Planning:

- Présentation et fonctionnement de X-Window
- MinilibX : la totale
- Figures simples
- Projections 3D $\rightarrow$ 2D
- Ray-casting (déjà des maths)
- Principe d'un Ray-Tracer
- Beurk (encore) des maths
- Première version
- De quoi s'amuser plus sérieusement

X-Window

- C'est un environnement utilisateur graphique sous UN*X (par opposition à un mode dit "texte" ou "console")
- Communément appelé X11 ou X11R6 (Version 11 Release 6)
- C'est le système graphique le plus courant sous UN*X
- Il a été conçu pour un environnement réseau

Architecture en 2 parties

- Une première partie appelée *Serveur X*

  - Pour chaque ensemble écran (ou `DISPLAY`) - souris - clavier, il y a un unique programme appelé serveur X.

  - C'est lui qui contrôle (au travers d'unix) l'affichage à l'écran, les mouvements de la souris, et qui récupère les touches tapées au clavier.

  - Il gère le système de fenêtrage que l'on a à l'écran.

  - Le serveur X dispose d'un protocole de dialogue grâce auquel il communique avec les programmes voulant utiliser un élément de notre ensemble écran - souris - clavier

  - `top` ou bien `ps -auwx` vous permettront de voir le programme serveur X. *XF86_S3*, *XFree86*, *XF86_SVGA*, ou encore *X* sont des noms courant pour un serveur X.

  - Comparez un PC et une alpha : pas de serveur X sur une alpha.

- Une deuxième partie appelée *Client X*

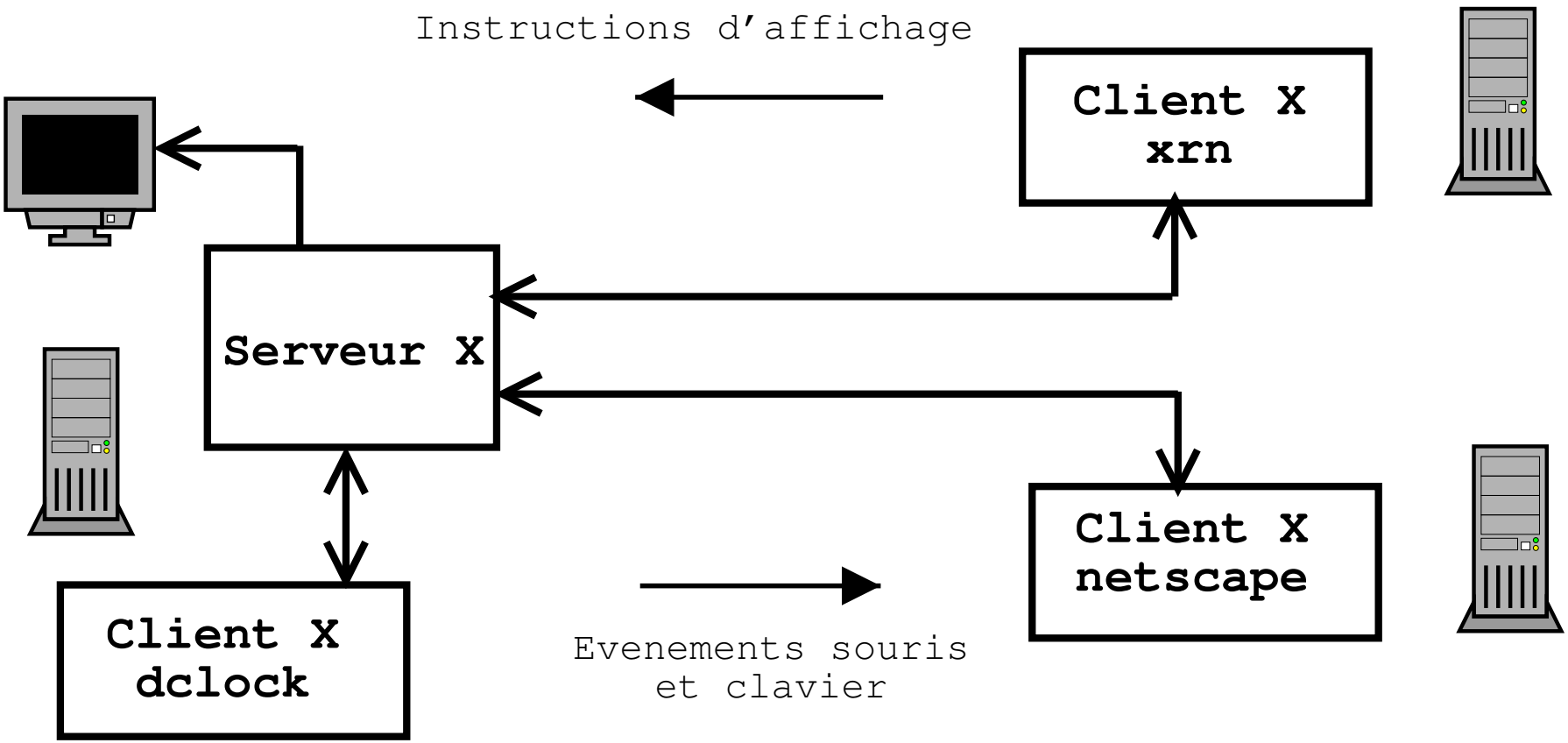
- C'est un programme qui souhaite obtenir des informations de la part du clavier, de la souris, et qui veut en afficher à l'écran.

