Membres du groupe :
OUANNOUGHI Kenza
VOICU Francesca
NDONG Papa-Moussa
KABORE Wandpegré

Conception du Langage *draw*++

Rapport du projet

Ing 1 GMA - Apprentis

Lien du projet sur Github : https://github.com/Fsv6/Projet_C_ING1/tree/main

Table des Matières

Table des Matières..... 2
Introduction.....3
1. La syntaxe et l'interface de notre langage draw++.....3
1.1 Les commandes disponibles..... 3
1.1.1 Commandes simples..... 3
1.1.2 Commandes complexes, en bloc.....6
1.2 L'Éditeur : l'interface utilisateur.....7
1.2.1 Accès et lancement de l'éditeur..... 7
1.2.2 Vue d'ensemble..... 7
1.2.3 Création d'un fichier .draw.....8
1.2.4 Ouverture et affichage du contenu d'un fichier .draw, avec coloration des mot-clés et variables..... 8
1.2.5 Modification du contenu d'un fichier .draw.....9
1.2.6 Interprétation d'un fichier .draw et exécution du fichier .py résultant..... 9
1.2.6 Assistance sur les commandes fournies par la commande help..... 10
1.2.7 Récapitulatif.....10
2. L'interpréteur et traducteur C.....11
2.1 La structure globale du code.....11
2.2 La lecture et le parsing des fichiers .draw..... 12
2.2.1 Première lecture du fichier .draw : interpret_draw_file()..... 12
2.2.2 Traitement commande par commande : process_command()..... 13
2.2.3 Compilation des commandes : execute_command()..... 14
2.2.4 Finalisation : finalize_execution()..... 14
2.3 Les scripts Python résultant de l'interprétation des fichiers .draw.....14
2.3.1 commands.c..... 14
2.3.2 python_functions_writer.c : définition des fonctions Python.....15
2.3.3 bloc_commands_manager.c : logique conditionnelle et boucles.....16
2.4 La gestion des erreurs implémentée..... 16
2.4.1 Dans l'éditeur.....16
2.4.2 Dans le fichier parseur de commandes..... 16
3. Exemples d'exécution.....17
3.1 Exemple 1 : Ouverture et modification d'un fichier .draw.....17
3.2 Exemple 2..... 18
3.3 Exemple 3..... 19
Conclusion..... 20

Introduction

Ce projet a pour but la conception d'un langage de programmation, draw++, permettant à un utilisateur de dessiner et d'animer simplement des formes géométriques à partir d'instructions prédéfinies, écrites dans un fichier .draw puis exécuté à sa demande.

De la création d'un fichier .draw à l'apparition des dessins sur l'écran, il y a trois étapes que nous avons implémentées et que nous décrivons dans ce rapport de projet. En effet, l'utilisateur interagit avec les fichiers .draw à partir d'un éditeur, qui à l'exécution lance un compilateur de commandes en C, produisant un fichier exécutable en Python chargé de produire parfaitement ce qui a été écrit par l'utilisateur usant du langage draw ++.

Les instructions implémentées et la grammaire du langage ont été pensées et choisies par nos soins, en nous basant sur les instructions équivalentes en python et les possibilités offertes par la librairie *turtle*, que nous avons choisie pour la création de nos curseurs, dessins et animations.

1. La syntaxe et l'interface de notre langage *draw*++

1.1 Les commandes disponibles

Pour le choix des commandes *draw*++ proposées, nous nous sommes basés sur les instructions possibles en Python et dans la librairie *turtle*, ainsi que leurs orthographes et leurs paramètres. Nous avons également imaginé quel genre de fonctionnalités serait utile à l'utilisateur. Ainsi, nous avons implémenté d'une part des commandes simples, en une ligne, limitant ainsi les mots inutiles à écrire et analyser. D'autre part, nous avons permis l'utilisation de commandes plus complexes, en blocs, pour complexifier les fonctionnalités du langage.

