Projet C

Vector Text-based Editor

Valentin Gourjon - Antoine Souben-Fink - Groupe A - 2027

eFrei
PARIS PANTHÉON-ASSAS UNIVERSITÉ

Table des matières

I. Introduction

Ce projet à pour objectif de nous faire découvrir le langage et ces fonctionnalités.
L'objectif est de recréer un éditeur d'image vectoriel dans la console, en utilisant des structures pour les différentes formes et même des listes chaînées pour le stockage, ce qui nous permet de se familiariser avec les pointeurs et les struct.

II. Présentation fonctionnelle

Ce projet inclut la majorité des fonctionnalités proposées dans le guide. En voici la liste :

- Utilisation du programme à l'aide de commandes
- Utilisation de calques (layers)
- Liste des formes : Point, Ligne, Carré, Rectangle, Cercle, Polygone, Courbe de Bézier
- Identification des calques et des formes par des ids uniques

III. Présentation technique

La structure de données utilisée majoritairement pour stocker les différentes formes ainsi que les Pixels est une liste doublement chaînée.

Les structures principales sont définies de cette manière :

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


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

On voit bien le lien entre le maillon, le maillon suivant et le maillon précédent.
De plus, voici tous les prototypes des fonctions implémentées pour gérer les liste chainées :

```
lnode* lst_create_node(void *data);
```

```
List* lst_create_list();
void lst_delete_list(List *list);
void lst_insert_head(List *list, lnode* pnew);
void lst_insert_tail(List *list, lnode* pnew);
void lst_insert_after(List *list, lnode *pnew, lnode *ptr);
void lst_delete_head(List *list);
void lst_delete_tail(List *list);
void lst_delete_node(List *list, lnode *ptr);
lnode* get_first_node(List *list);
lnode* get_last_node(List *list);
lnode* get_next_node(lnode *node);
void* get_previous_elem(List *list);
```

Avec l'ensemble de ses fonctions, nous pouvons facilement manipuler les listes, l'accès aux éléments, leur insertion et délétion. Ainsi, cette structure nous permet de facilement stocker tout type de données sans problème d'accès à la mémoire, évitant d'utiliser constamment la fonction realloc.

Pour la gestion des commandes, nous avons utilisé plusieurs fonctions, dont notamment :

- parseCommand(char* cmd) : prend en paramètre l'entrée effectuée par l'utilisateur (commande + arguments sous forme de string) et renvoie une variable de type Command (qui contient le nom de la commande appelée, un tableau d'arguments (char*), et le nombre d'arguments)

  Ci dessous le code de la fonction, dont le fonctionnement est assez intéressant

```
Command parseCommand(char* cmd) {
    // Convertit une chaîne de caractères en commande (commande + arguments + nombre d'arguments)
    Command command;
    command.command = malloc(sizeof(char) * 100);
    command.args = malloc(sizeof(char*));
    int arg_count = 0;
    int cmd_length = 0;
    while (cmd[cmd_length] != ' ' && cmd[cmd_length] != '\0' && cmd[cmd_length] != '\n') {
        command.command[cmd_length] = cmd[cmd_length];
        cmd_length++;
    }
    command.command[cmd_length] = '\0';
    if (cmd[cmd_length] == '\0' || cmd[cmd_length] == '\n') {
        command.arg_count = 0;
        return command;
    }
    for (int i = 0; i < strlen(cmd); i++) {
        if (cmd[i] == ' ') {
            arg_count++;
        }
    }
    command.arg_count = arg_count;
    for (int i = 0; i < arg_count; i++) {
        char* arg = malloc(sizeof(char) * 100);
```

```
            int space_count = 0;
            int arg_length = 0;
            for (int j = 0; j < strlen(cmd); j++) {
                if (cmd[j] == ' ' || cmd[j] == '\n') {
                    space_count++;
                }
                if (space_count == i + 1 && cmd[j] != ' ') {
                    arg[arg_length] = cmd[j];
                    arg_length++;
                }
            }
            arg[arg_length] = '\0';
            command.args[i] = arg;
        }
        return command;
    }
```

- execCmd : se charge de demander l'entrée de commande à l'utilisateur, d'appeler la fonction parseCommand pour parser ce que l'utilisateur a demandé, et d'exécuter les commandes

Pour les algorithmes de tracé de ligne, de cercle et de courbe, nous avons utilisé :

- Ligne : Algorithme de tracé de segment de Bresenham, dont le principe à été trouvé sur la page Wikipédia donnée dans le cours.

  Cet algorithme est assez long mais présente une gestion parfaite de tous les cas. Il est relativement optimisé et fonctionne parfaitement pour notre utilisation.

  De plus, lors de la programmation de cet algorithme, la partie du guide du projet qui en parle n'était pas sortie et donc nous n'avions pas connaissance de l'algorithme de tracé de ligne de Nicolas Flasque
- Cercle : Algorithme de tracé de cercle d'Andres, encore une fois à partir de la page Wikipédia donnée dans le cours.

  C'est un algorithme très optimisé qui reprend en partie le concept de l'algorithme de tracé d'arc de cercle de Bresenham mais en corrigeant les erreurs que celui-ci peut créer.