- Ce programme peut se situer n'importe-où sur le réseau. Il suffit de pouvoir établir une connexion réseau entre la machine où est le programme client X, et la machine où est le serveur X. $\rightarrow$ un programme sur une alpha peut afficher quelque chose sur votre écran.

- *netscape*, *dclock*, *gimp* ou bien encore *xrn* sont des exemples de clients X, qui grâce au reseau vont se connecter au serveur X associé à votre écran et lui demander d'afficher des fenêtres avec du texte, des images...

- En échange, le serveur X communiquera à ces clients X les données entrées au clavier et les mouvements de souris: ce sont les *Evènements*.

Architecture en 2 parties



La MiniLibX

- Les éléments de base
- La gestion des évènements
- L'utilisation d'images temporaires

Déjà vu

- Etablissement de la connexion réseau entre le client X et le serveur X:

```
void *mlx_init()
```

- Création d'une nouvelle fenêtre à l'écran:

```
void *mlx_new_window(void *mlx_ptr,int width,int height,char *title)
```

- Affichage d'un pixel de couleur dans la fenêtre :

```
int mlx_pixel_put(void *mlx_ptr,void *win_ptr,int x,int y,int color)
```

Exemple déjà vu

```
int    main()
{
  void *mlx_ptr;
  void *win_ptr;

  mlx_ptr = mlx_init();
  win_ptr = mlx_new_window(mlx_ptr,500,500,"Test 1");
  mlx_pixel_put(mlx_ptr,win_ptr,250,250,0xFFFFFF);
  while (42)
    ;
}
```

Les Evènements

- *Les 3 évènements disponibles avec la minilibX:*

  - Une touche a été tapée au clavier

  - Un bouton de la souris a été utilisé

  - Une partie de votre fenêtre a été effacé (appelé évènement *Expose*).

Les Evènements et la minilibX

- Nous allons fournir à la minilibX trois fonctions qui seront respectivement exécutées pour chaque type d'évènement qui se produit.

- La minilibX transmettra aussi des informations plus détaillées comme la touche qui a été frappée, ou avec quel bouton de la souris on a cliqué.

Evènement clavier

- Lorsqu'une touche a été frappée au clavier, la minilibX appellera votre fonction (que l'on baptise `gere_key` par exemple) de la façon suivante :

  ```
  gere_key(keycode,param);
  ```

  avec `int keycode` et `void *param` .

- Comment signaler à la minilibX qu'il faudra utiliser la fonction `gere_key` ? Grâce à :

  ```
  mlx_key_hook(void *win_ptr,int (*funct_ptr)(),void *param)
  ```

- Par exemple :

```
struct s_machin   my_var;
...
mlx_key_hook(win_ptr,gere_key,&my_var);
```

Evènement souris

- Comme pour le clavier, la minilibX appellera notre fonction d'une façon bien définie :

  ```
  gere_mouse(button,x,y,param);
  ```

  avec `int button`, `int x`, `int y` et `void *param` .
  - `button` peut prendre les valeurs 1,2 ou 3 pour le bouton gauche, milieu ou droit (2 boutons simultanés = 2 évènements)
  - `x` et `y` sont les coordonnées de la souris au moment du click, relatives au coin supérieur gauche de la fenêtre.

