LISV Laboratoire d'ingénierie des systèmes de Versailles

UVSQ université PARIS-SACLAY

Projet de programmation C

Pixel Tracer

Halim Djerroud

Février 2024

Table des matières

Présentation du projet

Préambule

Une image numérique est un fichier informatique permettant de stocker une image dans la mémoire de l'ordinateur (sous forme binaire). La création de cette image numérique peut être effectuée à partir de la saisie d'une image réelle par l'intermédiaire de dispositifs matériels (appareil photo numérique, caméra numérique, scanner, ...) ou produite complètement par un ordinateur, on parle alors d'image de synthèse. Une fois l'image représentée sous forme binaire, il est possible d'effectuer des traitements (modifications, transformations, filtres, ...) sur l'image grâce à des logiciels graphiques.

Comme souvent en informatique, la manière choisie pour représenter une donnée informatique va conduire à obtenir certains avantages mais aussi des inconvénients en fonction du contexte d'utilisation. Il existe deux types de représentation pour les images numériques :

— `Image matricielle` : les données de l'image sont représentées sous la forme d'une matrice de points à plusieurs dimensions. Pour des images à 2 dimensions les points sont appelés pixels.

Avantages : possibilité de modifier l'image pixel par pixel ce qui permet des nuances de couleurs importantes (dégradés, ombres, ...) et d'avoir des effets de textures.

Inconvénients : fichier lourd (même compressé), perte de qualité lors d'un agrandissement (visuel flou aussi appelé pixélisation de l'image). Une illustration du problème de pixélisation est donné en figure 1.

FIGURE 1 – Problème de pixélisation lors d'un agrandissement d'une zone de l'image matricielle. Image extraite du film *The Matrix* réalisé par Les Wachowski.

- Exemples de formats de fichiers matriciels : jpeg, gif, png, bmp - Exemples de logiciels pour la création d'images matricielles : Photoshop, After Effects, Gimp

— `Image vectorielle` : les données de l'image sont représentées d'un point de vue mathématique. Une image vectorielle décrit uniquement les formes géométriques de l'image avec différents attributs que l'ordinateur est chargé de tracer à l'écran.

Avantages : fichier léger, aucune de perte de qualité lors d'un redimensionnement (pas d'effet de pixélisation). Un exemple d'agrandissement sur une image vectorielle est donné en figure 2.

Inconvénients : Ne permet de représenter que des formes géométriques simples (segments,

arcs, cercles, courbes, ...) donc non adapté à la photographie.

FIGURE 2 – Aucune perte de qualité lors de l'agrandissement d'une zone de l'image

- Exemples de formats de fichiers vectoriels : svg, ai, eps, pdf
- Exemples de logiciels pour la création d'images vectorielles : Illustrator, CorelDRAW, *Inkscape*, *librecad*, *Dia Diagram Editor*

Note : Il est possible de convertir les images d'un type vers l'autre.

Dans le cadre de ce projet nous allons nous intéresser à la réalisation des images vectorielles et plus particulièrement au fonctionnement des logiciels de dessin vectoriel. Le dessin vectoriel est très utilisé pour la création d'illustrations, de graphiques ou de cartes, car il offre à l'utilisateur la possibilité d'observer avec précision et sans perte de détails les zones qui l'intéressent en fonction de l'échelle souhaitée.

1. Le dessin vectoriel

1.1. Principe de base

Le principe des logiciels de dessin vectoriel est de stocker les informations minimales permettant de reconstruire une forme géométrique à dessiner.

Par exemple, pour une forme cercle, il suffit de stocker la position du centre et le rayon en mémoire. Ce n'est qu'au moment de l'affichage à l'écran que les positions des pixels à colorier sont calculées afin de faire apparaître la forme géométrique finale. Il est bien évidemment possible d'ajouter d'autres informations en mémoire comme par exemple la couleur du contour du cercle ou sa couleur de remplissage.

Dans les logiciels de dessin vectoriel l'outil plume est très important car il permet de créer des formes plus complexes comme des courbes. Les courbes sont représentées en mémoire par des courbes de Bézier. Ces courbes sont polynomiales dont le degré est variable. Par exemple, une courbe de Bézier cubique est une courbe polynomiale de degré 3 et est définie par 4 points : 2 points correspondant aux extrémités de la courbe et 2 points correspondant à des points de contrôle. La position des points de contrôle permet d'ajuster la courbure.

Pour concevoir des dessins complexes, il est nécessaire de pouvoir travailler sur une partie de l'image sans affecter d'autres parties. L'image globale est décomposée en un ensemble de couches empilées les unes sur les autres. Chaque couche regroupe ainsi une partie des formes géométriques (i.e. les informations permettant sa reconstruction) de l'image suivant certains critères (type de forme, zone de l'image, etc). Il est donc possible de retoucher certaines parties de l'image de manière indépendante. C'est ce qu'on appelle en infographie les **calques**.

Une image vectorielle est composée d'une hiérarchie de plusieurs calques superposés les uns sur les autres dont chacun va regrouper un ensemble de formes géométriques ayant des propriétés communes. L'ensemble des formes géométriques d'un calque sont elles mêmes hiérarchisées à l'intérieur du calque. Cette hiérarchisation à plusieurs niveaux est très importante car elle précise l'ordre dans lequel les calques, et donc les formes géométriques qui les composent, vont se superposer lors de l'affichage à l'écran. Ainsi, une forme géométrique située à un niveau inférieur dans cette hiérarchie peut être partiellement ou entièrement recouverte par une forme géométrique d'un niveau supérieur (appartenant au même claque ou à un calque de niveau supérieur).

L'utilisation de calques permet ainsi d'appliquer des modifications ciblées sur des groupes de formes géométriques tout en donnant la possibilité de verrouiller l'accès sur le groupe afin d'éviter des erreurs de manipulation. Le système de calques est un élément essentiel pour les logiciels de dessin vectoriel et offre de nombreuses fonctionnalités pour travailler de manière précise sur les éléments d'une image afin de produire des images de qualité.

Ce qu'il faut retenir sur les calques :

- un calque peut contenir plusieurs formes géométriques ;
- une forme géométrique ne peut appartenir qu'à un seul calque ;
- les modifications attribuées sur un calque sont effectuées sur toutes les formes géométriques du calque ;
- les modifications attribuées sur une forme géométrique sont effectuées seulement sur cette forme géométrique ;
- il est possible de déplacer une forme géométrique d'un calque à un autre ;
- si un calque est déplacé dans la hiérarchie des calques, toutes les formes géométriques le constituant sont déplacées avec celui-ci.

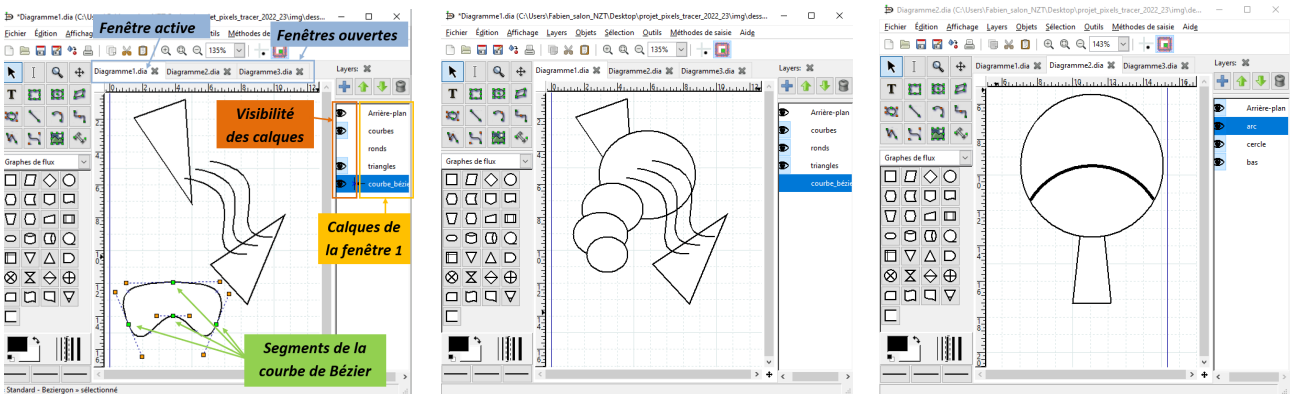
Les opérations principales sur les calques :

- créer un nouveau calque ;
- déplacer un calque dans la hiérarchie (monter ou descendre) ;
- renommer un calque ;
- supprimer un calque : entraîne la suppression des formes géométriques qu'il regroupe ;
- rendre visible ou invisible le calque : permet l'affichage à l'écran ou non des formes géométriques qu'il regroupe ;
- verrouiller ou déverrouiller le calque : autorise la modification ou non des formes géométriques qu'il regroupe.

La section suivante illustre ce système de superposition de calques à partir d'interfaces de logiciels de dessin vectoriel.

1.2. Exemples d'interface de logiciel de dessin vectoriel

Le logiciel Dia Diagram Editor (nommé Dia dans la suite du document) est logiciel de dessin vectoriel orienté pour la création de diagramme. La figure 3 montre l'interface du logiciel Dia. Ce logiciel donne la possibilité à l'utilisateur d'ouvrir plusieurs fenêtres dans une même session pour créer plusieurs images vectorielles et basculer de l'une vers l'autre facilement. Chaque fenêtre est composée d'une superposition de calques regroupant différentes formes géométrique.



(a) Contenu fenêtre 1 avec calque ronds non visible

(b) Contenu fenêtre 1 avec calque courbe de Bézier non visible et calque ronds visible

(c) Contenu fenêtre 2 et ses différents calques

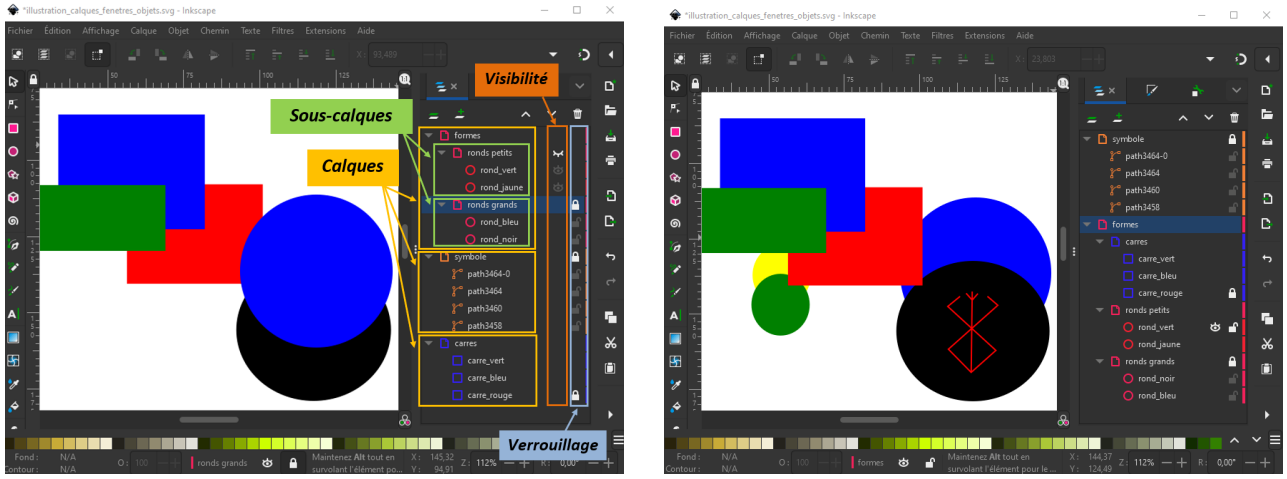
FIGURE 3 – Logiciel Dia Diagram Editor avec son système de calques par fenêtre

Sur la figure 3a le calque `ronds` n'est pas visible donc l'ensemble des formes géométriques de ce calque ne s'affichent pas sur la zone de dessin. On peut observer sur cette même figure une courbe de Bézier assez complexe ses différents points de contrôle visibles.

