Images et géométrie

1 Introduction

1.1 Récupération des fichiers

Depuis la ligne de commande, naviguez vers un répertoire vous appartenant, et exécutez la commande suivante, qui téléchargera et décompressera un répetoire `geometrie` contenant quelques sources C et un makefile :

```
curl cdn.sci-phy.org/mp2i/tp2-geometrie.tgz | tar xvz
```

Vous y trouverez notamment un fichier source C `basic_ppm.c` contenant quelques fonctions destinées à créer, modifier et enregistrer une image, et son fichier d'en-tête `basic_ppm.h`, ainsi qu'un fichier `geometrie.c` que l'on va chercher à compléter durant cette séance et un makefile [1] qui permet de compiler automatiquement tous les fichers avec « `make all` » et de les exécuter (après les avoir compilés si nécessaire) avec « `make run` ».

Dans le fichier `geometrie.c` se trouvent de nombreuses fonctions « à compléter », généralement vides. La fonction `test_question` contient différents tests déjà écrits : il suffit de renseigner, dans l'appel se trouvant dans la fonction `main`, via son argument, le numéro de la question que l'on traite pour activer différents tests visant à s'assurer du bon fonctionnement des fonctions concernées par la question. Le résultat attendu est décrit dans la question correspondante.

Rien ne vous empêche bien évidemment de modifier ou compléter ces tests selon vos besoins, afin de vérifier le bon fonctionnement de ce que vous écrivez ou comprendre pourquoi une fonction ne donne pas le résultat attendu. La réalisation de tests est une part importante et incontournable du développement logiciel, ainsi d'ailleurs qu'un attendu du programme.

1.2 Description des fichiers fournis

Les fichiers `basic_ppm` fournissent les choses suivantes :

- un type `color` désignant une couleur, dont les valeurs possibles sont identifiées par les noms `black`, `red`, `green`, `yellow`, `blue`, `magenta`, `cyan`, `white` et `gray` ;
- une fonction `init(nblgn, nbcol, c)` prenant en argument un nombre de lignes, un nombre de colonnes et une couleur, et initialisant une image aux dimensions demandées et remplie uniformément par la couleur demandée, et retourne un entier nul si l'initialisation a pu être effectuée et non nul sinon (la valeur retournée est alors une indication du problème rencontré) ;
- une procédure `set_pixel_color(i, j, c)` prenant en argument des coordonnées `i` et `j` et une couleur `c`, et changeant le pixel dont les coordonnées sont spécifiées dans la couleur demandée ;
- une fonction `write(filename)` prenant en argument un nom de fichier et enregistrant l'image sur laquelle on travaille au format PPM sous le nom demandé, et retourne un entier nul si l'enregistrement s'est bien passé et non nul sinon ;
- deux fonctions `nbl()` et `nbc()` retournant les dimensions de l'image (respectivement le nombre de lignes et le nombre de colonnes) actuellement manipulée (ces fonctions retournent -1 si aucune image n'a été initialisée).

Quelques précisions tout d'abord concernant ces fonctions et procédures. On ne travaille toujours que sur une seule image à la fois, c'est la raison pour laquelle il n'y a pas, parmi les paramètres, de précision indiquant l'image concernée. Il est impératif d'initialiser une image avec `init` avant de faire quoi que ce soit.

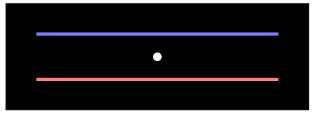
On utilise un système de coordonnées où i désigne le numéro de ligne (commençant à 0 en haut de l'image) et j le numéro de colonne (commençant à 0 à gauche de l'image). L'indexation se fait donc essentiellement comme pour une matrice, si l'on met à part l'indexation débutant à 0. Utiliser comme paramètre de `set_pixel_color` des coordonnées hors de l'image n'est pas permis (de même que de passer autre chose comme couleur que les couleurs proposées), mais le comportement du programme ne sera pas indéfini pour autant, la commande sera simplement ignorée, avec affichage d'un message d'erreur dans la console.

