaire $(\vec{N_i} \cdot \vec{L})$ donne le cosinus entre la normale $\vec{N_i}$ et la direction d'éclairage L.

On souhaiterait donc, récupérer les normales et les albedo de la surface observée en supposant un modèle de réflexion lambertien.

4.3 Estimation des normales et albedo

Le but ici, est d'estimer pour chaque pixel observé i , son albedo ρ_i et les composantes de sa normale p_i, q_i. Cela fait 3 paramètres à estimer par pixel observé. L idée de base de la SP, consiste a prendre plusieurs photos

de l'objet à reconstruire en variant la direction de la source lumineuse sans changer la position de la caméra. Vu que chaque image apporte une contrainte sur la normale et l'albedo (eq.1), trois images suffisent pour estimer p_i, q_i et ρ_i à condition que les directions des trois lumières ne soient pas coplanaires.

Si nous disposons de p images à i pixels chacune, l'équation eq.1 se réécrit sous forme matricielle comme :

$$\underbrace{\begin{pmatrix} I_0^0 & I_0^1 & \cdots & I_0^p \\ I_1^0 & I_1^1 & \cdots & I_1^p \\ \vdots & \vdots & \ddots & \vdots \\ I_i^0 & I_i^1 & \cdots & I_i^p \end{pmatrix}}_{I_{i\times p}} = \underbrace{\begin{pmatrix} \rho_0 \\ \rho_1 \\ \vdots \\ \rho_i \end{pmatrix}}_{\rho_i}^T \underbrace{\begin{pmatrix} p_0 & q_0 & 1 \\ p_1 & q_1 & 1 \\ \vdots & \vdots & \vdots \\ p_i & q_i & 1 \end{pmatrix}}_{N_{i\times 3}} \cdot \underbrace{\begin{pmatrix} l_x^0 & l_x^1 & \cdots & l_x^p \\ l_y^0 & l_y^1 & \cdots & l_y^p \\ 1 & 1 & \cdots & 1 \end{pmatrix}}_{L_{3\times p}}$$

Si la matrices des intensités $I_{i\times p}$ et des lumière $L_{3\times p}$ sont données, la matrice des normales est estimée par simple inversion matricielle :

$$I_{i\times p} \cdot L_{3\times p}^{-1} = \rho_i N_{i\times 3}$$

Les albedos des pixels sont donnés par la norme des normales estimées.

4.4 Normales → Surface 3D

Une fois les normales estimées, il faudra les intégrer pour produire un modèle 3D.

4.5 Calibrage des lumières

Le moyen le plus simple pour estimer la direction d'une source lumineuse est d't'utiliser une boule chromée. La spécularité de cet objet fait que la lumière sera vue sous forme de spot sur la surface de la sphère. Si les propriétés géométrique de la sphère sont connues (rayon, emplacement,...), la position du spot peut être convertie en vecteur direction.

FIGURE 6 – Les sources de lumières apparaissent sur une sphère chromée sous forme de spots.

Pour les besoin de ce projet, les orientations des lumières vous seront données.

5 Synthèse d'images par L-systèmes

5.1 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 :

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

FIGURE 7 – 3D L-System

- 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 ńmotż `AB` à la génération suivante. Voici le résultat sur six générations :