Sur la figure 3b le calque rond est maintenant visible tandis que le calque `courbe_bézier_fermée` ne l'est plus. Les quatre cercles regroupés dans le calque `ronds` sont donc présents dans la zone de dessin tandis que la courbe de Bézier fermée n'apparaît plus.

Sur la figure 3c c'est la fenêtre 2 qui est active. Dans cette fenêtre tous les calques sont visibles et s'affichent dans la zone de dessin qui lui est propre. Il est intéressant de remarquer le principe de superposition de calques sur cette figure. Dans la hiérarchie de calques de cette fenêtre le calque `ronds` est en dessous du claque `courbes` et au dessus du calque `triangles`. Lors du tracé dans la zone de dessin l'ensemble des quatre cercles du calque `ronds` apparaissent donc par dessus les triangles. Les trois courbes, regroupées dans le calque `courbes`, apparaissent par dessus les triangles et les cercles.

Le logiciel Inkscape est un logiciel de dessin vectoriel gratuit et open source. Il permet de faire des illustrations, des icônes, des logos ou diagrammes. Ce logiciel a les mêmes icônes que le logiciel Dia pour le verrouillage et la visibilité des calques. Il propose cependant la possibilité de créer des calques imbriqués comme illustré sur la figure 4. Un calque peut être composé d'un ensemble de sous-calques, chacun composé d'un ensemble de formes géométriques.



(a) hiérarchie originale (b) hiérarchie modifiée

FIGURE 4 – Logiciel Inkscape avec son système de calques imbriqués

On peut remarquer sur la figure 4a que le sous-calque `ronds_petits` est non visible. Il n'apparaît donc pas sur le dessin et c'est donc la forme `rond_bleu` qui apparaît au premier plan dans la zone de dessin car il est placé le plus haut dans la hiérarchie et visible.

Sur la figure 4b la hiérarchie de calques a été modifiée par rapport à celle présente sur la figure 4a. Le calque `carres` (de couleur bleu dans zone des calques) a été intégré dans le calque `formes` (de couleur rouge). Il est donc devenu un sous-calque du calque forme et se retrouve au dessus des sous-calques `ronds_petit` et `ronds_grand` (de couleurs rouges). La forme `rond_noir` a été remontée dans la hiérarchie du sous calque `ronds_grands`. Il recouvre donc une partie du rond bleu sur le dessin dorénavant. Le calque `symbole` (de couleur orange) a été déplacé au dessus de tous les autres calques dans la hiérarchie. Ce calque regroupe un ensemble de courbes de Bézier qui apparaissent donc au premier plan sur la zone de dessin (par dessus le rond bleu et le rond noir).

2. Objectifs du projet

Dans ce projet, nous souhaitons développer une application de dessin vectoriel en mode textuel.

L'application à réaliser devra au minimum permettre à un utilisateur la création d'une image vectorielle simple à l'aide d'un ensemble de commandes utilisateurs et d'afficher cette image en mode textuel. L'application doit donc être capable de gérer au minimum une fenêtre et un calque regroupant toutes les formes géométriques créées par l'utilisateur.

Pour ce faire, nous devons nous intéresser à trois aspects principaux :

— la représentation mémoire de toutes les informations de notre application. En particulier la manière de sauvegarder les informations des formes géométriques permettant de les afficher ;
— une interface en ligne de commandes (sous la forme d'un menu) permettant à l'utilisateur d'effectuer les actions permises par l'application ;
— l'affichage à l'écran en mode textuel de l'image créée en appliquant les algorithmes de dessin décrits dans les sections correspondantes.

A partir de cette version minimale, il vous sera demandé d'implémenter diverses améliorations comme par exemple :

— la création de forme plus complexes : les courbes de Bézier ;
— une optimisation du code pour l'ajout, la suppression ou le déplacement d'une forme dans la hiérarchie (utilisation de listes chaînées) ;
— un système de calques multiples (imbriqués ou non) ;
— un système de fenêtres multiples ;

Vous pourrez bien évidemment implémenter d'autres améliorations comme la possibilité : d'appliquer des transformations sur les formes de l'image (rotation, translation, etc), zoomer/dézoomer, de déplacer une forme d'un calque à un autre, déplacer des calques dans la, de sauvegarder / charger une image dans un fichier texte, de proposer des commandes permettant d'annuler ou refaire une ancienne commande (Undo / Redo).

2.1. Représentation mémoire des données

L'idée est de réfléchir à la représentation des formes géométriques, des fenêtres et des calques. La figure 5 illustre une possible organisation des formes. Sur cette figure on peut constater que les données dans l'application sont stockés comme des tableaux imbriqués. L'application possède plusieurs fenêtres (tableau de fenêtres) dont chacune est composée d'un tableau de calques. Chaque calque peut contenir un ensemble de formes géométriques identiques ou différentes.

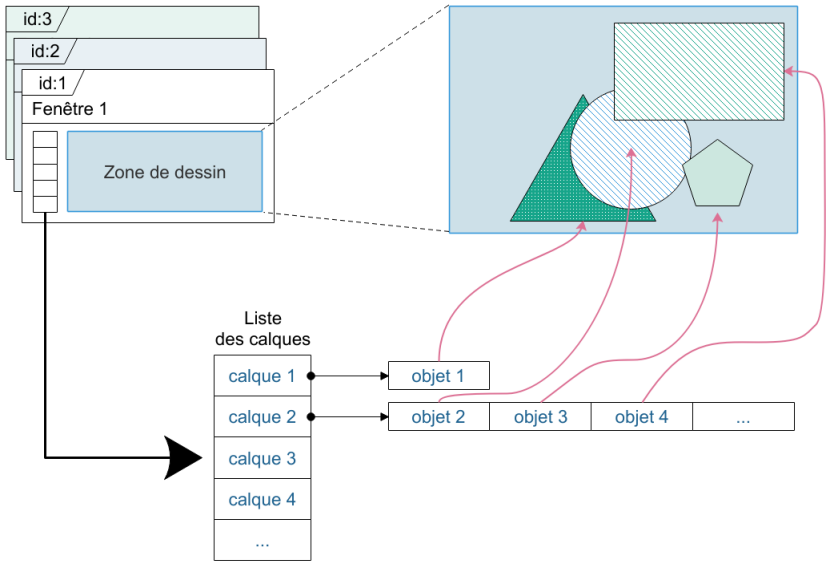


FIGURE 5 – Schéma mémoire simplifié d'une application de dessin vectoriel.

2.2. Interface en ligne de commande

Nous allons devoir créer un menu permettant à l'utilisateur d'effectuer les actions permises dans le logiciel telles que l'ajout d'une forme géométrique au calque ou l'affichage des informations des formes déjà ajoutées.

Cette interface utilisateur se présentera sous la forme d'un menu pour l'utilisateur.

Exemple de menu :

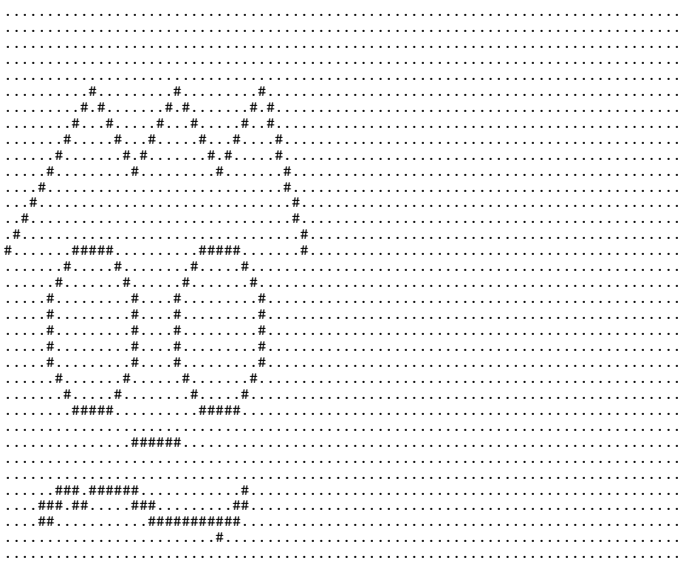
```
Veuillez choisir une action :
    A- Ajouter une forme
    B- Afficher la liste des formes
    C- Supprimer une forme
    D- Tracer le dessin
    E- Aide
    [Autres actions]
>> Votre choix : A
    Veuillez choisir une action :
        1- Ajouter un point
        2- Ajouter une ligne
        3- Ajouter cercle
        4- Ajouter un carre
        5- Ajouter un rectangle
        6- Ajouter un polygone
        7- Revenir au menu precedent
    >> Votre choix : 2
        Saisir les informations de la ligne :
        >> Saisir le premier point x1 y1 : 12 16
        >> Saisir le deuxieme point x2 y2 : 14 18
    >> Votre choix : 3
        Saisir les informations du cercle :
        >> Saisir le point centre x1 y1 : 24 10
        >> Saisir le rayon du cercle : 7
    >> Votre choix : 7
Veuillez choisir une action :
    A- Ajouter une forme
    B- Afficher la liste des formes
    C- Supprimer une forme
    D- Tracer le dessin
    E- Aide
    [Autres actions]
>> Votre choix : B
    Liste des formes :
        1 : CIRCLE 20 10 5
        2 : CIRCLE 20 25 5
        3 : LINE 5 20 10 25
        4 : POLYGON 15 0 5 10 10 15 5 20 10 25 5 30 15 35
        5 : CURVE 35 5 25 5 40 30 30 30
>> Votre choix : D
```

2.3. Génération du dessin à l'écran

Pour dessiner les différentes formes géométriques à l'écran, il existe des algorithmes de tracé pour chaque forme de base. Ceux-ci permettent de générer une approximation de la forme désirée sur la base de ses paramètres. Le détail de chacun de ces algorithmes est décrit dans la partie 2.