2 Premiers pas

2.1 Tracé de lignes horizontales et verticales et de rectangles

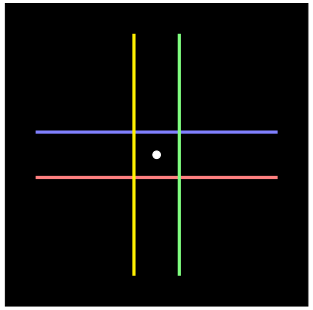
1. Compléter la fonction `hline(i, j1, j2, c)` pour qu'elle trace une ligne horizontale de la couleur `c` sur la ligne `i` entre les colonnes `j1` et `j2` incluses. On supposera que (i, j_1) et (i, j_2) sont des coordonnées valides, **mais il n'est fait aucune hypothèse supplémentaire** sur `j1` et `j2`.

Si la fonction `hline` est correctement écrite, l'image `resultat.ppm` produite par l'exécution du programme (grâce à l'appel à la fonction `test_question`) doit montrer deux lignes horizontales bleue et rouge exactement de même longueur et un pixel (un point) blanc entre les deux, comme illustré ci-dessous :

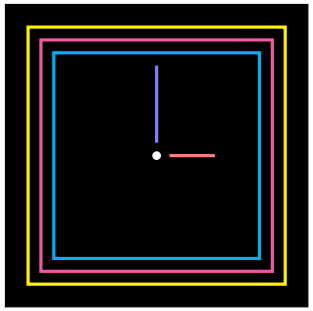


1. Le fichier `makefile` est fourni comme une commodité pour pouvoir se concentrer sur la programmation sans perdre de temps sur les aspects de compilation, mais il vous faudra impérativement savoir compiler un programme C sans aide lors des oraux des concours !

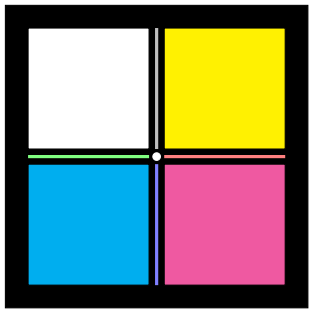
2. Faire de même pour la fonction `vline(i1, i2, j, c)` qui doit tracer une ligne verticale entre les points (i_1, j) et (i_2, j) (inclus), avec les mêmes règles que précédemment. Le test [2] ajoute deux lignes verticales verte et jaune de même hauteur au résultat précédent.



3. Développer la fonction `box(i1, j1, i2, j2, c)` traçant un rectangle (non rempli) entre les points (i_1, j_1), (i_1, j_2), (i_2, j_2) et (i_2, j_1). Aucune hypothèse n'est faite sur ces points si ce n'est qu'ils se trouvent dans l'image. Le test doit afficher trois carrés imbriqués et une horloge indiquant 3h, toujours avec un point central.



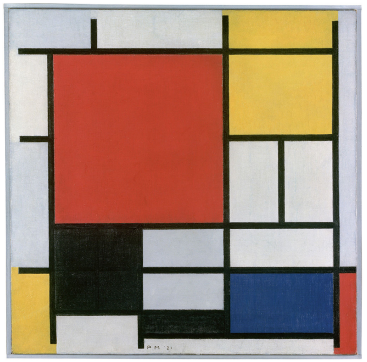
4. Faire de même pour la fonction `boxf(i1, j1, i2, j2, c)` qui trace un rectangle cette fois-ci *rempli* par la couleur demandé. Le test doit montrer quatre carrés pleins, quatre lignes et un point central, le tout encerclé par un contour carré.