- Du côté de la minilibX :

  ```
  mlx_mouse_hook(void *win_ptr,int (*funct_ptr)(),void *param)
  ```

- Par exemple :

```
struct s_machin   my_var;
...
mlx_mouse_hook(win_ptr,gere_mouse,&my_var);
```

Evènement expose

- Lorsqu'une partie ou la totalité de notre fenêtre doit être réaffichée, la minilibX appellera notre fonction :

  ```
  gere_expose(param);
  ```

  - Dès la première apparition de la fenêtre, un évènement expose sera envoyé à la minilibX (à votre programme ..).

- Du côté de la minilibX :

  ```
  mlx_expose_hook(void *win_ptr,int (*funct_ptr)(),void *param)
  ```

- Par exemple :

```
struct s_machin   my_var;
...
mlx_expose_hook(win_ptr,gere_expose,&my_var);
```

Sauf que...

- Sauf qu'il en manque un bout :

  ```
  mlx_loop(void *mlx_ptr);
  ```

  Cette fonction de la minilibX va faire une boucle infinie pour traiter les évènements en provenance du serveur X et exécuter la bonne fonction.

- La boucle infinie contient les actions suivantes :

  - *Y-a-t-il un nouvel évènement ?*
    - *Oui : appel à la fonction associée à cet évènement*
  - *Appel à la fonction générique*

- La "fonction générique" est une de vos fonctions communiquée à la minilibX grâce à :

  ```
  mlx_loop_hook(void *mlx_ptr,int (*funct_ptr)(),void *param)
  ```

  et sera exécutée de la façon suivante :

  ```
  gere_loop(param);
  ```

Indispensable

- *Attention:* l'appel à la fonction `mlx_loop` est indispensable pour que votre programme fonctionne avec les évènements.

- Exemple typique :

```
int    main()
{
  void *mlx_ptr;
  void *win_ptr;
 [...]
  mlx_ptr = mlx_init();
  win_ptr = mlx_new_window(mlx_ptr,500,500,"Test 1");
  mlx_expose_hook(win_ptr,gere_expose,&my_var);
  mlx_key_hook(win_ptr,gere_key,&my_var);
  mlx_loop(mlx_ptr);
}
```

Les images

- Ça rame ?

- On vous l'a déjà dit, c'est normal.

- Mais on va quand même accélérer les choses :
  - Création d'une image temporaire en mémoire.
  - On dessine dans cette image, pas dans la fenêtre à l'écran.
  - On affiche d'un seul coup le contenu de l'image dans la fenêtre.

Détour par la carte vidéo

- L'image temporaire qui sera créée par la minilibX aura le même format que l'image stockée par la carte video, afin d'optimiser la copie entre les deux.

- Pour chaque point de l'image, on va stocker une couleur. Suivant la configuration, plus ou moins de couleurs seront disponibles simultanément à l'écran. Les configurations connues sont *256*, *32000* et *16 millions* de couleurs.

- Plus il y a de couleurs à l'écran, plus il faut de place en mémoire pour stocker la couleur d'un pixel :
256 $\rightarrow$ *1 octet*, *32000* $\rightarrow$ *2 octets*, *16M* $\rightarrow$ *3 octets*.

Le format de nos images

- Une image sera stockée dans une unique zone continue de mémoire.

- Chaque pixel aura en mémoire 1, 2 ou 3 octets dédiés.

- Ces groupes d'octets représentants des pixels sont rangés de gauche à droite et de haut en bas, du pixel $(0,0)$ jusqu'à celui situé en bas à droite.

- Concrètement :
  Une image de 100×50 pixels, soit 5000 pixels, et chaque pixel est codé sur 3 octets. On a donc une image stockée dans une zone mémoire de 15000 octets. Pour passer d'un pixel au suivant (vers sa droite), on saute 3 octets. Pour passer d'une ligne à la suivante (vers le bas), on saute 100 pixels $\times$ 3 octets, soit 300 octets.

Couleur RGB $\neq$ octets en mémoire

- Les octets en mémoire pour chaque pixel ne correspondent pas forcément aux 3 octets d'une couleur exprimée en RGB.