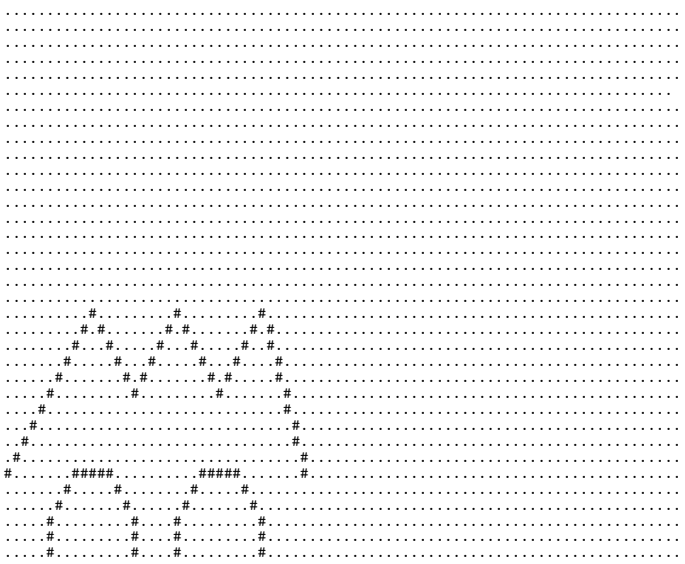
Exemple de dessin généré :

```
Veuillez choisir une action :
    A- Ajouter une forme
    B- Afficher la liste des formes
    C- Supprimer une forme
    D- Tracer le dessin
    E- Aide
    [Autres actions]
>> Votre choix : B
    Liste des formes :
        1 : CIRCLE 20 10 5
        2 : CIRCLE 20 25 5
        3 : LINE 5 20 10 25
        4 : POLYGON 15 0 5 10 10 15 5 20 10 25 5 30 15 35
        5 : CURVE 35 5 25 5 40 30 30 30
>> Votre choix : D
```



Il est à noter que les formes géométriques créées à partir de coordonnées se retrouvant partiellement ou totalement à l'extérieur de la zone de dessin sont acceptées. Celles-ci seront stockées en mémoire mais n'apparaiteront pas à l'écran.
Dans le cas où seulement une partie de la forme est à l'extérieur de la zone de dessin, il est nécessaire d'afficher à l'écran la portion du dessin se retrouvant à l'intérieur de la zone de dessin.

Exemple : Même figure dessinée avec des coordonnées différentes



Première partie
Structures de données des formes

Pour débuter ce travail, nous allons d'abord développer la partie du code de l'application permettant de créer et de stocker des formes spécifiques : point, segment, cercle, carré, rectangle et polygone. On ne s'intéressera donc pas à la notion de calques, de fenêtres ou du dessin des formes à l'écran pour le moment. Comme nous le verrons plus en détails à la section 4, nous allons devoir également créer un type abstrait (générique) pour la gestion de ces formes. Ce type générique sera indépendant d'une forme particulière afin de faciliter les développements qui suivront dans la deuxième partie.

Dans cette partie nous allons créer un module composé des fichiers `shapes.c` et `shapes.h`. Ce module va regrouper les types structurés de chaque forme ainsi que les fonctions associées : de création, d'affichage et de destruction. On fait référence dans cette partie aux fonctions d'affichage permettant d'afficher les informations liées à chacune des formes et non du tracé de ces formes à l'écran.

3. Les structures spécifiques des formes disponibles

3.1. Structure `Point`

Un point est une forme graphique qui représente une position dans l'espace. Notre application de dessin vectoriel étant textuelle nous utiliserons un caractère de remplissage pour le représenter dans notre espace à 2 dimensions. Dans la suite du document, nous prendrons comme caractère de remplissage le '#'.

La représentation en mémoire d'un point se fera par une variable de type structuré `Point` composé de deux champs entiers `pos_x` et `pos_y` représentant ses coordonnées sur les axes x et y.

```
typedef struct {
    int pos_x;
    int pos_y;
}Point;
```

Pour manipuler un point, les fonctions suivantes doivent être définies :

```
Point *create_point(int px, int py);
void delete_point(Point * point);
void print_point(Point * p);
```

Où :

- La fonction `Point *create_point(int px, int py)` permet d'allouer dynamiquement une variable de type structuré `Point` dont les coordonnées sont données en paramètre.
- La fonction `void delete_point(Point * point)` permet de libérer la mémoire allouée au point donné en paramètre.
- La fonction `void print_point(Point * p)` permet d'afficher à l'écran les informations d'un `Point` sous la forme suivante : `POINT [p(pos_x, pos_y)]`

Exemple d'utilisation

```
Point * p = create_point (10, 15);
print_point(p);
delete_point(p);
```

Résultat à l'écran

```
POINT 10 15
```

3.2. Structure `Line`

Une ligne est une forme géométrique correspondant à un segment délimité par deux points. Elle est représentée en mémoire comme une structure composée de deux champs de type `Point` représentant les extrémités du segment. Les variables de type `Point` étant créées dynamiquement, les champs de la structure devront être de type `Point*`.

```
typedef struct line {
    Point *p1;
    Point *p2;
}Line;
```

Les fonctions associées à ce type structuré sont les suivantes :

```
Line *create_line(Point * p1, Point * p2);
void delete_line(Line * line);
void print_line(Line * line);
```

Où :

- La fonction `Line *create_line(Point * p1, Point * p2)` permet d'allouer dynamiquement un segment de type structuré `Line` à partir de deux points donnés en paramètres.
- La fonction `void delete_line(Line * line)` permet de libérer la mémoire allouée au segment donné en paramètre.
- `void print_line(Line * line)` permet d'afficher les informations d'un segment selon le format décrit dans l'exemple ci-après : `LINE [x1, y1, x2, y2]`

Exemple d'utilisation

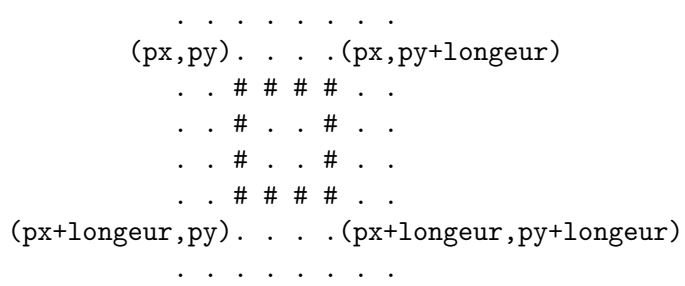
```
Point * p1 = create_point (10, 15);
Point * p2 = create_point (21, 25);
Line * l = create_line (p1 ,p2);
print_line (l);
delete_line(l);
delete_point(p1);
delete_point(p2);
```

Résultat à l'écran

```
LINE 10 15 21 25
```

3.3. Structure `Square`

La forme carré est représentée par un point dans l'espace (coin supérieur gauche) et une longueur. À partir de ces informations vous définirez vous même la structure `Square`. Les positions des 3 autres coins peuvent être calculées comme suit :

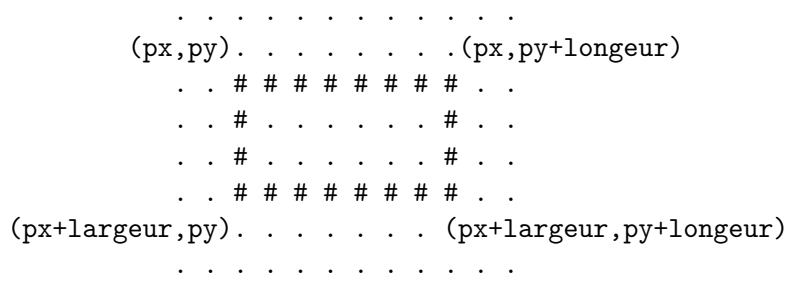


Un carré est donc représenté en mémoire comme une structure ayant deux champs : un point et une longueur. De la même façon que pour les structures précédentes, il est nécessaire d'implémenter trois fonctions associées à la manipulation de ce type structuré :

```
Square *create_square(Point * point, int length);
void delete_square(Square * square);
void print_square(Square * square);
```

3.4. Structure `Rectangle`

La forme rectangle est représentée par un point (coin supérieur gauche), une longueur et une largeur. À partir de ces informations vous définirez vous même la structure `Rectangle`. Les positions des 3 autres coins peuvent être calculées comme suit :



Tout comme pour les structures précédentes, il est nécessaire d'implémenter trois fonctions pour la manipulation de ce type structuré :

```
Rectangle *create_rectangle(Point * point, int width, int height);
void delete_rectangle(Rectangle * rectangle);
void print_rectangle(Rectangle * rectangle);
```

3.5. Structure `Circle`

La forme cercle est représentée par un point et un rayon. À partir de ces informations vous définirez vous même la structure `Circle`. De la même façon que pour les structures précédentes, il est nécessaire d'implémenter trois fonctions associées à la manipulation de ce type structuré :

```
Circle *create_circle(Point * center, int radius);
void delete_circle(Circle * circle);
void print_circle(Circle * circle);
```

3.6. Structure `Polygon`

Le polygone est représenté comme un ensemble de points à relier. Le type structuré de cette forme est constitué de deux champs : un tableau dynamique de points et sa taille `n`. Le tableau de points doit être dynamique (taille logique égale à la taille physique) car la taille `n` n'est pas connue à la compilation, c'est l'utilisateur qui le précisera avec la commande de création d'un polygone.

Note : Il est important de rappeler que dans ce projet les variables de type *Point* sont créés dynamiquement par la fonction `create_point` qui retourne donc une variable de type *Point**. Le deuxième champ de la structure `Polygon` doit être déclaré comme un tableau 1D dynamique dont chaque case est de type `Point*`. Ce champ est donc de type *Point***.

NB : Pour avoir un polygone fermé, il est nécessaire de que les coordonnées de son premier point soient les mêmes que celles du dernier point.

La structure d'un polygone est la suivante :

```
typedef struct polygon {
    int n;
    Point ** points; // tableau 1D dynamique de Point*
}Polygon;
```

Les fonctions qui lui sont associées sont encore une fois similaires à ce qui a été fait pour les précédentes structures :

```
Polygon *create_polygon(int n, int** tab);
void delete_polygon(Polygon * polygon);
void print_polygon(Polygon * polygon);
```

4. Structure générique `Shape`

Nous avons maintenant à disposition un ensemble de fonctions spécifiques pour gérer chacune de nos formes. Cependant le système de calque présenté dans la partie 1.1 à la page 5 doit permettre de regrouper des formes de type différent. De plus, chaque forme doit pouvoir être identifiée de manière unique dans l'application. Ce numéro doit être indépendant de la forme mais aussi du calque auquel la forme appartient car l'utilisateur doit avoir la possibilité de faire basculer une forme d'un calque à un autre.

L'application doit donc être en mesure de manipuler un type abstrait permettant de représenter un type de forme quelconque : *Point*, *Line*, *Square*, etc. C'est ce qu'on appelle la **généricité** en programmation. En langage C cette généricité est obtenue avec un type particulier de pointeur : `void *`.

Compléments d'information sur le type `void*`

Pour rappel, un pointeur est une variable permettant de stocker l'adresse d'une autre variable du programme. Le pointeur pointe ou référence une autre variable. L'opérateur de déréférencement "*" appliqué sur un pointeur permet de récupérer la valeur de la variable référencée. Cet opérateur de déréférencement ne fonctionne que si le type de la variable référencée est connue : c'est pour cette raison qu'il est obligatoire de préciser le type de la variable référencée lors de la déclaration d'un pointeur.

Le type `void*` est un type de pointeur dit **générique** : il permet de stocker l'adresse d'une variable dont le type n'est pas connue. **L'opérateur de déréférencement sur ce type de pointeur est illégale** car la machine ne sait pas combien d'octets lire à cette adresse pour récupérer la valeur, ni la convention de codage binaire utilisée par ailleurs (cf. module de système numérique !).

Vous avez déjà utilisé des fonctions génériques utilisant le type `void*` : `printf` et `malloc`.