- $n = 0$, `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 20ř 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
}
```

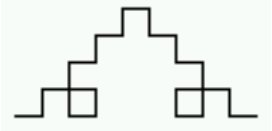
où `angle` détermine que l'on tourne de $90°$ avec les symboles `+` et `-`.

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.

6 Ray Tracing

FIGURE 8 – Ray tracing

6.1 Introduction

Le *ray tracing*, ou *lancer de rayons*, est une technique permettant de générer des images de synthèse. Cette méthode cherche à simuler le parcours inverse de la lumière de la scène vers l'oeil, et permet de gérer réflexions et réfractions de cette lumière sur les objets présents dans la scène.

6.2 Principe

Pour chaque pixel de l'image à générer, un rayon est lancé du point de vue vers la scène. Lors d'une intersection avec un objet de la scène, la trajectoire de ce rayon est réfléchie ou réfractée (suivant les propriétés de l'objet), le rayon continue sa course, et ainsi de suite jusqu'à rencontrer un objet non réfléchissant et non réfractant, ou jusqu'à un nombre défini de réflection/réfraction. Chacun de ces impacts apportera une contribution à la couleur apparente du pixel courant, suivant des coefficients de réflexion/réfraction.

Pour chacun de ces impacts, pour déterminer la luminosité en ces points, un rayon est lancé depuis chaque source lumineuse. Combinée à la couleur propre de l'objet, on peut ainsi déterminer la couleur finale en ce point. Un schéma est présenté en figure 9.

Pour le calcul de l'illumination d'un point, on se basera sur le *modèle d'illumination de Phong*, décrit ici : http://fr.wikipedia.org/wiki/Ombrage_Phong. On ne s'intéressera pas à la dernière partie, l'interpolation de Phong, non cohérente avec le lancer de rayon. Pour les différents modèles d'illumination existants, voir http://ph.ris.free.fr/these/, chapitre 2.

Pour la construction des objets 3D à visualiser, on se basera sur les *CSG*, ou *Constructive Solid Geometry*. Le principe est de construire des objets plus ou moins complexes en faisant des opérations simples sur des objets de bases : union, intersection, différence (pour de plus amples détails ainsi que des exemples en image, voir http://en.wikipedia.org/wiki/Constructive_solid_geometry).

6.3 A faire au minimum

1. Comprendre le principe du ray tracing (voir http://fr.wikipedia.org/wiki/Lancer_de_rayon en particulier) ;
2. Commencer par un programme simple ne gérant que des sphères affichées uniformément (pas de notion de source lumineuse, seulement le terme ambiant du modèle de Phong) ;
3. Rajouter la gestion d'une autre primitive géométrique : le plan orienté, définissant un demi espace ;

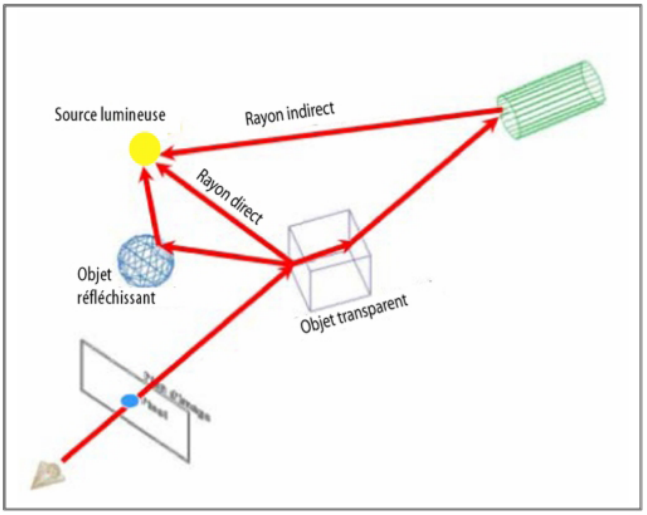


FIGURE 9 – Trajectoire d'un rayon

4. Utiliser les *CSG* pour construire de nouveaux objets. On pourra par exemple construire un cube de façon simple en utilisant des plans orientés ou encore tester des objets plus avancés (exemple : un dé peut-être construit par différences entre un cube et plusieurs sphères, etc.) ;
5. Rajouter la gestion des occlusions : un objet n'est éclairé par une source lumineuse que s'il n'existe aucun autre objet entre lui et cette source ;