1.1.1 Commandes simples

Voici la liste des commandes simples disponibles, leur usage et les paramètres nécessaires à leur bon fonctionnement :

- CURSOR : permet de créer un curseur à une certaine position (x,y), et de choisir sa visibilité. Par défaut, sa couleur est noire et sa visibilité est activée.
  Paramètres :
  - id (entier) : identifiant donné au curseur
  - x (entier) : position x
  - y (entier) : position y
  - Visible (boolean) : choix de la visibilité

  Exemple:

CURSOR 1 100 100 TRUE

- COLOR : permet de modifier la couleur d'un curseur
  Paramètres :
  - id (entier) : identifiant donné au curseur
  - color(string) : couleur choisie

  Exemple:
  COLOR 1 blue

- THICKNESS: permet de modifier l'épaisseur de trait d'un curseur
  Paramètres :
  - id (entier) : identifiant donné au curseur
  - thickness(float) : épaisseur

  Exemple:
  THICKNESS 1 0.3

- MOVE: permet d'avancer un curseur vers l'avant d'une certaine distance donnée
  Paramètres :
  - id (entier) : identifiant donné au curseur
  - distance(float) : distance donnée en pixels

  Exemple:
  MOVE 1 50

- GOTO: permet de déplacer le curseur à une certaine position (x,y). Le curseur se téléporte en quelque sorte
  Paramètres :
  - id (entier) : identifiant donné au curseur
  - x (entier) : position x
  - y (entier) : position y

  Exemple:
  GOTO 1 150 -200

- ROTATE : permet de modifier l'angle d'un curseur et donc la direction vers laquelle il pointe
  Paramètres :
  - id (entier) : identifiant donné au curseur
  - angle(float) : angle donné exprimé en degrés

  Exemple:
  ROTATE 1 -90

- LINE : permet de tracer une ligne depuis la position actuelle du curseur, vers l'avant et d'une certaine distance donnée.
  Paramètres :
    - id (entier) : identifiant donné au curseur
    - distance(float) : distance donnée en pixels
    - id_figure (entier) : identifiant donné à la figure de type "line"

  Exemple:
  LINE 1 50 1

- CIRCLE : permet de tracer un cercle d'un certain rayon.
    - id (entier) : identifiant donné au curseur
    - rayon(float) : rayon donné en pixels
    - id_figure (entier) : identifiant donné à la figure de type "circle"

  Exemple:
  CIRCLE 1 50 2

- SQUARE : permet de tracer un carré d'un certain côté.
    - id (entier) : identifiant donné au curseur
    - côté(float) : côté donné en pixels
    - id_figure (entier) : identifiant donné à la figure de type "square"

  Exemple:
  SQUARE 1 20 3

- RECTANGLE : permet de tracer un rectangle d'une certaine longueur et largeur.
    - id (entier) : identifiant donné au curseur
    - largeur(float) : largeur donnée en pixels
    - longueur (float) : longueur donnée en pixels
    - id_figure (entier) : identifiant donné à la figure de type "rectangle"

  Exemple:
  RECTANGLE 1 50 4

- POINT : permet de tracer un point à la position actuelle du curseur
    - id (entier) : identifiant donné au curseur
    - id_figure (entier) : identifiant donné à la figure de type "point"

  Exemple:
  POINT 1 50 5

- ARC : permet de tracer un demi-cercle(arc) d'un certain rayon.
    - id (entier) : identifiant donné au curseur
    - rayon(float) : rayon donné en pixels
    - id_figure (entier) : identifiant donné à la figure de type "arc"

  Exemple:
  ARC 1 50 6

- ANIME : permet de déplacer une ou plusieurs figures vers une direction donnée à l'aide d'un angle et d'une certaine distance donnée.
  Paramètres :
  - ids (entier) : identifiant donné au curseur
  - distance(float) : distance donnée en pixels
  - angle(float) : angle donné en pixels, pour préciser la direction

  Exemple:
  ANIME [1,2,3,4,5,6] 50 5 25

- SET : permet de créer une variable et de lui donner une valeur
  Paramètres :
  - variable_name(string) : nom de la variable
  - expression (float ou bien nom d'une autre variable) : valeur donnée à la variable

  Exemple:
  SET VARIABLE X = 10

1.1.2 Commandes complexes, en bloc

- IF : permet de réaliser une ou plusieurs commandes dans un bloc en fonction du respect d'une condition sur une ou les coordonnées de la position d'un curseur
  Paramètres :
  - id (entier) : identifiant du curseur
  - condition_x : condition sur la position X du curseur si voulu
  - condition_y : condition sur la position Y du curseur si voulu