La fonction `malloc` permet d'allouer dynamiquement n'importe quel type de variable ou de tableau. Il n'existe qu'une seule version de la fonction `malloc` quelque soit le type de ce qui est alloué (variable de type `int`, tableau 1D de `char`, tableau 2D de `float`, etc). La généricité de cette fonction est obtenue par le type de la valeur de retour qui est `void*`. La fonction renvoie ainsi l'adresse d'une zone mémoire de n'importe quel type. Comme le savez, cette valeur de retour doit être **castée** (implicitement ou explicitement par le programmeur) vers un nouveau pointeur du type attendu pour être utilisée dans la suite du code.

La fonction `printf` permet avec le format `%p` d'afficher n'importe quel type d'adresse. Dans ce cas de figure, la généricité de la fonction est obtenue en utilisant un paramètre de type `void*`. Lors de l'appel à la fonction, le pointeur ou l'adresse spécifiée en argument est `casté` implicitement vers un `void*` afin d'être afficher.

Ce qu'il faut retenir sur les pointeurs de type `void*` :

- ce type de pointeur permet de pointer sur n'importe quel type de variable (types de base, structurés, etc) ;
- ce type est utilisé pour rendre des fonctions génériques du point de vue de leur entrées et/ou sortie, c'est à dire indépendante d'un type particulier.
- L'arithmétique des pointeurs n'existe pas sur ce type de pointeur. Il est obligatoire de `caster` le pointeur (implicitement ou explicitement) vers le type souhaité pour qu'une opération de déréférencement soit possible par exemple.
- Attention à l'ambiguïté du mot clé `void` l'expression `void*`. Un pointeur de ce type ne pointe pas vers une variable de type `void` et cette expression ne doit pas non plus être interprétée comme un pointeur vers "rien".

4.1. Type structuré `Shape`

Pour gérer cette généricité pour les formes nous allons créer un type structuré `Shape` utilisant le type particulier `void*` pour le champ permettant de pointer vers n'importe quel type de structure correspond aux formes que nous avons définie dans la section précédente. Un champ supplémentaire représentera le numéro d'identification (nommé `ID` dans la suite du document) afin d'identifier de manière unique la forme dans l'application.

Note : Ce type générique donne la possibilité d'ajouter une nouvelle forme à notre application en limitant l'impact sur le code déjà écrit. Cette type d'avantage correspond à l'indicateur appelé maintenabilité pour la qualité logiciel.

Ainsi, notre nouveau type structuré Shape sera composé d'un identifiant (id), un pointeur générique vers n'importe quelle forme ainsi que le type de la forme pointée (`SHAPE_TYPE shape_type`).

Étant donné qu'il existe une liste limitée de formes géométriques, nous allons utiliser une énumération pour leurs types.

```
typedef enum { POINT, LINE, SQUARE, RECTANGLE, CIRCLE, POLYGON} SHAPE_TYPE;

typedef struct shape {
    int id;                 // identifiant unique de la forme
    SHAPE_TYPE shape_type;  // type de la forme pointé
    void *ptrShape;         // pointeur sur n'importe quelle forme
}Shape;
```

Afin de manipuler ces formes de façon générique, nous devons implémenter les fonctions suivantes :

```
Shape *create_empty_shape(SHAPE_TYPE shape_type);
Shape *create_point_shape(int px, int py);
Shape *create_line_shape(int px1, int py1, int px2, int py2);
Shape *create_square_shape(int px, int py, int length);
Shape *create_rectangle_shape(int px, int py, int width, int height);
Shape *create_circle_shape(int px, int py, int radus);
Shape *create_polygon_shape(int lst[], int n);
void delete_shape(Shape * shape);
void print_shape(Shape * shape);
```

L'idée est que la fonction `Shape *create_empty_shape(SHAPE_TYPE shape_type)` alloue la zone mémoire qui contiendra un type de forme donné en paramètre. Pour ce faire, nous proposons de l'écrire comme suit :

```
Shape *create_empty_shape(SHAPE_TYPE shape_type) {
    Shape *shp = (Shape *) malloc(sizeof(Shape));
    shp->ptrShape = NULL;
    shp->id = 1; // plus tard on appelera get_next_id();
    shp->shape_type = shape_type;
    return shp;
}
```

Puis, créer une fonction générique pour chaque type de forme. Nous vous proposons le code de la fonction `Shape *create_empty_shape(SHAPE_TYPE shape_type)` ci-dessous :

```
Shape *create_point_shape(int px, int py) {
    Shape *shp    = create_empty_shape(POINT);
    Point *p      = create_point(px, py);
    shp->ptrShape = p;
    return shp;
}
```

Il est donc demandé de comprendre et recopier les deux fonctions fournies puis de déduire le code des fonctions suivantes :

— `Shape *create_line_shape(int px1, int py1, int px2, int py2)` permet de de créer une forme `Shape` vide, puis lui associe une forme de type ligne ;

— La fonction `Shape *create_square_shape(int px, int py, int length)` permet de créer une formr `Shape` vide, puis lui associe une forme de type carré ;

— La fonction `Shape *create_rectangle_shape(int px, int py, int width, int height)` permet de créer une forme `Shape` vide, puis lui associe une forme de type rectangle ;

— La fonction `Shape *create_cercle_shape(int px, int py, int radus)` permet de créer une forme `Shape` vide, puis lui associe une forme de type cercle

— La fonction `Shape *create_polygon_shape(int lst[], int n)` permet de créer une forme `Shape` vide, puis lui associe une forme de type polygone. Cette fonction doit aussi vérifier que le nombre de points n est un multiple de deux, car le tableau doit contenir un nombre pair d'éléments ;

— La fonction `void delete_shape(Shape * shape)` permet de supprimer une forme.

— La fonction `void print_shape(Shape * shape)` permet d'afficher à l'écran l'id de la *shape* son type puis appellera la fonction `print_...` de la forme encapsulée (voir l'exemple ci-après).

Exemple d'utilisation

```
Shape * f1 = create_line_shape (10, 15, 21, 25);
print_shape (f1);
delete_shape (f1);
```

Résultat à l'écran

```
LINE 10 15 21 25
```

4.2. Gestion des numéros uniques

Les formes (*Shapes*) sont identifiées par un numéro unique. Pour garantir cette unicité, la méthode à utiliser est la suivante :

1. Créer une variable globale **global_id** initialisée à 0
2. Incrémenter la valeur de cette variable à chaque création d'une forme géométrique. Elle est donc modifiée dans toutes les fonctions de création des formes quelques soient leurs types.

Nous vous proposons de le faire comme suit dans les fichiers `id.h` et `id.c`.

```
unsigned int global_id = 0;
unsigned int get_next_id();
```

— La fonction `unsigned int get_next_id()` permet d'incrémenter le compteur et de retourner sa dernière valeur. Notez que la première forme créée aura pour **id** la valeur 1.

Deuxième partie
Affichage de l'image à l'écran

5. Structures de données pour l'affichage

5.1. Structure `Area`

L'écran d'un ordinateur est composé de pixels, l'origine `(0,0)` se situe en haut à gauche de l'écran. Dans ce projet nous souhaitons garder cette disposition. C'est à dire l'axe X est sur la verticale et l'axe Y et sur l'horizontale, comme le montre la figure 6.

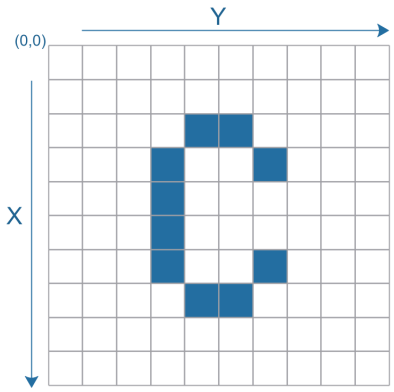


FIGURE 6 – Écran

Cette zone de dessin est vue comme étant une matrice (un tableau à 2D) où chaque case est considérée comme un pixel. Par ailleurs, dans une zone de dessin, il est possible de dessiner plusieurs formes géométriques. Ainsi, pour représenter ce comportement, nous allons utiliser la structure `Area` qui est à définir dans le fichier `area.h` et `area.c`. Cette structure doit contenir tous les pixels à afficher ainsi que leurs états (vide, rempli). La structure doit contenir aussi toutes les formes (tableau de pointeurs sur `Shape`) qui sont dessinées dans cette `area` (pour l'instant notre programme est composé d'une seule zone de dessin).

```
#define SHAPE_MAX 100   // Nombre maximum de formes
#define BOOL int

struct area {
    unsigned int width;  // Nombre de pixels en largeur ou nombre de colonnes (axe y)
    unsigned int height; // Nombre de pixels en hauteur ou nombre de lignes (axe x)
    BOOL** mat;          // Matrice des pixels de taille (width * height)
    Shape* shapes[SHAPE_MAX]; // Tableau des formes;
    int nb_shape;      // Nombre de formes dans le tableau shapes (taille logique)
};
typedef struct area Area;
```

Les deux variables `width` et `height` représentent la taille de la zone d'affichage réel à l'écran. La matrice `mat` est une tableau 2D de `int`, chaque valeur peut contenir soit `0` pour designer un pixel vide ou `1` pour dire que le pixel est plein est noir.

Pour gérer, la zone de dessin, il est demandé de définir les fonctions suivantes :

```
Area* create_area(unsigned int width, unsigned int height);
void add_shape_to_area(Area* area, Shape* shape);
void clear_area(Area* area);
void erase_area(Area* area);
void delete_area(Area* area);
void draw_area(Area* area);
void print_area(Area* area);
```

Où :

- La fonction `Area* create_area(unsigned int width, unsigned int height)` permet de créer une zone de type `area` et retourne un pointeur vers celle-ci. Elle doit aussi allouer la mémoire nécessaire pour stocker le tableau des pixels à afficher à l'écran.
- La fonction `void add_shape_to_area(Area* area, Shape* shape)` permet d'ajouter une forme à la zone de dessin, c'est à dire insérer dans le tableau des formes la forme donnée en paramètre et incrémenter `nb_shape`.
- La fonction `void clear_area(Area* area)` permet d'initialiser tous les pixels à `0`.
- La fonction `void erase_area(Area* area)` permet de supprimer toutes les formes dessinées dans cette zone de dessin. C'est à dire supprimer (et ne pas oublier de libérer la mémoire) toutes les formes qui sont dans le tableau `shapes`.
- La fonction `void delete_area(Area* area)` supprime une zone de dessin avec l'ensemble des *formes* associées.
- La fonction `void draw_area(Area* area)` permet de dessiner une forme dans la zone de dessin. Son objectif est de parcourir toutes les formes *shapes*. Puis, pour chaque forme de type *shape*, elle doit appeler la fonction `Pixel** create_shape_to_pixel(Shape* shape)` (qui sera décrite plus loin dans ce document) qui doit nous renvoyer la liste des pixels à colorier. Enfin, cette liste de pixels est parcourue et les valeurs à 1 sont affectées aux bons pixels dans la `mat` ;
- La fonction `void print_area(Area* area)` permet d'afficher à l'écran les pixels de la matrice `mat` à l'aide de la fonction `printf()`. L'affichage se fera en utilisant des caractères ASCII. Par exemple : utiliser le caractère '.' pour un pixel vide et le caractère '#' pour un pixel rempli.