2. On pensera à modifier l'argument de l'appel à la fonction `test_question` dans `main`!

2.2 Utilisation (facultatif)

Si vous le souhaitez, vous pouvez compléter la fonction `Mondrian` pour qu'elle réalise une oeuvre dans le style de « *Composition en rouge, jaune, bleu et noir* », par Piet Mondrian [3], illustrée ci-dessous. « `rand()%nbl()` » [4] et « `rand()%nbc()` » retournent respectivement un index de ligne et de colonne choisi aléatoirement, si vous souhaitez l'envie d'aller dans cette voie. La fonction est automatiquement appelée en utilisant le numéro de cette question dans l'appel à `test_question`.



3 Tracé de lignes obliques

3.1 Première approche

On souhaite à présent tracer des lignes « obliques », c'est-à-dire pas nécessairement horizontales ou verticales. On souhaite toutefois qu'elles soient aussi propres que possibles.

Dans un premier temps, on suppose que l'on veut tracer une droite entre les points (i_1, j_1) et (i_2, j_2) où les deux extrémités sont des points valides de l'image, vérifiant $i_1 \leqslant i_2$ et $j_1 \leqslant j2 \leqslant j1 + (i_2 - i_1)$. Il s'agit donc d'une droite se dirigeant dans une direction comprise entre le sud et le sud-est.

5. Écrire une première fonction `line_case1_basic(i1, j1, i2, j2, c)` déterminant, pour tout $i_1 \leqslant i \leqslant i_2$, l'entier j le plus proche possible de l'endroit où la droite géométrique doit se trouver, et utilise ce calcul pour tracer la droite souhaitée (il doit y avoir un unique pixel pour chaque $i_1 \leqslant i \leqslant i_2$). On tolérera l'utilisation de flottants ici. Il est vivement conseillé de raisonner sur un dessin, et de retrouver dans ses souvenirs le théorème de Thalès. Le test montre trois paires de points rouges, entre lesquels on doit voir apparaître autant de droites bleues.

3. Notons que « Piet » est un langage de programmation ésotérique (fr.wikipedia.org/wiki/Piet) créé par David Morgan-Mar, dont les codes sources consistent en des images similaires à celle illustrée ici, utilisant dix-huit couleurs en plus du noir et du blanc. On ne demande pas que l'œuvre produite soit un code source Piet qui effectue quelque chose d'intéressant, mais si vous avez beaucoup de temps à perdre...

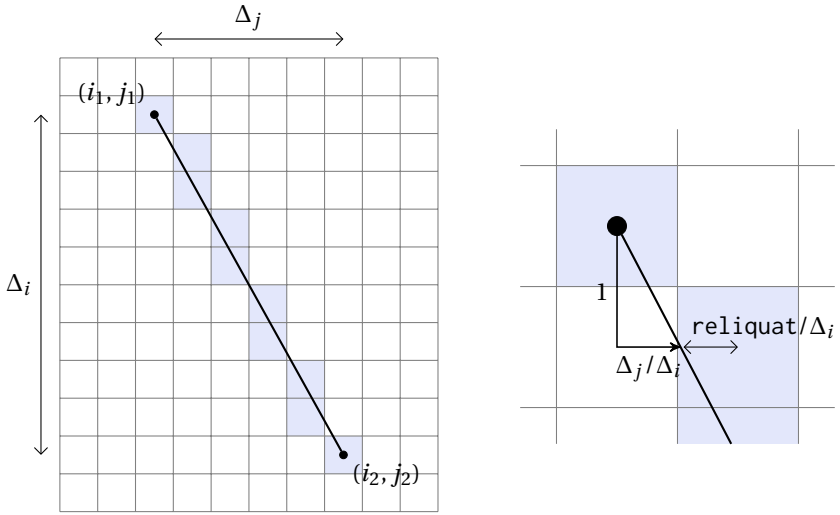
4. On ajoutera le header `<stdlib.h>` pour `rand()`.