- La minilibX contient une fonction de transfert entre une couleur définie par ses composantes RGB et les octets qui seront stockés en mémoire vidéo (ou dans une image):

```
unsigned int  mlx_get_color_value(void *mlx_ptr,int color)
```

- `color` contient sur 3 des 4 octets les composantes RGB de notre couleur. La fonction renvoie un `unsigned int` qui contient les 8/16/24 bits contenus dans la mémoire vidéo pour coder la couleur voulue. On mettra ces mêmes bits dans les octets de l'image temporaire correspondants au pixel voulu.

Création d'une image

- Réalisé au moyen de :

  `void *mlx_new_image(void *mlx_ptr,int width,int height)`

  `width` et `height` définissent la largeur et la hauteur de l'image créée. La fonction renvoie un identifiant qui permettra l'utilisation ultérieure de l'image.

- Exemple :

```
[...]
void *mlx_ptr;
void *img_ptr;
[...]
mlx_ptr = mlx_init();
[...]
img_ptr = mlx_new_image(mlx_ptr,300,300);
```

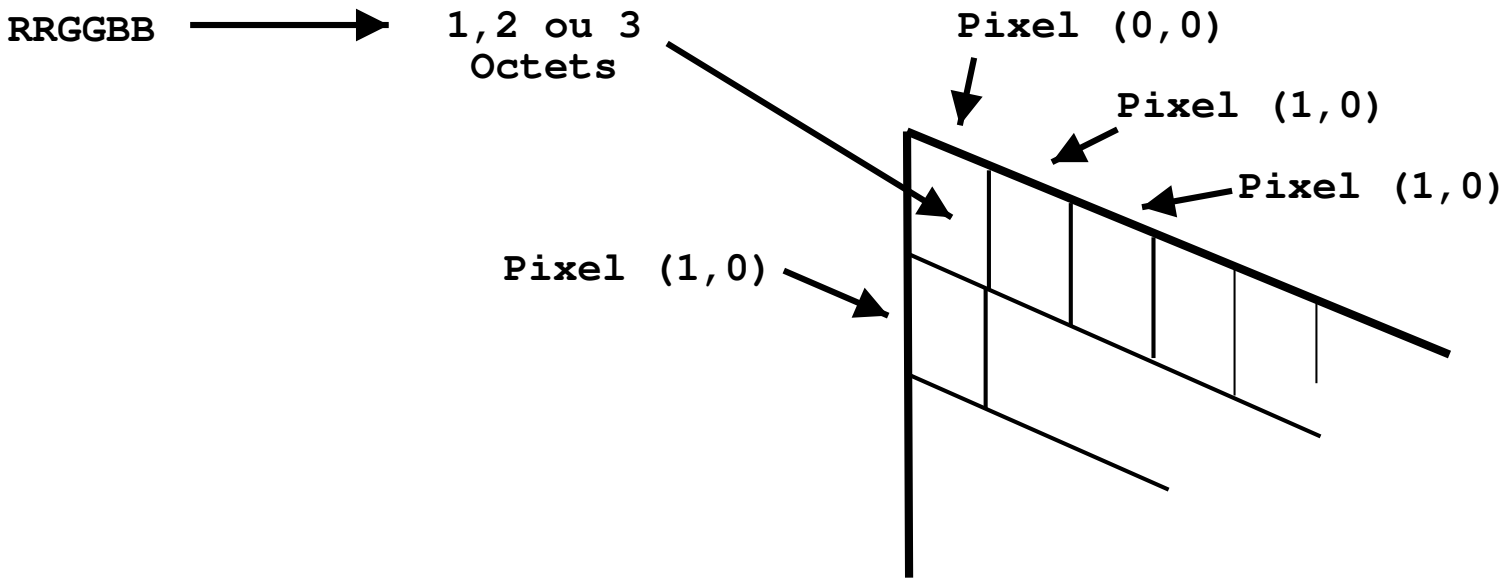
Remplir l'image

- La minilibX va nous fournir les informations nécessaires pour remplir correctement l'image que l'on vient de créer :

```
char *mlx_get_data_addr(void *img_ptr,int *bits_per_pixel,
                        int *size_line,int *endian)
```

- La fonction renvoie un `char *` qui pointe sur la zone mémoire où sont stockés les octets des pixels. Les 3 variables `bits_per_pixel`, `size_line` et `endian` contiendront respectivement:
  - le nombre de bits dédiés à un pixel (8, 16 ou 24)
  - la taille en octets d'une ligne de pixels
  - l'*endian* de la machine où se trouve le serveur X (0: little endian, 1: big endian).