Exemple d'utilisation :

```
Area * draw_zone = create_area(12, 14);
Shape * shape1   = create_line_shape(5, 5 , 10, 10);
Shape * shape2   = create_cercle_shape(5, 5, 4);
add_shape_to_area (draw_zone, shape1);
add_shape_to_area (draw_zone, shape2);
draw_area   (draw_zone);
print_area  (draw_zone);
erase_area  (draw_zone);
draw_area   (draw_zone);
print_area  (draw_zone);
delete_area (draw_zone);
```

```
1er appel a draw_area
.............
...#####.....
..##...##....
.##.....##...
.#.......#...
.#...#...#...
.#....#..#...
.##....###...
..##...##....
...#####.#...
..........#..
.............
```

```
2eme appel a draw_area
..........
..........
..........
..........
..........
..........
..........
..........
..........
..........
..........
..........
```

5.2. Structure `Pixel`

Pour chaque forme que nous avons créée dans la partie précédente il nous faut la transformer en pixels et l'afficher à l'écran. C'est à dire calculer les points à remplir dans la matrice 2D décrite dans le début de la partie II.

Pour réaliser cette partie nous proposons de procéder de la manière suivante :

1. Initialiser la matrice 2D à 0
2. Pour chaque forme :
   a) Calculer les positions (coordonnées des cases dans la matrice) des pixels à remplir selon le type de la forme
   b) Remplacer les 0 par des 1 dans la matrice aux positions retournées dans l'étape précédente.
3. Parcourir la matrice, pour chaque case :
   — si la case = 0 alors afficher `'.'`
   — si la case = 1 alors afficher '#'

Pour ce faire, il faut définir une structure `Pixel` avec les fonctions ci-dessous :

```
struct pixel {
    int px;
    int py;
};
typedef struct pixel Pixel;
Pixel *create_pixel(int px, int py);
void delete_pixel(Pixel * pixel);
```

Où :

— La fonction `Pixel* create_pixel(int px, int py)` permet d'allouer un espace mémoire de type `Pixel`, de l'initialiser et de retourner l'adresse de l'espace mémoire alloué.
— La fonction `void delete_pixel(Pixel* pixel)` permet de libérer l'espace mémoire alloué par la fonction précédente.

6. Tracer les formes

Afin de calculer la position des pixels à remplir dans la matrice (étape 2.a), nous allons utiliser des algorithmes spécialisés pour chaque type de forme. Ces algorithmes vont nous retourner la liste des pixels à mettre à 1. L'idée est d'implémenter le modèle de fonction ci-dessous pour chaque type forme :

```
void pixel_forme(Shape* shape, Pixel** pixel, int* nb_pixels).
```

Où : `forme` $\in \{point, line, cercle, ...\}$.

Celle-ci prend en paramètres :

- Un pointeur sur la forme à dessiner ;
- Un tableau de pointeurs sur des pixels car la fonction qui permet de créer un pixel (Cf. section 5.2) retourne l'adresse de l'espace mémoire alloué pour un pixel (un pointeur), d'où `Pixel ** pixel`) ;
- Un pointeur sur un entier représentant la taille logique du tableau de pixels et qui se met à jour à chaque fois qu'un pixel est ajouté au tableau.

6.1. Tracer un point

La fonction `pixel_point` est la fonction la plus simple à écrire. Elle permet de stocker dans le tableau des pixels, la position du point à tracer. Elle correspond à la fonctionnalité que nous avons proposée précédemment : `POINT 10 15`.
On pourrait imaginer ici une fonction qui alloue seulement un `Pixel`, lui attribue les positions `px` et `py` et retourne un pointeur sur ce pixel. Néanmoins, afin de pouvoir utiliser les fonctions de calcul de pixels de façon générique, il faut qu'elles aient toutes le même prototype. C'est pour cela que cette fonction va créer un tableau de pixels d'une seule case et va placer son seul pixel dans cette case. La taille du tableau sera ainsi égale à 1. Ci-après le prototype de cette fonction :

```
void pixel_point(Point* shape, Pixel** pixel, int* nb_pixels);
```

Pour vous aider à implémenter cette fonction et vous donner une idée d'implantation des fonctions suivantes, nous vous proposons ici une solution :

```
void pixel_point(Shape* shape, Pixel** pixel, int* nb_pixels)
{
    Point* pt    = (Point*) shape->ptrShape;
    pixel_tab    = (Pixel**) malloc (sizeof (Pixel*));
    pixel_tab[0] = create_pixel(pt->pos_x, pt->pos_y);
    *nb_pixels   = 1;
}
```

6.2. Tracer ligne

La fonction dont le prototype est donnée ci-après, permet de calculer la position des pixels à remplir pour tracer une ligne sans trous. Par exemple la fonctionnalité proposée dans application `LINE 2 2 12 18` permettra de tracer la ligne suivante :

```
....................
....................
..#.................
...##...............
.....#..............
......##............
........##..........
..........#.........
............##......
..............#.....
..............##....
................##...
..................#..
```

```
void pixel_line(Line* line, Pixel** pixel, int* nb_pixels);
```

Pour calculer la positon des pixels que va occuper une ligne, on pourrait utiliser la fonction :

$$f(x) = ax + b$$

Cependant, l'écran d'un ordinateur est discret ($\neq$ continu), utiliser directement cette méthode peut engendre des lignes avec des trous notamment lorsque la ligne est diagonale. Pour tracer une ligne sans trous sur un écran discret, il existe de nombreux algorithmes de tracé de segment dont l'algorithme de Bresenham. Afin de simplifier le tracé, nous allons utiliser l'algorithme de Nicolas Flasque (Enseignant à Efrei Paris) qui se base sur une méthode utilisant uniquement des nombres entiers et non pas des flottants.

L'idée globale est que pour calculer la position des pixels que va occuper une ligne, nous prenons en paramètres les coordonnées de début ($x1$,$y1$) et les coordonnées de fin ($x2$,$y2$). Les pixels de la ligne sont calculés en effectuant des décalages entre les deux positions début et fin : (dx,dy).

6.3. Tracer un Cercle

La fonctions *pixel_ circle* dont le prototype est donné ci-après, permet de calculer la position des pixels à remplir pour tracer un cercle. Par exemple la fonctionnalité proposée précédemment : `CIRCLE 4 4 3` permettra de tracer le cercle suivant :

```
void pixel_circle(Cercle* shape, Pixel** pixel, int *nb_pixels);
```

```
..........
...###....
..#...#...
.#.....#..
.#.....#..
.#.....#..
..#...#...
...###....
..........
```

Pour calculer tous les pixels à générer pour dessiner un cercle, il est demandé d'utiliser l'algorithme de tracé de cercle d'Andres. Cet algorithme est aussi connu sous le nom de Algorithme de tracé d'arc de cercle de Bresenham. Il est plus efficace que d'autres algorithmes de tracé de cercles, car il utilise uniquement des opérations entières (pas de virgules flottantes) avec moins de calculs.

Pour tracer un cercle, l'idée est de générer les pixels d'un seul octant (un huitième $\frac{1}{8}$), et de déduire le reste par symétrie comme illustré dans la figure 7. Grâce à cela, le nombre de calculs à faire est réduit.

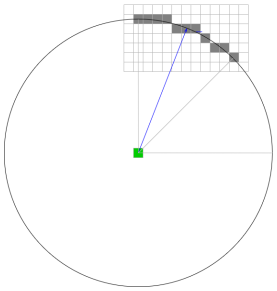


FIGURE 7 – Cercle dans un seul octant

L'algorithme calcule les points du cercle en utilisant uniquement des valeurs entières. Ainsi, les points ne sont pas toujours exactement sur le cercle mais le plus proche possible. L'algorithme suit un processus de décision pour déterminer le point suivant à tracer, en utilisant une évaluation de la distance entre le dernier point tracé et le cercle réel pour déterminer si le point suivant doit être en diagonale, horizontal ou vertical. Les étapes suivies sont donc comme suit :

Pour un point P de centre p_x et p_y et de rayon *rayon* :

1. Initialiser deux variables x et y (coordonnées du cercle) à (0,rayon) ;
2. Initialiser l'erreur de tracé delta (d dans l'algorithme décrit ci-après) à (1-rayon) ;
3. Répéter jusqu'à ce que x dépasse y :
   a) Dessiner huit points symétriques autour du centre du cercle, en ajoutant le pixel (x,y) à chaque étape au tableau des pixels. Cela dessine les huit octants du cercle simultanément. Les coordonnées de ces 8 points selon l'algorithme de tracé de cercle d'Andres sont les suivantes :
      — Point1 (p_x + x, p_y + y)
      — Point2 (p_x + y, p_y + x)
      — Point3 (p_x - x, p_y + y)
      — Point4 (p_x - y, p_y + x)
      — Point5 (p_x + x, p_y - y)
      — Point6 (p_x + y, p_y - x)
      — Point7 (p_x - x, p_y - y)
      — Point8 (p_x - y, p_y - x)
   b) Si l'erreur de tracé *delta* est négative, on incrémente x et on met à jour l'erreur de tracé en ajoutant $2x + 1$.
   c) Sinon, on décrémente y, on incrémente x et on met à jour l'erreur de tracé en ajoutant $2x - 2y + 2$.
4. Lorsque x devient supérieur à y, cela signifie que tous les points du cercle ont été ajoutés au tableau des pixels.

Le pseudo algorithme décrivant cette génération de points est comme suit :

Data: cercle, tab[], *k
Result: liste des pixels dans le tableau tab[], k le nombre d'éléments
Function `pixel_cercle(`*Shape *cercle, int tab[], int k*`) :`

$x \leftarrow 0$;
$y \leftarrow cercle \rightarrow radus$;
$d \leftarrow cercle \rightarrow radus - 1$;
$k \leftarrow 0$ `// nombre d'éléments dans le tableau de pixels tab`
Pixel *px $\leftarrow$ *null*
while $y \geq x$ **do**
    `// Ajouter le point pour le premier octant`
    $px = create\_pixel(cercle \rightarrow center \rightarrow pos\_x + x, cercle \rightarrow center \rightarrow pos\_y + y);$
    $tab[k++] \leftarrow px$;
    `// Ajouter le point pour l'octant d'en face`
    $px = create\_pixel(cercle \rightarrow center \rightarrow pos\_x + y, cercle \rightarrow center \rightarrow pos\_y + x);;$
    $tab[k++] \leftarrow px;$
    `// Ajouter la même choses pour les six octants restant`
    `// ....`
    if $d \geq 2 * x$ **then**
        $d- = 2 * x + 1;;$
        $x++;;$
    end
    else if $d2 * (cercle \rightarrow radus - y)$ **then**
        $d+ = 2 * y - 1$;
        $y--;$
    end
    else
        $d+ = 2 * (y - x - 1);;$
        $y--;;$
        $x++;;$
    end
end

6.4. Tracer carré, rectangle et polygone

Pour calculer la position des pixels qu'occupe un carré, il suffit tout simplement de calculer la position des pixels des 4 segments, en utilisant le même procédé utilisé dans la fonction `pixel_line`.