6. Rajouter des sources lumineuses ponctuelles, sans gestion de réflection ni réfraction : chaque sphère aura un aspect mat (on ajoute le terme de diffusion du modèle de Phong). On se se préoccupera pas non plus des occlusions éventuelles.
7. Rajouter le terme spéculaire dans le modèle d'illumination.

6.4 Extensions possibles

1. Rajouter un affichage progressif : ne pas calculer les pixels ligne par ligne mais par grilles de plus en plus fines, en les affichant au fur et à mesure ;
2. Placer la description des scènes dans un fichier annexe : votre programme principal ne comportera alors que le "'moteur de rendu"' et se contentera d'interpréter le fichier de données ;
3. Rajouter la gestion des reflets.
4. Optimiser le calcul d'intersections avec une notion de boites englobantes ;
5. ...

6.5 Quelques liens en plus

– Support de cours sur le *ray tracing* :
http://www.cs.virginia.edu/~gfx/courses/2004/Intro.Fall.04/handouts/05-raycast.pdf ;
– Explications et exemples sur le modèle d'illumination de Phong (et d'autres) :
http://artis.imag.fr/~Nicolas.Holzschuch/cours/CIV01/CIV01_materials_small.pdf,
http://www.cs.virginia.edu/~gfx/courses/2004/Intro.Fall.04/handouts/06-light.pdf ;

7 Fast Marching

7.1 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 plannification de chemin en robotique.

Nous vous proposons d'implémenter cette méthode pour *la coloration interactive d'images en niveau de gris* et *la recoloration de parties d'images couleur*. L'idée est simple. L'utilisateur doit poser des graines de couleur sur les parties adéquates de l'image et l'algorithme du fast marching se charge alors de propager les couleurs tout en respectant les contours dans l'image comme illustré à la fig. 7.1.

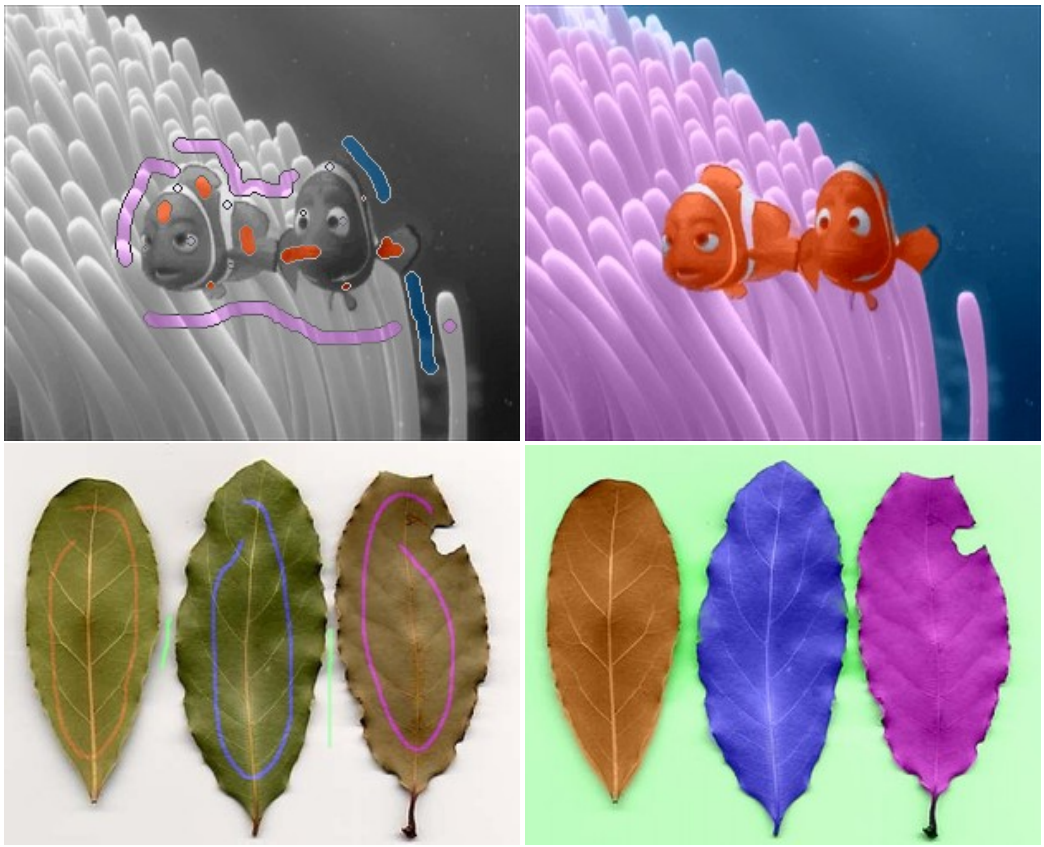


FIGURE 10 – Coloration de *Nemo* et recoloration d'images de *feuilles*.

7.2 Espaces couleurs

Pour colorer une image initialement en niveau de gris, on travaille dans l'espace couleur $YCbCr$. Or, une image couleur doit être sous format RGB pour être affiché correctement à l'écran. La conversion RGB-$YCbCr$ est donné par les relations suivantes :

$$\begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix} = \begin{bmatrix} 16 \\ 128 \\ 128 \end{bmatrix} + \begin{bmatrix} 0.257 & 0.504 & 0.098 \\ -0.148 & -0.291 & 0.439 \\ 0.439 & -0.368 & -0.071 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{2}$$

et

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1.164 & 0.000 & 1.596 \\ 1.164 & -0.392 & -0.813 \\ 1.164 & 2.017 & 0.000 \end{bmatrix} \begin{bmatrix} Y - 16 \\ Cb - 128 \\ Cr - 128 \end{bmatrix} \tag{3}$$

Attention, pour que les relations soient applicables, il faut s'assurer que
- les valeurs R,G et B ont des valeurs entre 0 à 255.
- Y a des valeurs entre 16 et 235. Cb et Cr ont des valeurs entre 16 et 240

Il faudra donc écréter si nécessaire les valeurs de chaque composante couleur pour une conversion correcte.