Remplir l'image



- Exemple :

```
char *data;
int  bpp;
int  sizeline;
int  endian;
[...]
data = mlx_get_data_addr(img_ptr,&bpp,&sizeline,&endian);
```

Recommandations

- *Attention:* la fonction `mlx_get_color_value` vous renvoie un `int`. Dans cet `int`, seuls certains octets seront utiles (ceux que l'on va mettre dans notre image).
  Il s'agira toujours des *octets de poids faible*, c'est à dire les octets à droite lors de la représentation en hexadécimal, comme on le fait pour mettre nos composantes RGB dans un `int`.

- Faits attention à l'endian de la machine sur laquelle vous vous trouvez et l'endian du serveur X: Les octets de poids faible ne sont pas toujours situés au même endroit. Dans certaines configurations, vous pourrez être ammenés à inverser l'ordre des octets à mettre dans votre image.

Pour un pixel

- Changer la couleur sur un pixel va nécessiter les manipulations suivantes :
  - Trouver en mémoire le début de la ligne de notre pixel
  - Trouver les octets dédiés à notre pixel sur cette ligne
  - Changer ces octets par ceux de la nouvelle couleur

- Exemple :
  On veut changer la couleur du pixel 42×142 pour mettre du orange (`0x00FFA500`). On fonctionne en 16 bits par pixels, soit 2 octets.
  Les 2 octets de mon pixel sont donc à l'adresse
  `data+142×size_line+2×42`
  Il ne reste qu'à les garnir avec les 2 octets obtenus par `mlx_get_color_value`.

Affichage de l'image

- Une fois notre image créée et remplie, il ne reste qu'à la transmettre au serveur X pour l'afficher dans une fenêtre:

```
mlx_put_image_to_window(void *mlx_ptr,void *win_ptr,
                        void *img_ptr,int x,int y)
```

En plus des trois identifiants (connection, fenêtre et image), on trouve les coordonnées `x` et `y` (relatives à la fenêtre) où l'image sera affichée.

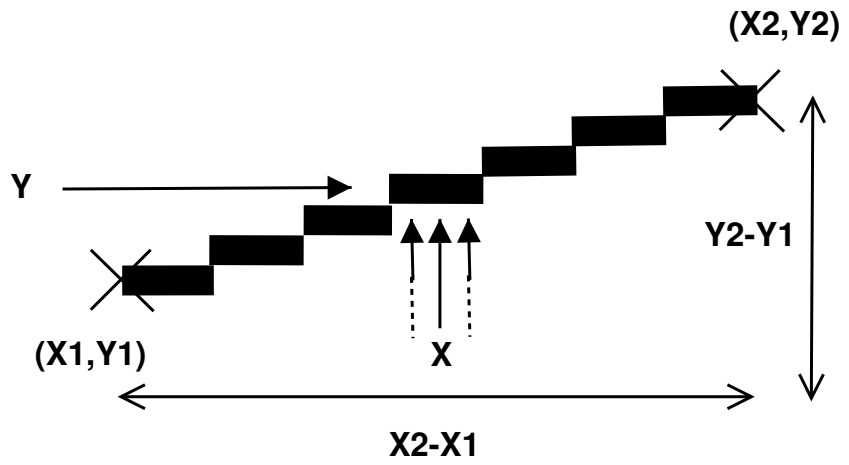
Figures simples

- droite
- cercle
- ellipse
- remplissage

La droite

- On trace une droite entre les points $(x1, y1)$ et $(x2, y2)$.

- Première étape : on traite un seul cas.
  $x1 <= x2$ et $(x2 - x1) >= abs(y2 - y1)$



- Pour chaque y il y a plusieurs x possibles, mais pour chaque x il n'y a qu'un seul y. Pour tout x entre $x1$ et $x2$, on calcule l'unique y (simple règle de 3, ou Thalès), et on affiche le pixel correspondant.