Nous rappelons grâce au schéma ci-après la méthode de calcul des positions des 4 segments.

```
                . . . . . . . . .
          (px,py). . . .(px,py+longeur)
                . . # # # # . .
                . . #   .   # . .
                . . #   .   # . .
                . . # # # # . .
(px+longeur,py). . . .(px+longeur,py+longeur)
                . . . . . . . . .
```

Pour calculer les pixels d'un rectangle, le calcul est identique au calcul précédent. Pour les polygones la

méthode est encore plus simple. Il suffit de relier l'ensemble des points par des segments. Le prototype de ces fonctions est identique aux fonctions précédentes et qui sont exprimés comme suit :

```
void pixel_square(Square* square, Pixel** pixel, int* nb_pixels);
void pixel_rectangle(Rectangle* rectangle, Pixel** pixel, int* nb_pixels);
void pixel_polygon(Polygon* polygon, Pixel** pixel, int* nb_pixels);
```

7. Transformer une forme quelconque en un ensemble de pixels

Les fonctions que nous venons d'écrire permettent de tracer un ensemble de pixels (stockés dans un tableau dans notre cas) d'une forme donnée. Nous souhaitons généraliser cela et écrire une fonction qui prend en paramètre une forme quelconque de type `Shape` et qui génère l'ensemble (un tableau) de pixels la constituant. Nous proposons de l'écrire sous le prototype ci-après où elle prend en paramètre une forme *shape* et retourne en sortie un tableau de pixels. La taille du tableau `nb_pixels` est un paramètre modifiable.

```
Pixel** create_shape_to_pixel(Shape * shape, int* nb_pixels);
```

Bien sûr, il ne faut pas oublier de libérer la mémoire des pixels alloués

```
void delete_pixel_shape(Pixel** pixel, int nb_pixels);
```

Troisième partie
Gestion des commandes

Nous allons devoir créer des commandes utilisateurs spécifiques permettant de manipuler les formes géométriques afin de les tracer dans la zone de dessin de l'application. Le rafraîchissement de l'affichage de la zone de dessin se fait de manière automatique après l'ajout d'une forme.

Celles-ci vont indiquer des ordres d'exécution à donner à l'application. Chaque commande doit être de la forme : <**nom commande**> **[paramètres]**. La présence des paramètres est optionnelle car cela dépend du type de l'ordre à effectuer.

Dans ce projet, la liste des commandes que le programme doit interpréter est la suivante :

- `clear` : effacer l'écran
- `exit` : quitter le programme
- `point x y` : ajouter un point
- `line x1 y1 x2 y2` : ajouter un segment reliant deux points (x1, y1) et (x2, y2)
- `circle x y radius` : ajouter un cercle de centre (x, y) et de rayon `radius`
- `square x y length` : ajouter un carré dont le coin supérieur gauche est (x, y) et de côté `length`.
- `rectangle x y width height` : ajouter un rectangle dont le coin supérieur gauche est (x, y), de largeur `width` et de longueur `height`
- `polygon x1 y1 x2 y2 x3 y3 ... ...` : ajouter un polygone avec la liste des points donnés
- `plot` : rafraîchir l'écran pour afficher toutes les formes géométriques de l'image (en fonction des règles d'affichage)
- `list` : afficher la liste de l'ensemble des formes géométriques qui composent l'image ainsi que toutes leurs informations
- `delete id` : supprimer une forme à partir de son identifiant `id`.
- `erase` : supprimer toutes les formes d'une image.
- `help` : afficher la liste des commandes ainsi qu'un mini mode d'emploi permettant à l'utilisateur d'utiliser les commandes correctement.

8. Les commandes utilisateur

Pour gérer la ligne de commande on va utiliser une boucle infinie qui va repérer ces étapes :

1. Afficher un prompt (par exemple `"»"`).
2. Lire l'entrée de l'utilisateur en utilisant la fonction `fgets()` de la *libc*.
3. Analyser la commande et remplir la structure `struct command`.
4. Vérifier si la commande est correcte.
5. Exécuter la commande de l'utilisateur.
6. Revenir à l'étape 1.

9. Structure d'une commande

Les commandes utilisateur sont sous la forme suivante :

- Commencent toujours par une chaîne de caractère qui est le nom de la commande.
- Peut être suivie par :
  - Des entiers
  - Une autre chaîne suivie par :
    - Des entiers
    - Une autre chaîne suivie par des entier

Donc on peut déduire d'une commande est un ensemble de chaînes de caractères suivi par un ensemble d'entiers. Voici quelques exemples :

```
>> line 10 12 30 45
>> delete 10
>> help
>> list
>> liste layers
>> select layer 12
>> plot
```

Pour simplifier l'analyse de la ligne de commande, on va tout d'abord lire la chaîne de caractère saisie par l'utilisateur à l'aide `fgets()` puis on sépare cette chaînes en un ensemble de chaînes en utilisant le caractère `' '` (espace) comme séparateur.

Chaque token[1] sera placé dans la structure `struct command`. Le premier token étant le nom de la commande va être stocké dans le champs `name`. Pour le reste, si le token est un entier, alors on le rajoute dans le tableau `int_params`, si c'est une chaîne alors on la rajoute dans le tableau `str_params`. Les entiers `int_size` et `str_size` représentent respectivement le nombre d'éléments dans les deux tableaux `int_params` et `str_params`. Voir l'exemple à la figure 8.

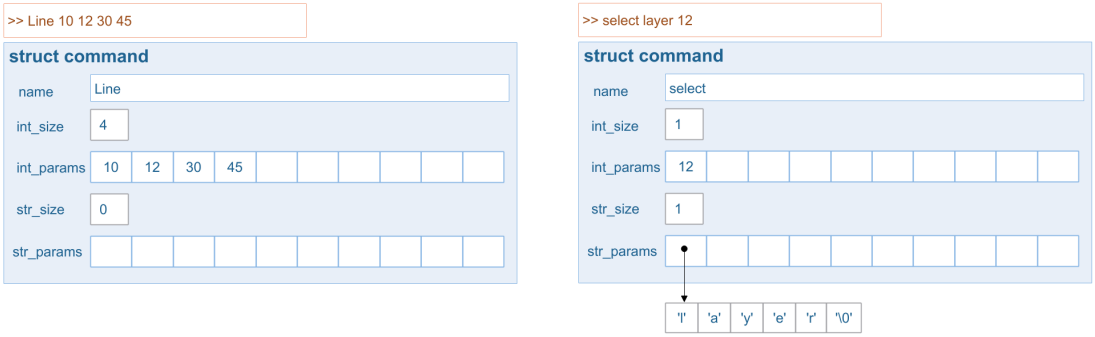


FIGURE 8 – Exemple de création d'un objet commande à partir d'une commande utilisateur saisie au clavier.

```
struct command {
    char name[50];
    int int_size;
    int int_params[10];
```

1. pour désigner un mot ou un entier dans notre cas.

```
    int str_size;
    char* str_params[10];
};
typedef struct command Command;

Command* create_commande();
void add_str_param(Command* cmd, char* p);
void add_int_param(Command* cmd, int p);
void free_cmd(Command* cmd);
int read_exec_command(Command* cmd);
void read_from_stdin(Command* cmd);
```

— La fonction `Command* create_commande()` permet de créer une variable de type *command*.

— La fonction `void add_str_param(Command* cmd, char* p)` permet d'insérer dans le tableau `str_params` la chaîne de caractère donnée en paramètre et incrément *str_ size* ;

— La fonction `void add_int_param(Command* cmd, int p)` permet d'insérer dans le tableau `int_params` l'entier donné en paramètre et incrémente *int_ size* ;

— La fonction `void free_cmd(Command* cmd)` permet de supprimer une *command* donné en paramètre et libérer l'espaces occupé par les chaînes de caractères dans le tableau `str_params`.

— La fonction `int read_exec_command(Command* cmd)` permet d'exécuter la commande saisie par l'utilisateur.

— La fonction `void read_from_stdin(Command* cmd)` demande à l'utilisateur de saisir une ligne au clavier, puis parse la ligne du paramètre `cmd`.

Quatrième partie
Fonctionnalités avancées

Il est possible d'aller encore plus loin dans le projet en développant des fonctionnalités supplémentaires, à savoir :

— Courbe de Bézier
— Système de calques
— Utiliser les listes chaînées au lieu des tableaux

10. Courbe de Bézier

Un ingénieur chez Renault, Pierre Bézier, inventa en 1962 une méthode pour tracer des des courbes lisses à l'aide d'un des outils informatiques. Cet algorithme appelé **algorithme de Bézier** un moyen efficace et précis de dessiner des courbes en 2D.

L'algorithme de Bézier utilise un ensemble de points de contrôle pour définir une courbe. Comme vous pouvez le voir dans les figures 9 et 10.

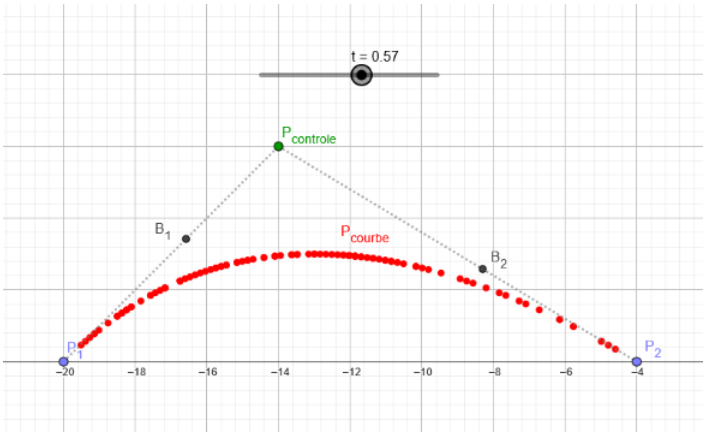


FIGURE 9 – idée de screen pour les explications - 1 point de contrôle (fait par Fabien avec l'appli Geogebra).

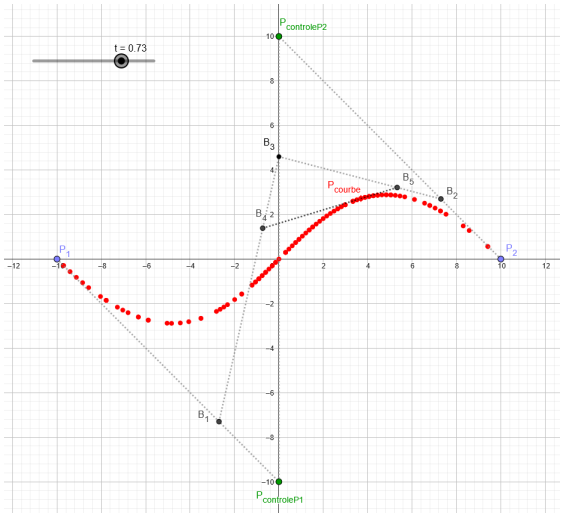


FIGURE 10 – idée de screen pour les explications - 2 point de contrôle (fait par Fabien par l'appli Geogebra).

10.1. Algorithme de Bézier

1. Définir deux point comme étant des points de contrôle. Dans la pratique il est possible d'utiliser n'importe quel nombre de points de contrôle >= 1. Cependant, dans ce projet, on utilisera uniquement deux points de contrôle.
2. Déterminer les coefficients des fonctions pour chaque point de contrôle comme suit

$$B(i,n,t) = (\frac{n!}{i! \times (n-i)!} \times 1 - t^{n-i} * t^{i})$$

   — i étant l'indice du point de contrôle, n est le nombre total de points de contrôle -1, et t est un paramètre compris entre 0 et 1.