3.2 Algorithme de Bresenham

L'ennui avec cette méthode est qu'elle n'est pas très efficace car elle effectue beaucoup de calculs. Nous allons essayer de l'améliorer. Si l'on regarde bien les droites produites, on voit que l'on passe d'un pixel au suivant sur la droite soit en descendant d'un pixel, soit en descendant d'un pixel et en se décalant d'un pixel vers la droite. Le tout est donc de savoir pour quelles lignes il faut se déplacer vers la droite lors de la descente !

Notons $\Delta_i = i_2 - i_1$ et $\Delta_j = j_2 - j_1$. Si l'on suit la droite *géométrique*, lorsque l'on se déplace de 1 unité vers le bas, on se déplace de Δ_j/Δ_i unités vers la droite. Seulement, dans l'image, on travaille avec des déplacements entiers.

Pour tracer la droite, l'algorithme de Bresenham fonctionne de la façon suivante : on initialise une variable entière `i` à `i1` et une variable entière `j` à `j1`, ainsi qu'une variable `reliquat`, initialisée à 0. Cette variable contiendra, à tout instant, Δ_i fois l'erreur (relative) entre la position courante (i, j) et le point réel (i, j') de la droite ayant la même ordonnée.



Compte tenu de l'expression Δ_j/Δ_i introduite précédemment, on comprend aisément que le reliquat correspond bien toujours à un entier.

Puis, tant que `i`$\leqslant$`i2` :

- on affiche un point aux coordonnées courantes (i, j) ;
- on incrémente `i` ;
- on ajoute à `reliquat` la quantité $\Delta_i \times \frac{\Delta_j}{\Delta_i} = \Delta_j$;
- si `reliquat`$>\left\lfloor \frac{\Delta_i}{2} \right\rfloor$, alors on incrémente `j` et on retire Δ_i à `reliquat`.

6. Essayer de bien comprendre ce que fait cet algorithme, la raison pour laquelle il fonctionne, et montrer que l'on va bien atteindre le point (i_2, j_2) à l'issue de l'algorithme.

7. En déduire une fonction `line_case1(i1, j1, i2, j2, c)` implémentant l'algorithme de Bresenham, toujours dans le cas $i_1 \leqslant i_2$ et $j_1 \leqslant j2 \leqslant j1 + (i_2 - i_1)$. Le test doit donner le même résultat que précédemment, mais les droites liant les points rouges sont cette fois jaunes.

8. Proposer une fonction `line_case2(i1, j1, i2, j2, c)` s'inspirant de la fonction précédente pour traiter le cas $i_1 \leqslant i_2$ et $j_1 - (i_2 - i_1) \leqslant j2 \leqslant j1$ (droite de direction entre le sud et le sud-ouest). Le test devra faire apparaître trois paires de points et trois droites supplémentaires, vertes.

9. Enfin, faire de même pour des fonctions `line_case3(i1, j1, i2, j2, c)` vérifiant $i_1 \leqslant i_2$ et $j_2 > j_1 + (i_2 - i_1)$ (droite de direction entre l'est et le sud-est) et `line_case4(i1, j1, i2, j2, c)` vérifiant $i_1 \leqslant i_2$ et $j_2 \leqslant j_1 - (i_2 - i_1)$ (droite de direction entre l'ouest et le sud-ouest). Le test ajoute encore deux fois trois droites, magenta et cyan.

10. Utiliser les quatre fonctions précédentes pour construire une unique fonction `line(i1, j1, i2, j2, c)` tracant une ligne entre deux points (i_1, j_1) et (i_2, j_2) quelconques à l'intérieur de l'image. Le test devra afficher vingt-quatre paires de points et vingt-quatre droites jaune entre ces points, partant du centre de l'image dans toutes les directions.

Note : dans les implémentations usuelles de l'algorithme de Bresenham, on effectue un « décalage » de $\Delta_i/2$ de la valeur de `reliquat` car il est souvent plus facile de regarder si une valeur est negative ou positive plutôt que de la comparer à une valeur telle que $\Delta_i/2$. On a préféré ici faciliter la compréhension de l'algorithme, le changement ensuite n'est pas bien compliqué (on initialise `reliquat` à $\lfloor -\Delta_i/2 \rfloor$ et on fait la comparaison avec 0).