La droite

- Exemple :

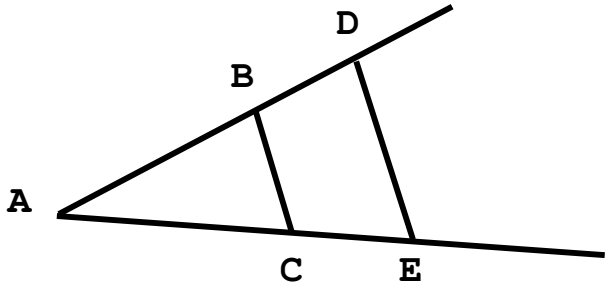
```
int draw_cas_un(int x1,int y1,int x1,int y2,...)
{
  int x;

  x = x1;
  while (x<=x2)
   {
     mlx_pixel_put(...,...,x,y1+((y2-y1)*(x-x1))/(x2-x1),...);
     x++;
   }
}
```

La règle de 3

- Appelez ça comme vous voulez, *règle de trois*, *Thalès*, *produit en croix* ou encore *proportionnalité*, la seule chose qu'on vous demande, c'est de le connaître.

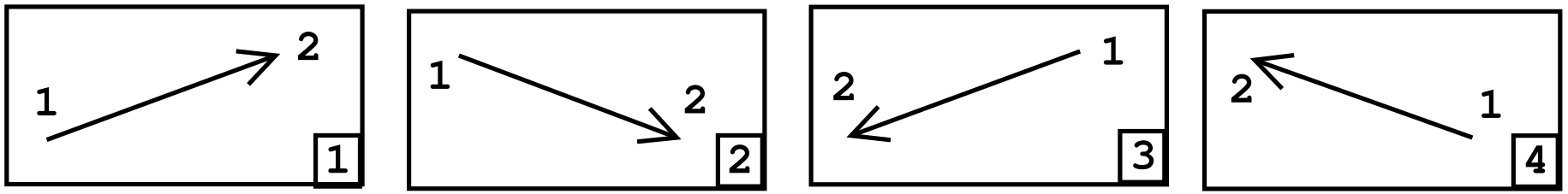
- *Si on a 25 pommes pour 5 personnes, combien a-t-on de pommes pour 8 personnes ?*
  Et combien a-t-on de personnes pour 30 pommes ?

- Ou bien encore :



$$\frac{AB}{AD} = \frac{AC}{AE} = \frac{BC}{DE}$$

La droite - les autres cas



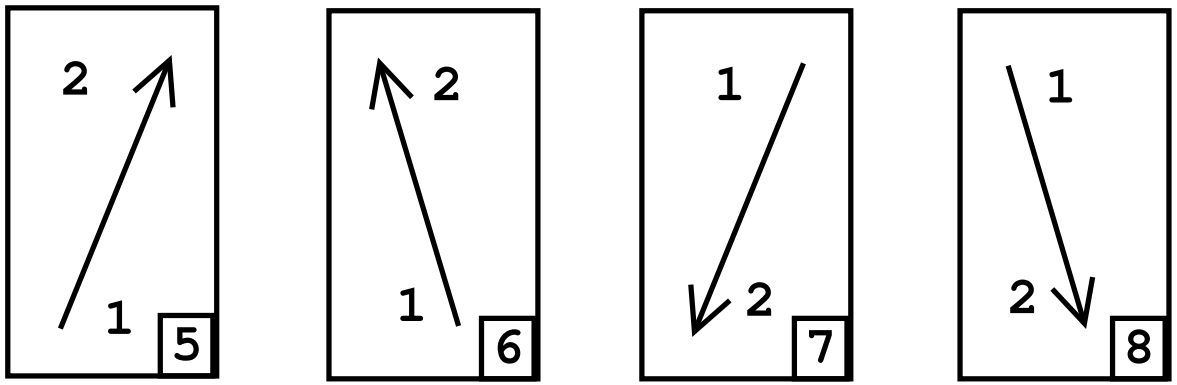
1 - C'est notre premier cas.

2 - L'algorithme de notre premier cas fonctionne. En effet, si $(y2 - y1)$ est négatif, le résultat de la règle de 3 le sera aussi.

3 - En échangeant les 2 points, on retrouve le cas 1. Il faut appeller la fonction de tracé du cas 1 en inversant $(x1, y1)$ et $(x2, y2)$.