3. Pour chaque point de la courbe, on calcule la somme pondérée des points de contrôle en utilisant les coefficients des fonctions précédents. La formule pour calculer un point de la courbe est la suivante :

$$P(t) = \sum_{i=0}^{n} B(i,n,t) * P_i$$

   — P_i est le i-ème point de contrôle et n est le nombre total de points de contrôle -1.

4. Finalement, tracez la courbe en reliant les points calculés dans l'étape précédente.

10.2. Algorithme de Casteljau

L'algorithme de Casteljau est une méthode alternative pour calculer une courbe de Bézier. Il utilise une méthode de subdivision récursive pour calculer les points de la courbe de Bézier plutôt que de calculer directement les coefficients des fonctions de base de Bézier.

L'algorithme de Casteljau fonctionne de la manière suivante :

1. Définir un ensemble de points de contrôle.
2. Définir un paramètre t compris entre 0 et 1.
3. À l'étape initiale, on trace une ligne droite reliant chaque paire de points de contrôle adjacents.

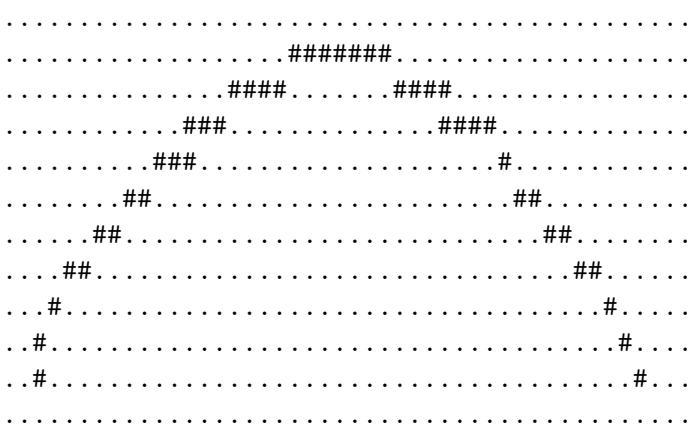
4. À chaque étape, on calcule un nouvel ensemble de points en prenant une interpolation linéaire entre chaque paire de points de la ligne droite calculée à l'étape précédente.
5. On répète l'étape 4 jusqu'à ce qu'il ne reste plus qu'un seul point.
6. Ce dernier point est le point sur la courbe de Bézier correspondant au paramètre t.

Il est possible d'implémenter la sauvegarde et le tracé d'une forme correspondant à une Courbe de Bézier en utilisant l'algorithme de Casteljau.

L'algorithme est récursif et cela est parfait d'un point de vue pédagogique :-)

```
void pixel_curve(Curve * curve, Curve ** pixel, int *nb_pixels);
```

```
    Exemple : curve 10 5 0 25 0 30 10 45
```



10.3. Implémentation de l'algorithme de Casteljau

Cet algorithme implémente une fonction `pixel_curve` qui dessine une courbe de Bézier à l'aide de la méthode de Casteljau. Cette fonction prend en entrée un pointeur vers une structure Shape contenant les informations nécessaires pour dessiner la courbe, ainsi qu'un tableau *tab* qui contiendra les pixels représentant la courbe.

La courbe de Bézier est définie par quatre points de contrôle $p1$, $p2$, $p3$ et $p4$, qui sont stockés dans une structure appelée **Curve**. La fonction `pixel_curve` calcule une série de points de la courbe en appelant la fonction `cj_calc` avec différents paramètres t, où t varie de 0 à 1 avec un pas de 0,001.

La fonction `cj_calc` implémente l'algorithme de Casteljau pour calculer le point de la courbe de Bézier correspondant à un certain paramètre t. L'algorithme de Casteljau utilise une technique de subdivision récursive pour calculer les points intermédiaires de la courbe de Bézier. La fonction `calc_point_median` calcule le point médian entre deux points donnés en interpolant leurs coordonnées en fonction du paramètre t.

Une fois que la fonction `cj_calc` a calculé un point de la courbe de Bézier, la fonction `pixel_curve` crée un pixel correspondant à ce point et l'ajoute à la fin du tableau `tab`. Cette opération est répétée pour chaque valeur de t, ce qui produit une approximation de la courbe de Bézier sous forme de pixels.

La fonction `calc_point_median`

La fonction nommée `calc_point_median` prend deux pointeurs de `Point` et un double `t` en arguments. La fonction calcule le point médian entre $p1$ et $p2$ selon la valeur de t et renvoie un nouveau point *result*. Le point *result* est créé en calculant les valeurs x et y du point médian et en les stockant dans une nouvelle structure *Point*. Finalement, la fonction renvoie le point *result*.

Function `calc_point_median`($Point * p1, Point * p2, double\ t$)
  $double\ x \leftarrow p1 \rightarrow pos\_x * (1-t) + p2 \rightarrow pos\_x * t;$
  $double\ y \leftarrow p1 \rightarrow pos\_y * (1-t) + p2 \rightarrow pos\_y * t;$
  $Point\ result \leftarrow Point x, y;$
  return result ;

La fonction `cj_calc`

La fonction `cj_calc` prend en paramètre un tableau de pointeurs vers des structures *Point*, un entier num_pt représentant le nombre de points dans le tableau et un double t en arguments. La fonction calcule une courbe de Bézier en utilisant l'algorithme de Casteljau. La fonction crée d'abord un tableau tmp_pt de points pour stocker les points intermédiaires calculés lors de l'algorithme de Casteljau. Ensuite, la fonction copie les pointeurs du tableau d'entrée dans le tableau temporaire. La fonction parcourt ensuite le tableau temporaire à l'envers en utilisant deux boucles. Dans la première boucle, la variable i parcourt le tableau de la fin vers le début. Dans la deuxième boucle, la variable j parcourt le tableau de 0 jusqu'à $i-1$. Pour chaque valeur de j, la fonction calcule le point médian entre `tmp_pt[j]` et `tmp_pt[j+1]` en appelant la fonction `calc_point_median` et stocke le résultat dans `tmp_pt[j]`. Finalement, la fonction renvoie le premier élément du tableau temporaire, qui représente le point final de la courbe de Bézier.

Function `cj_calc`($Point ** points, int\ num\_pt, double\ t$)
  $Point\ tmp\_pt[num\_pt];$
  for $i \leftarrow 0$ **to** $num\_pt - 1$ **do**
    $tmp\_pt[i] = *points[i];$
  end
  for $i \leftarrow num\_pt - 1$ **to** 1 **do**
    for $j = 0; j < i; ++j$ **do**
      $tmp\_pt[j] =$ `calc_point_median`$(tmp\_pt[j],\ tmp\_pt[j+1],\ t);$
    end
  end
  return $tmp\_pt[0];$

La fonction `pixel_curve`

La fonction génère des pixels pour une courbe de Bézier à quatre points en utilisant l'algorithme de Casteljau. Elle prend en entrée un pointeur `Curve` qui représente la courbe de Bézier, et un tableau `lst` auquel sont ajoutés les pixels de la courbe et finalement k qui représente le nombre de pixels.

La fonction `pixel_curve` commence par extraire les quatre points de contrôle de la courbe à partir de `Curve`. Dans ce projet nous utilisons uniquement des courbes à quatre points donc $num\_pt \leftarrow 4$. Puis, la fonction initialise t à 0 et utilise une boucle **for** pour calculer les coordonnées d'un point sur

la courbe de Bézier pour chaque valeur de t dans l'intervalle $[0, 1[$ avec un pas de 0.001. Pour cela, elle appelle la fonction `cj_calc`, qui implémente l'algorithme de Casteljau, avec les points de contrôle et la valeur de t. Elle crée ensuite un pixel avec les coordonnées du point calculé et ajoute ce pixel au tableau *tab*.

Enfin, elle retourne le tableau `tab` qui contient les pixels de la courbe de Bézier.

Data: curve, tab[], *k
Result: liste des pixels dans le tableau tab[], k le nombre d'éléments
Function `pixel_curve(`*Shape *curve, int tab[], int * k*`) :`
  $tab\_points \leftarrow [curve \rightarrow p1, curve \rightarrow p2, curve \rightarrow p3, curve \rightarrow p4]$;
  $num\_pt \leftarrow 4$;
  $k \leftarrow 0$;
  $t \leftarrow 0$;
  for t ***from*** 0 ***to*** 1.0 ***by*** 0.001 **do**
    $cjp1 \leftarrow cj\_calc(tab\_points, num\_pt, t)$;
    $px \leftarrow$ création d'un Pixel($cjp1.pos\_x$, $cjp1.pos\_y$);
    $tab[k++] \leftarrow px$;
  end

Algorithm 1: Algorithme de Casteljau

11. Système de calques

Les calques sont des structures d'organisation de dessins pour l'utilisateur. Ils vont permettre de créer des couches de dessin qui se superposent les unes sur les autres. Chaque calque peut contenir une ou plusieurs formes. Les claques sont identifiés par un `id` unique, et l'utilisateur peut les afficher ou les cacher. Si un calque est caché alors toutes les formes de ce calque ne seront pas affichées à l'écran mais restent en mémoire. L'utilisateur peut décider à tout moment de les afficher ou de les cacher.

11.1. Système de calques à 1 niveau

Dans cette partie nous proposons d'inclure, par défaut, les formes dans des claques. Dès le lancement de l'application, un calque par défaut doit être créé. Pour gérer les calques nous proposons d'ajouter les commandes suivantes :

— `list layers` : affiche la liste des calques disponibles. Le calque marqué par un astérisque (`*`) est le calque sélectionné, c'est a dire si on créé une forme elle sera attachée à ce calques. Par la suite, il faut afficher l'id du calque, et si le calque est visible `V` ou caché `H` (`V:` visible / `H:` Hide). Finalement, afficher le nom du calque.
L'affichage se fait de la manière suivante :
`(*/-) id (visible/ou pas) nom du claque`

```
Exemple d'utilisation :
    >> list layers
     *   2 (V) Layer 1
    done
```

— `new layer [layer_name]` : permet de créer un calque avec le nom `layers_name` donné en paramètre. Le nom est optionnel, si l'utilisateur ne fourni pas de nom alors, on lui attribut un nom automatiquement sous la forme `layers_id`. Le calque créé devient automatiquement sélectionné.
Voir l'exemple de création de calque ci-dessous :

```
Exemple d'utilisation :
    >> list layers
     *   2 (V) Layer 1
    done
    >> new layer
    done
    >> list layers layer_name
     -   2 (V) Layer 1
     *   3 (V) layer_name
    done
```

— `select layer [ID]` : permet de sélectionner un claque en donnant son id.

```
Exemple d'utilisation :
    >> list layers
     -   2 (V) Layer 1
     *   3 (V) layer_name
```

```
done
>> select layer 2
done
>> list layers
 *   2 (V) Layer 1
 -   3 (V) layer_name
done
```