Votre image au départ est en niveau de gris. Ce niveau de gris sera la luminance Y. Pour colorer l'image, il faut connaître deux composantes manquantes, à savoir les chrominances Cb et Cr inconnues pour l'instant. Si on souhaite colorer une partie de l'image en jaune. Les chrominances Cb et Cr dans cette région seront celles du jaune, pour afficher toute cette partie en jaune. Remarquez, dans notre exemple, que le niveau de gris ne correspond pas forcément à la luminance réelle du jaune : c'est nécessaire pour préserver les structures géométriques dans les images.

7.3 Implémentation

Soient I une image en niveau de gris et $\mathcal{C}$ l'ensemble de couleurs différentes posées un partout dans l'image. La coloration des images se fait en deux étapes :

1. Dans un premier temps, l'objectif est de calculer une carte de distance par *fast marching* pour chaque couleur différente qu'on aura dénombrée.
2. Une fois cette étape faite, les chrominances finales en chaque pixel sont les moyennes des chrominances des couleurs dont les poids sont inversement proportionnels à la distance géodésique.

7.3.1 Calcul de la carte de distances pour une couleur donnée.

Fixons une couleur $c \in \mathcal{C}$. Soient $\mathcal{P}$ le support rectangulaire de l'image I et $\mathcal{P}_0$ le support des gribouillis dont la couleur est c. $\mathcal{P}_0$ est un sous-ensemble de $\mathcal{P}$. Dans le cas de *Nemo*, si $c :=$ magenta, alors

$$\mathcal{P}_0 := \{(x, y) \in I : I(x, y) := \text{magenta}\}. \tag{4}$$

Une carte de distance est en fait une image où la valeur de chaque point est égale à la distance à une certaine structure. *Mathématiquement*, une telle image vérifie l'équation *Eikonale* (eq. 5) pouvant être résolue par la technique du *fast marching*.

Le sous-ensemble $\mathcal{P}_0$ constitue le niveau zéro, autrement dit l'ensemble des points ayant une distance nulle. On considère aussi une carte de potentiel $\tilde{P}$ définie sur le plan représentant le « coût »d'un déplacement infinitésimal en un point. On appelle action minimale en un point p du plan l'énergie minimale intégré le long de tous les chemins reliant p à un élément p_0 de $\mathcal{P}_0$ [4] :

$$\|\mathcal{U}(p)\| = \inf_{\mathcal{A}_{p_0,p}} \int \tilde{P}(C(s))ds$$

où $\mathcal{A}_{p_0,p}$ est l'ensemble des chemins admissibles entre p et p_0. C est donc un élément de $\mathcal{A}_{p_0,p}$ paramétré par son abscisse curviligne. Dans le cas particulier où la carte de potentiel est uniformément égale à 1, $\mathcal{U}$ est la carte de distance (euclidienne) à la structure $\mathcal{P}_0$.

Cette carte d'action minimale satisfait l'équation *Eikonale* :

$$\|\nabla \mathcal{U}\| = \tilde{P} \qquad \text{et} \qquad \mathcal{U}(p_0) = 0\ ,\ \forall p_0 \in \mathcal{P}_0. \tag{5}$$

En pratique, une telle équation se traduit par le schéma numérique suivant :

$$(\max\{u - \mathcal{U}_{i-1,j}, u - \mathcal{U}_{i+1,j}, 0\})^2 + (\max\{u - \mathcal{U}_{i,j-1}, u - \mathcal{U}_{i,j+1}, 0\})^2 = \tilde{P}^2_{i,j}, \tag{6}$$

où la solution u est la valeur estimée de l'action minimal au point (i, j) : $\mathcal{U}_{i,j}$. On prendra $\tilde{P} = \|\nabla I\|_2$

Algorithme du *Fast Marching* en 2D

- Définitions :