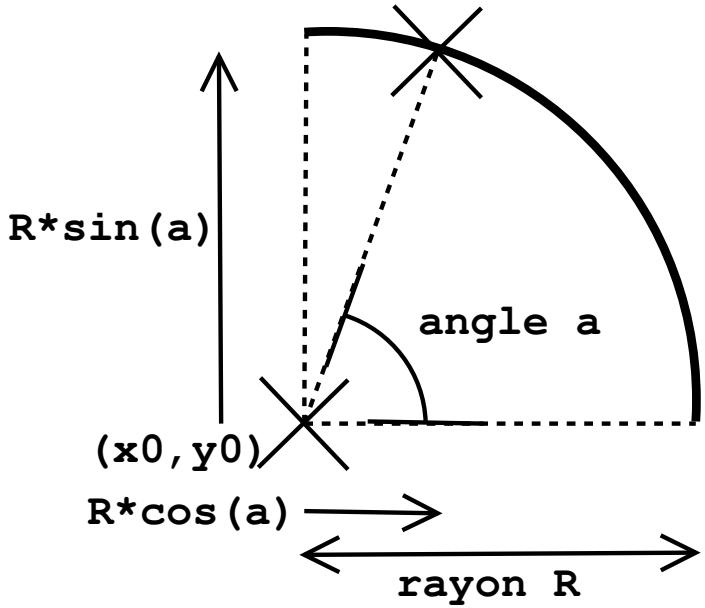
4 - En échangeant les 2 points, on retrouve le cas 2.

La droite - les autres cas



5 - Cette fois-ci, c'est pour chaque y qu'il n'y a qu'un seul x. Pour tout y entre $y1$ et $y2$, on calcule l'unique x et on met le pixel correspondant. Il faut créer une nouvelle fonction. La fonction du premier cas pourra être utilisée, mais en inversant $x1$ et $y1$, $x2$ et $y2$, et en effectuant un `mlx_pixel_put(..,y,x,..)` au lieu d'un `mlx_pixel_put(..,x,y,..)`.
6 - Ce cas est traité par le cas 5 (mêmes raisons que pour le 2).
7 - En échangeant les 2 points, on retrouve le cas 5.
8 - De même, l'échange des 2 points rejoint le cas 6.

Le cercle



- Les points sur le cercle ont pour coordonnées : $(x0 + R \times \cos(a), y0 + R \times \sin(a))$

- Il suffit de faire varier a entre 0 et 360 (ou 2π) pour obtenir tous les points du cercle.

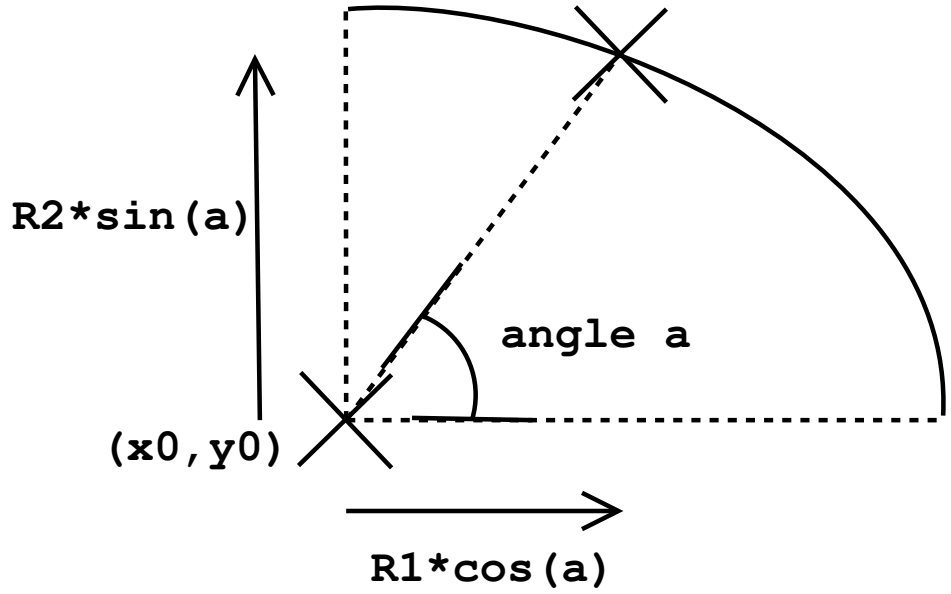
Cercle à trou ?

- Concrètement, pour chaque nouveau pixel, on augmente l'angle a un petit peu. Plus le cercle est grand, plus l'augmentation de a devra être petite, sinon on a un cercle à trous.