— `delete layer [id]` : permet de supprimer un calque en donnant son `id`.

```
Exemple d'utilisation :
    >> list layers
     *   2 (V) Layer 1
     -   3 (V) layer_name
    done
    >> delete layer 3
    >> list layers
     *   2 (V) Layer 1
```

— `set layer visible [id]` / `set layer hide [id]` : permet d'afficher ou de cacher un calque.

```
Exemple d'utilisation :
    >> list layers
     *   2 (V) Layer 1
     -   3 (V) layer_name
    done
    >> set layer hide 3
    >> list layers
     *   2 (V) Layer 1
     -   3 (H) layer_name
    done
```

Pour gérer les calques dans notre programme nous avons besoin d'une structure de données qui va représenter un calque (voir plus bas la structure : `struct layer`). Étant donné qu'il va y avoir plusieurs calques, on va les gérer dans un tableau. Comme vous avez pu le constater lors de la gestion des tableaux de pixels dans partie précédente, il faut à chaque fois passer en paramètre un pointeur (voir un double pointeur) qui représente le tableau suivi le la taille du tableau, qu'il faut aussi faire un passage par adresse pour pouvoir changer sa valeur (car la taille du tableau change au fur et à mesure d'ajouter des éléments dedans). Ici nous souhaitons nous éviter cette lourdeur et créer directement une structure `array` qui va nous faciliter le travail. Pour vous simplifier la tâche à vous aussi, nous allons vous fournir le code sous forme de deux fichier *array.h* et *array.c* :

```c
// array.h
#ifndef _ARRAY_H_
#define _ARRAY_H_

#define MAX_SIZE 1000

struct array {
```

```
  unsigned int max_size;
  unsigned int size;
  void*        tab[MAX_SIZE];
};

typedef struct array Array;

Array* array_create();
void array_delete(Array* array);
unsigned int array_get_size_max(Array* array);
unsigned int array_get_size(Array* array);
void* array_get_element_at (Array* array, int pos);
int array_insert_element (Array* array, void* element);

#endif


// array.c
#include "array.h"

Array* array_create(){
  Array* array = (Array*) malloc (sizeof(Array));
  array->size = 0;
  array->max_size = MAX_SIZE;
  return array;
}

void array_delete(Array* array){
  free (array);
}

unsigned int array_get_size_max(Array* array){
  return array->max_size;
}

unsigned int array_get_size(Array* array){
  return array->size;
}

void* array_get_element_at (Array* array, int pos){
  if(pos < array_get_size(array))
    return array->tab[pos];
  return NULL;
}

int array_insert_element (Array* array, void* element){
  if(array_get_size(array)>= array_get_size_max(array)) return -1;
  array->tab[array_get_size(array)] = element;
  array->size = array->size + 1;
  return array->size - 1;
}
```

Une fois que notre `Array` est prêt. Nous allons définir les calques comme suit :

```c
#define VISIBLE 0
#define UNVISIBLE 1
struct layer {
    unsigned int id;
    char name[255];
    unsigned char visible;
    Array shapes;
};

typedef struct layer Layer;
typedef Array LayersList;  // ici on va utiliser un notre type Array

Layer *create_layer(int id, char *name);
void delete_layer(Layer * layer);
LayersList *create_layers_list();
void delete_layers_list(LayersList * layer_list);
void add_layer_to_list(LayersList * layer_list, Layer * layer);
void remove_layer_from_list(LayersList * layer_list, Layer * layer);
void set_layer_visible(Layer * layer);
void set_layer_unvisible(Layer * layer);
void add_shape_to_layer(Layer * layer, Shape * shape);
void remove_shape_to_from(Layer * layer, Shape * shape);
```

Où :

- La fonction `Layer* create_layer(int id, char* name)` permet de créer un objet layer en lui donnant un nom. Si `*name == NULL` alors on lui attribut un nom automatiquement avec l'id du layer (`layer_ID`).
- La fonction `void delete_layer(Layer* layer);` permet de supprimer un layer, sans oublier de libérer la mémoire allouée.
- La fonction `LayersList* create_layers_list();` permet de créer une liste de calques et renvoie un pointeur sur cette liste.
- La fonction `void delete_layers_list(LayersList* layer_list)` ; permet de supprimer la liste des calques et libérer la mémoire des calques qui la composent.
- La fonction `void add_layer_to_list(LayersList* layer_list, Layer* layer);` permet d'ajouter un calque à liste des calques.
- La fonction `void remove_layer_from_list(LayersList* layer_list, Layer* layer);` permet de supprimer de la liste le caque pointé par `layer`.
- La fonction `void set_layer_visible(Layer* layer);` permet de rendre un calque visible, en le mettant simplement à `VISIBLE` l'attribut `visible`.
- La fonction `void set_layer_unvisible(Layer* layer);` identique à la précédente fonction, en attribuant au champs `visible` la valeur `UNVISIBLE`.
- La fonction `void add_shape_to_layer(Layer* layer, Shape* shape);` permet d'ajouter une forme à un calque donné en paramètre.
- La fonction `void remove_shape_to_from(Layer* layer, Shape* shape);` permet de supprimer la forme donnée en paramètre du calque.

11.2. Système de calques à plusieurs niveaux

Parfois dessiner un objet complexe (comme un *soleil* par exemple) peut être constitué de plusieurs formes : un cercle et un certain nombre de lignes qui partent du centre de ce même cercle. Il serait donc intéressant de regrouper les formes qui le constitue dans un calque tout seul. Mais peut être l'objet soleil à lui seul ne justifie pas de constituer un calque. Dans certain logiciels de dessin il est possible de créer des sous calques c'est à dire des calques imbriqués dans d'autres calques. Dans l'exemple précédent on peut alors imaginer un calque ciel qui contient des sous calques, un premier qui va contenir les nuages, et un second le soleil. Ainsi, il est possible par exemple de déplacer le soleil (par translation) d'une position à une autre sans déplacer le ciel au complet.

Objectif

Nous souhaitons proposer un système similaire pour notre application.

12. Les listes chaînées

Tout au long de ce projet nous avons utilisé des tableaux, parfois statiques et parfois dynamiques. La difficulté de l'utilisation des tableaux réside dans le fait que :

1. À chaque fois il faut garder en mémoire leur taille logique, particulièrement lourd lorsqu'un tableau est passé en paramètre et que la taille de celui-ci est modifiée dans la fonction.
2. Plus ennuyeux encore, la taille physique des tableaux est statique, pour modifier cette taille il faut passer par la fonction realloc qui peut déménager le tableau, ainsi son adresse initiale n'est plus valable ce qui peut causer des problèmes.
3. La suppression d'un élément au milieu du tableau nous oblige à décaler le reste des éléments.
4. Un avantage certain avec les tableaux qui offrent un accès en O(1) pour chercher un élément à une position donnée. Or dans notre projet à aucun nomment nous utilisons les accès directs.

Pour résoudre le premier problème, nous avons proposé une première solution dans la partie 11 sur les calques. La solution consiste à utiliser une structure pour stocker à la fois le tableau et sa taille (array). De plus toutes les opérations sur le tableau se font depuis des fonctions.

Pour résoudre le reste des problèmes, nous proposons d'utiliser des listes doublement chaînées, car il n'y a pas de taille limite qui nous obligerait à faire un `realloc`. De la même façon que notre dernière solution nous allons stoker notre liste dans une structure afin de faciliter son passage en paramètre et aussi d'utiliser un certain nombre de fonctions pour les différentes opérations.

Nous montrons ci-après une proposition d'une structure de liste :

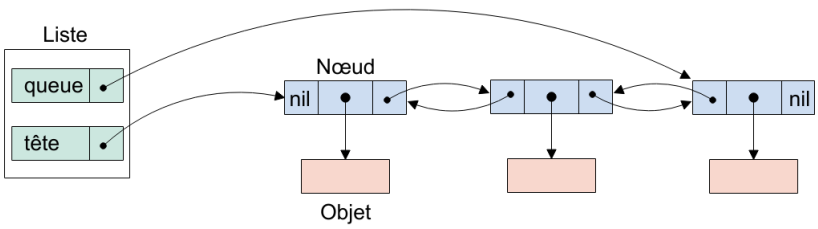


FIGURE 11 – Structure d'une liste doublement chaînée

```
typedef struct lnode_ {
    void *data;
    struct lnode_ *prev;
    struct lnode_ *next;
};
typedef struct lnode_ lnode;

struct list_ {
    lnode *head;
    lnode *tail;
};
typedef struct list_ LIST;
```

Que l'on peut gérer avec les fonctions suivantes :

- La fonction `lnode* lst_create_lnode(void *dat);` permet de créer un nouveau noeud et retourne un pointeur vers ce noeud. Ce qui nous permettra d'insérer ce noeud dans la liste à l'aide des fonctions suivantes.
- La fonction `list* lst_create_list();` permet de créer une liste vide.
- La fonction `void lst_delete_list(list* lst);` permet de supprimer tous les éléments de la liste. Attention, cette fonction libérera uniquement les noeuds de la liste. Si ces noeuds contiennent des pointeurs vers des objets alloués par la fonction `malloc`, ils ne seront pas libérés.
- La fonction `void lst_insert_head(list* lst, lnode* pnew);` permet d'insérer le noeud donné en paramètre au début de la liste.
- La fonction `void lst_insert_tail(list* lst, lnode* pnew);` permet d'insérer le noeud donné en paramètre à la fin de la liste.
- La fonction `void lst_insert_after(list* lst, lnode* pnew, lnode* ptr);` permet d'insérer le noeud donné en paramètre juste après l'élément pointé par `ptr`.
- La fonction `void lst_delete_head(list* lst);` permet de supprimer le premier élément de la liste, cette fonction ne libère pas l'espace pointé par `data`.
- La fonction `void lst_delete_tail(list* lst);` permet de supprimer le dernier élément de la liste, cette fonction ne libère pas l'espace pointé par `data`.
- La fonction `void lst_delete_lnode(list* lst, lnode* ptr);` permet de supprimer l'élément pointé par `ptr`, cette fonction ne libère pas l'espace pointé par `data`.
- La fonction `void lst_erase(list* lst);` permet de supprimer tous les éléments de la liste, cette fonction ne libère pas l'espace pointé par `data`.
- La fonction `lnode* get_first_node(list* lst);` permet de retourner un pointeur sur le premier élément de la liste.
- La fonction `lnode* get_last_node(list* lst);` permet de retourner un pointeur sur le dernier élément de la liste.
- La fonction `lnode* get_next_node(list* lst, lnode* lnode);` permet de retourner un pointeur sur l'élément qui est juste après l'élément pointé par `ptr`.
- La fonction `void* get_previous_elem(list* lst, lnode* lnode);` permet de retourner un pointeur sur l'élément qui est juste avant l'élément pointé par `ptr`.

Objectif

L'objectif est de remplacer les tableaux statiques par des listes chaînées pour la liste des calques et les formes insérées dans les calques.

Donc plus précisément la structure `area` ressemblera :

```
struct area {
    unsigned char id;
    char name[255];
    unsigned int width;
    unsigned int height;
    char** area;
    LIST lst_layers; // liste de calques
};
```

Puis dans les claques, on utilisera aussi une liste chaînée pour représenter les formes contenues dans un calque :

```
struct layer {
    unsigned int id;
    char name[255];
    unsigned char visible;
    LIST shapes;  // liste de formes
};
```