  De plus, son implémentation est très rapide, chaque calcul n'étant fait qu'une fois pour les 8 octants du cercle, réduisant grandement sa complexité algorithmique.
- Courbe de Bézier : Algorithme de Casteljau, dont le principe est donné dans le document guide fourni

Comme expliqué précédemment, les formes sont stockées dans des listes doublement chaînées. Les formes disponibles sont le Point, la Ligne, le Cercle, le Carré, le Rectangle, le Polygone et la Courbe de Bézier. La déclaration de leur structure est effectuée comme ceci :

```
typedef struct {
    int x;
```

```
    int y;
} Point;

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

typedef struct {
        Point* center;
        int radius;
} Circle;

typedef struct {
    Point* p;
    int width;
    int height;
} Rectangle;

typedef struct {
    Point* p;
    int side;
} Square;

typedef struct {
    int nbPoints;
    Point** points;
} Polygon;

typedef struct {
    Point* p1;
    Point* p2;
    Point* p3;
    Point* p4;
} Curve;
```

Pour rassembler les formes dans un seul type, la structure Shape est utilisée, qui est défini comme ceci :

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

typedef struct shape_ {
    ShapeType type;
```

```
    unsigned int id;
    void* ptrShape;
} Shape;
```

Ainsi, toutes les formes sont facilement manipulable, les fonctions associées a Shape fonctionnant par disjonction de cas à l'aide de l'instruction