4 Davantage de primitives

4.1 Cercles et disques

On souhaite à présent tracer un cercle de centre (i_c, j_c) et de rayon $r \geqslant 0$, toujours aussi propre que possible. On s'intéresse tout d'abord à la section d'un huitième de cercle située entre le point à l'ouest et le point au sud-ouest.

11. Quelles sont les coordonnées du point à l'ouest ? Se convaincre que, comme dans le premier cas étudié pour les lignes, dans cette zone du cercle, on peut le tracer point par point avec seulement deux déplacements possibles : vers le bas, ou une combinaison d'un déplacement vers le bas et d'un déplacement vers la droite.

12. Comment détecter que l'on a atteint le point au sud-ouest ? On s'efforcera, comme précédemment, de ne travailler que sur des entiers (on pourra s'intéresser au composantes du vecteur allant du centre du cercle (i_c, j_c) au point courant (i, j)).

13. En s'inspirant de l'algorithme de Bresenham, réfléchir à une méthode permettant de

décider s'il faut effectuer un déplacement vers le bas ou en diagonale. Il s'agit d'un problème plus complexe que celui lié au tracé d'une droite, on ne cherche pas nécessairement à avoir la plus grande précision possible. On pourra par exemple choisir d'avoir les points les plus loins possibles de (i_c, j_c) mais sans jamais dépasser une distance $r + 0.5$.

Note : pour éviter les racines, une bonne idée consiste à toujours garder les distances au carré! Pour les dimensions usuelles d'une image, il n'y a pas de risque de provoquer de débordement.

14. En déduire une ébauche de fonction `circle(ic, jc, r, c)` traçant le huitième de cercle concerné. On supposera $r \geqslant 0$ et que l'intégralité du morceau de cercle est bien dans l'image.

15. En utilisant intelligement des symétries, modifier la fonction `circle(i, j, r, c)` afin qu'elle trace un cercle complet.

16. Proposer, en s'inspirant de `circle` (et en utilisant éventuellement `hline`), une fonction `circlef(ic, jc, r, c)` prenant en argument les coordonnées d'un centre, un rayon et une couleur et affichant un disque centré en (i, j) de rayon `r` (toujours supposé positif) et de couleur `c`. On supposera que l'intégralité du disque est bien dans l'image. On pourra s'aider de `hline` :

17. Modifier la fonction précédente pour que la fonction `circlef(i, j, r, c)` accepte un disque pouvant dépasser des limites de l'image et n'essaie pas de modifier la couleur de points situés en-dehors de l'image. On pourra supposer qu'il n'y a pas de risque de débordement dans les calculs.

4.2 Triangles colorés

18. Proposer une fonction `trianglef(i1, j1, i2, j2, i3, j3, c)` prenant les coordonnées de trois sommets d'un triangle (on suppose tous les points dans l'image) et trace un triangle rempli avec la couleur `c`.

Les fichiers `basic_ppm` fournissent également une fonction supplémentaire appelée `set_pixel_rgb(i, j, r, g, b)` prenant en argument les coordonnées d'un point de l'image et trois entiers compris entre 0 et 255, représentant les composantes rouge, vert et bleu d'une couleur, et changeant la couleur du point spécifié selon les arguments fournis.

19. Proposer une fonction `trianglec(i1, j1, i2, j2, i3, j3)` traçant dans l'image un triangle plein tel que le sommet 1 est rouge, le sommet 2 est vert, le sommet 3 est rouge, et que pour tout point à l'intérieur du triangle est une couleur constituée d'un mélange des trois couleurs primaires, interpolées linéairement depuis les trois sommets (le barycentre du triangle sera gris, les milieux des côtés jaune foncé, cyan foncé et magenta foncé).