  - points *Alive* : points où la valeur de l'action minimale $\mathcal{U}$ est déterminée et ne changera plus.
  - points *Trial* : prochains points de la grille à être examinés, ce sont les voisins immédiats des points *Alive*. Une valeur U de $\mathcal{U}$ a été calculée en ces points uniquement à l'aide des points *Alive* et de l'équation (6).
  - points *Far* : tous les autres points de l'image.
- Initialisation :
  - points *Alive* : l'ensemble de départ $\mathcal{P}_0$, on fixe $U(p_0) = \mathcal{U}$
  - points *Trial* : L'ensemble des voisins de $\mathcal{P}_0$ (en 4-connexité) avec pour valeur initiale $U(p) = \tilde{P}(p)$
  - points *Far* : tous les autres points de l'image avec $U(p) = \infty$
- Boucle :
  - On considère le point p_{min} de l'ensemble *Trial* avec l'action U la plus faible. Ce point est retiré de l'ensemble *Trial* et devient *Alive* : l'action minimale est déterminée en ce point et $\mathcal{U}(p_{min}) = U(p_{min})$.
  - Tous les voisins de p_{min} n'appartenant pas déjà à l'ensemble *Alive* sont ajoutés à l'ensemble des points *Trial*. L'action minimale U est estimée en ces points à l'aide des points *Alive* voisins.

L'implémentation d'un tel algorithme requiert une grille caractérisant l'état d'un point $\{$*Alive*, *Trial*, *Far*$\}$ et une liste de points *Trial*, devant toujours être triée dans l'ordre croissant de l'action minimale U estimée. Il faudra donc gérer un tri initial de cette liste (lors de la phase d'initialisation) et l'insertion rapide de nouveaux éléments dans cette liste ordonnée.

Pour cela, l'utilisation d'une file de priorité de la STL (cf. `http://www.cplusplus.com/reference/stl/`) sera nécessaire. La compréhension de l'extension 2 de l'exercice 6 à faire à la maison vous aidera probablement à mieux concevoir l'implémentation du *fast marching*.

4. Cela n'est pas sans rappeler l'optique géométrique

7.3.2 Calcul de chrominances par moyennage

Pour calculer les chrominances Cb et Cr pour chaque pixel (i, j) de l'image, on vous propose d'appliquer la formule suivante :

$$Cb(i,j) = \frac{\sum_{c\in\mathcal{C}} W(d_c(i,j))Cb(c)}{\sum_{c\in\mathcal{C}} W(d_c(i,j))} \quad (7)$$

$$Cr(i,j) = \frac{\sum_{c\in\mathcal{C}} W(d_c(i,j))Cr(c)}{\sum_{c\in\mathcal{C}} W(d_c(i,j))} \quad (8)$$

où d_c est la distance du pixel (i, j) par rapport au gribouillis de couleur c, distance calculée précédemment par fast marching, et

$$W(r) = r^{-b} \quad (9)$$

où b est une valeur strictement positive à expérimenter.

7.4 Travail à fournir

1. Implémenter et vérifier la technique de coloration.
2. Réaliser une interface graphique simple avec Imagine++ permettant de poser des graines de couleur. Comme dans *Photoshop*, il faut donc être capable de :
   – Charger une image.
   – Choisir la taille du pinceau et la couleur associée pour y peindre les graines et de les gommer.
   – Annuler ou recommencer une action (pour la pose de graine) pour la pose des graines (*undo*/*redo*). On pourra utiliser une pile.
   – Sauvegarder l'image ainsi colorée.
3. Réimplémenter le calcul de carte de distances avec l'algorithme de Dijkstra, beaucoup plus simple, et comparer visuellement.