```
switch (shape -> type) {
    case
```

Une des étapes les plus difficiles était la gestion des SegmentationFault, qui était dûe aux tableaux en C et à l'allocation dynamique, compliquée. Pour résoudre ce problème, l'utilisation des listes doublement chaînées a été salvatrice et a permis la facilitation de l'insertion et de la délétion des éléments dans les listes.

La zone de dessin est définie dans la structure Area, qui contient sa largeur, sa longueur, ainsi que la liste de tous les calques, qui est une liste chaînée de structures layer. Chaque structure layer possède une liste chaînée des formes qui y sont dessinées. De plus, leur visibilité est définie grâce à leur donnée "visible". Le calque actuellement sélectionné est défini dans la zone de dessin par son id, stocké dans la variable "id_layer".

Afin de simplifier au maximum l'exécution du programme, nous avons aussi utilisé GitHub Actions afin d'automatiser la compilation et la mise à disposition d'exécutables pour toutes les plateformes (voir https://github.com/DocSystem/ScribbleVibes)

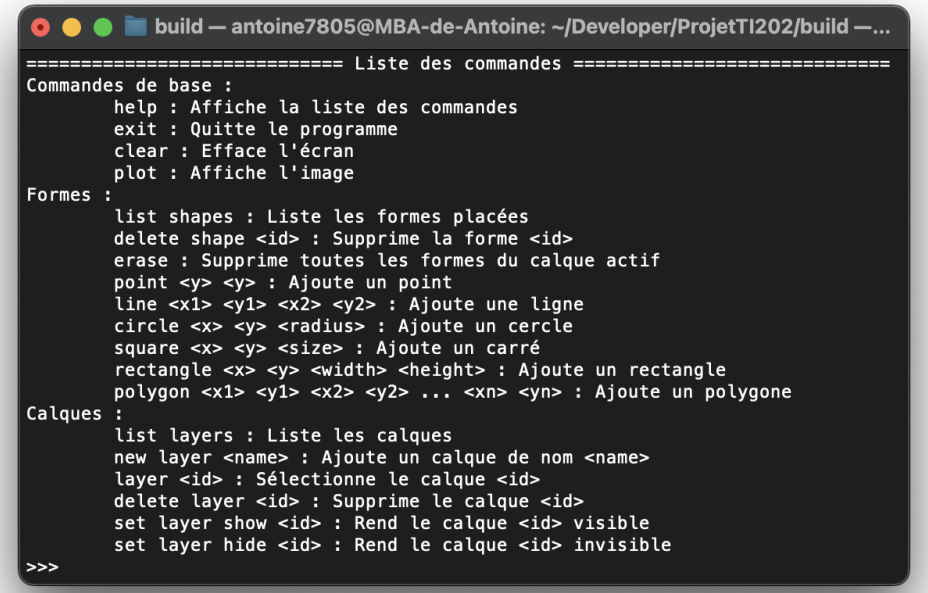
Aussi, afin de supporter un maximum d'applications de terminal et de systèmes d'exploitations différents, nous avons utilisé des instructions type **#if** (pour différencier la compilation sur Windows, macOS, Linux, et autre) afin de permettre le meilleur fonctionnement peu importe la plateforme utilisée (notamment utilisé sur les fonctions getWindowSize et pour vérifier s'il est possible d'utiliser des emojis)

IV. Ordre d'appel des fonctions

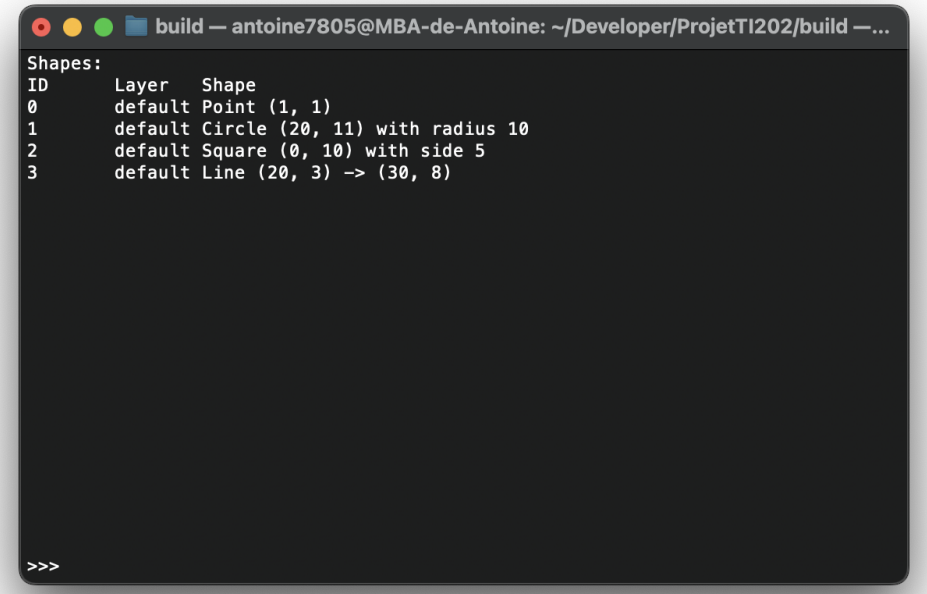
L'ordre d'appel des fonctions est détaillé dans ce graphique.

V. Résultats

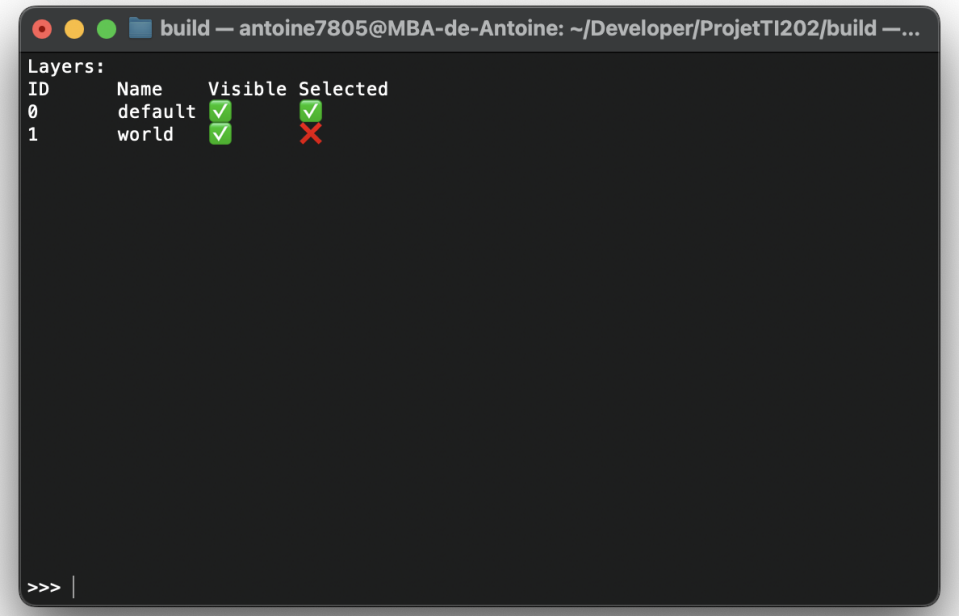
Affichage du **plot**



Affichage de **help** (page d'aide - liste des commandes)



Affiche de **list shapes** (liste des formes



)

Affichage de **list layers** (liste des calques)

VI. Conclusion

Ce projet fut intéressant à réaliser, nous avons pu apprendre beaucoup de choses en le réalisant, notamment la gestion avancée de l'affichage dans un terminal, nous avons aussi créé des algorithmes de tracé de ligne, de cercle, ou encore de courbes.

La manipulation des pointeurs a été compliquée au début, mais avec le temps et au fur et à mesure du projet, nous avons appris à nous y habituer et les appréhendons désormais sans problème.

Nous avons aussi pu développer davantage notre culture informatique grâce à l'apprentissage de ce nouveau langage qui augmente l'étendue de nos capacités.

La résolution des erreurs était une étape parfois difficile mais nécessaire et qui nous a obligé à accentuer notre rigourosité pour faciliter ce passage.

Une autre compétence qui a été défiée pendant ce projet est le travail d'équipe, notamment la synchronisation des fichiers dont la gestion à finalement été implémentée grâce à Github, permettant de travailler simultanément sans être un obstacle au test de l'autre.

Pour conclure, ce projet a été très enrichissant aussi bien sur les points techniques tels que le langage C en lui-même et l'algorithmique/programmation en général, mais aussi sur l'aspect moral, développant des softs-skills qui nous seront utiles dans le travail en entreprise.