- Il faut donc pour chaque cercle calculer de combien on augmente l'angle. Le nombre de pixels appartenant au cercle est à peu près inférieur au nombre de pixel du carré contenant le cercle: $8 \times R$. C'est le nombre de pixel que l'on va tracer.
Pour cela, on augmente donc notre angle de $\frac{2\pi}{8R}$.

Ellipse

- La façon de tracer une ellipse est identique à celle du cercle, mais avec un rayon différent entre les x et les y.



En C

- Ok pour les `double` ainsi que pour l'utilisation des fonctions `sin()` et `cos()` de la librairie mathématique.

- Attention à ne pas oublier

  ```
  #include <math.h>
  ```

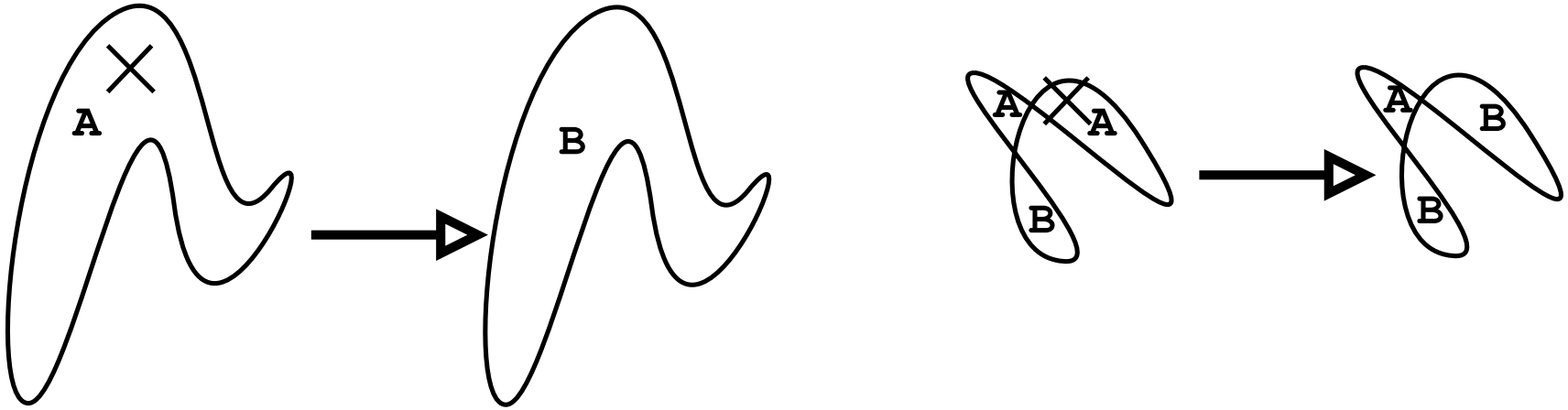
  ou le prototypage de `sin()` et `cos()` sinon vos `double` seront transformés en `int` avant d'en prendre le cosinus ou le sinus.

- Les fonctions `sin()` et `cos()` sont dans la librairie mathématique. N'oubliez pas `-lm` à la compilation.

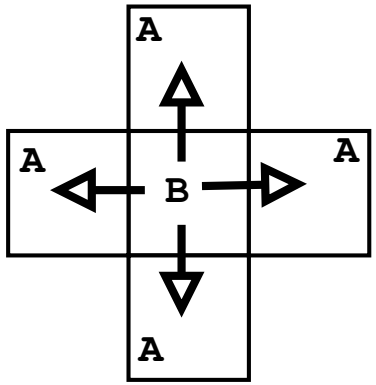
- N'utilisez les doubles *QUE SI C'EST VRAIMENT NECESSAIRE*. Les calculs sur `int` sont plus rapides (cf. miniproj 2).

Remplissage

- On opère dans une image avec un dessin quelconque.
- Le remplissage consiste à remplacer la couleur A d'une forme continue par une couleur B donnée.
- La fonction de remplissage reçoit en paramètre les coordonnées d'un pixel dans la forme à remplir. De proche en proche, elle atteint tous les pixels de la forme.



Remplissage



- La fonction de remplissage va se rappeller elle-même 4 fois avec les coordonnées de chacun de ses voisins. Sauf si un voisin n'est pas de la couleur A, ou bien hors de l'image.

- Cela implique de pouvoir connaitre la couleur d'un pixel. X-Window ne le permet pas pour un pixel affiché à l'écran. Il faut donc travailler dans une image, et faire une fonction opposée de celle qui change un pixel dans une image.