4. Montrer avec des exemples qu'il est déjà possible de partitionner une image de manière interactive avec l'implémentation actuelle du fast marching [5]. Cette méthode permet alors également de segmenter des images en autant de parties possibles.
5. Il est facile de se limiter à re-colorer quelques parties d'une image en couleur. Comment faire ? Etoffer alors l'interface graphique.

Amélioration de la coloration d'images Lorsque les contours d'une image ne sont pas très nets ou dégradés, il est souvent nécessaire de poser beaucoup de graines un peu partout au niveau des contours pour s'assurer d'une bonne coloration d'images, ce qui devient très vite fastidieux pour l'utilisateur. Pour minimiser l'effort manuel, implémentez *le filtrage de Canny* (cf. `http://fr.wikipedia.org/wiki/Algorithme_de_Canny`) pour localiser les contours. Voir comment cette localisation permet de dissuader la méthode de colorer les images sans déborder au niveau des contours de manière stricte.

8 Jeu de Pacman

8.1 Introduction

Il s'agit ici de programmer un jeu de PacMan. Les règles du jeu sont les suivantes :
– Le terrain de jeu possède un passage secret reliant la gauche et la droite du labyrinthe (cf. figure 11).
– Un nombre donné de "vitamines" sont disposées au hasard (on pourra commencer par 4).
– Il y a un PacMan et au moins 4 fantômes.
– Quand le PacMan a mangé toutes les "gommes" et "vitamines", le niveau est terminé.
– Lorsque Le PacMan mange l'une des "vitamines", alors il peut manger les fantômes et les fantômes n'avancent qu'un coup sur deux pendant 35 tours de jeu.
– Au début de chaque niveau, les fantômes sont placés dans leur "repère". Ils y sont replacés après s'être faits mangés. Les fantômes ainsi que le Pacman peuvent sortir du "repère" mais ne peuvent pas y rentrer.
– Le Pacman possède 5 vies.
– Points marqués :
  – gomme : 10 points.
  – vitamine : 50 points.
  – fantôme : 200 points, puis 400 points, puis 800 points, puis 1600 points (au cours de la même prise de "vitamine").
  – tableau entier : 500 points.

5. Ce serait une extension 3 de l'exercice 6…

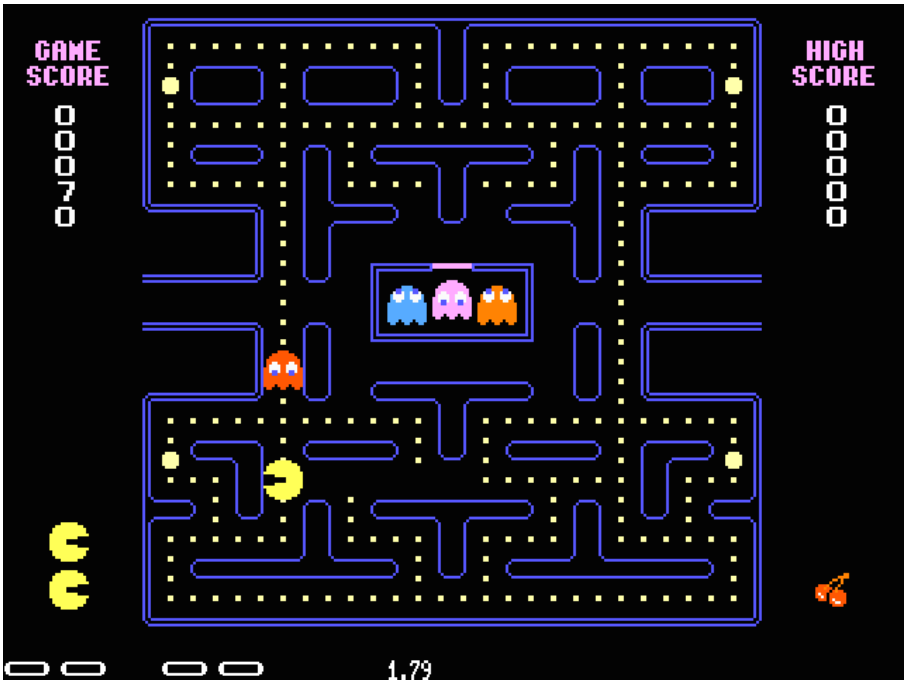


FIGURE 11 – Terrain de jeu.

8.2 Travail à fournir

On vous demande de programmer ici l'environnement de jeu. Le plateau de jeu, se lira à partir d'un fichier texte similaire à celui proposé en pièce jointe (libre à vous d'en créer d'autres sur le même modèle). On propose de représenter le plateau de jeu par une matrice de chiffres allant de 0 à 5 dont les valeurs ont la signification suivante :

- 0 : rien.
- 1 : gomme.
- 2 : vitamine.
- 3 : mur.
- 4 : sortie du repère des fantômes.
- 5 : repère des fantômes.

Pour gérer les déplacements et les affichages d'imagettes, on s'inspirera du TP 9, Tron. Des imagettes sont fournies, mais vous pouvez aussi bien en utiliser d'autres si vous préférez.

On vous demande de gérer le déplacement des fantomes de manière à ce que le jeu soit le plus dur possible. Enfin, on gèrera la sauvegarde et l'affichage des meilleurs scores.

9 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.