  Exemple:
  ```
  IF CURSOR WITH ID = 1 IS IN POSITION X > 10 {
        COLOR 1 RED,
        CURSOR 2 -100 20 TRUE,
  } ELSE {
  }
  ```

- FOR : permet de réaliser une ou plusieurs commandes dans un bloc en fonction d'itération sur une variable
  Paramètres :
  - variable(entier) : variable sur laquelle on itère
  - (min, max) : intervalle sur lequel on veut itérer

  Exemple:
  ```
  FOR I IN RANGE (1, 10) {
        MOVE 1 I*10
  }
  ```

- WHILE : permet de réaliser une ou plusieurs commandes tant qu'une condition est respectée
  Paramètres :
  - variable(entier) : variable sur laquelle on itère
  - condition_x : condition sur la position X du curseur si voulu
  - condition_y : condition sur la position Y du curseur si voulu

  Exemple:
  WHILE CURSOR WITH ID = 1 IS IN POSITION X > 10 {
  COLOR 1 RED,
  CURSOR 2 -100 20 TRUE,
  }

1.2 L'Éditeur : l'interface utilisateur

1.2.1 Accès et lancement de l'éditeur

Afin d'accéder à l'éditeur du langage draw, l'utilisateur doit interagir en mode console. Ainsi, il doit ouvrir un terminal et exécuter le fichier *.exe* du programme, dont la localisation dans son gestionnaire de fichiers dépend d'où il l'a téléchargé.

Il est également possible pour l'utilisateur de compiler le projet lui-même pour produire le fichier *.exe*. Il lui faudra toutefois faire attention à avoir un compilateur de C (type gcc)bien téléchargé sur son ordinateur.

1.2.2 Vue d'ensemble

Une fois le projet exécuté, l'utilisateur accède au menu principal de l'éditeur, qui propose plusieurs options et s'affiche ainsi.

```
=== MAIN MENU ===
1. Create a .draw file
2. Open a .draw file
3. Compile a file
4. Quit
Enter a number for your choice or type 'help' for assistance:
```

C'est ce menu textuel qui constitue l'interface principale pour l'utilisateur souhaitant créer, ouvrir, modifier un fichier *.draw* ou en lancer l'interprétation.

Le code de l'éditeur est contenu dans le fichier *main.c* et s'articule autour de fonctions dédiées (telles que *create_file(), open_file(),* etc.) qui sont chacune responsables d'une

fonctionnalité proposée par l'éditeur. Par ailleurs, le *main()* propose une boucle *do-while,* permettant ainsi à l'utilisateur de sélectionner les options souhaitées en tapant le chiffre correspondant, jusqu'à ce qu'il décide de quitter l'application. C'est en implémentant une structure *switch-case* au sein de la boucle, que le code est capable de lancer les fonctions correspondantes au chiffre lu dans la console avec une fonction C *scanf.*

L'éditeur est donc également le gestionnaire du répertoire de fichiers .draw créé au sein du projet et dédié au stockage des fichiers respectifs.

1.2.3 Création d'un fichier *.draw*

Lorsque l'utilisateur choisit l'option 1, "Create a draw file", l'éditeur demande un nom de fichier (sans extension) à l'utilisateur, puis se base sur la réponse pour construire un chemin complet *(../draw_files/nom_fichier.draw)*. Ce chemin sera ensuite utilisé pour y ouvrir le nouveau fichier en mode écriture. L'utilisateur pourra alors y écrire ses instructions ligne par ligne, jusqu'à taper le terme "*END*", signifiant ainsi à l'éditeur qu'il a fini d'écrire le fichier. Les instructions sont alors sauvegardées automatiquement

1.2.4 Ouverture et affichage du contenu d'un fichier *.draw,* avec coloration des mot-clés et variables

Lors du choix de l'option 2 : "Open a draw file", l'éditeur :

1. Affiche la liste des fichiers disponibles dans le répertoire draw_files.
2. Demande à l'utilisateur de sélectionner le fichier qu'il souhaite consulter.
3. Ouvre le fichier en lecture et affiche son contenu
4. Propose de modifier le fichier

Il est important de préciser que lors de l'affichage du contenu d'un fichier .draw, l'éditeur colore les mot clés en rouge, les variables en vert et les accolades délimitant les blocs en bleu. Pour ce faire, la fonction *display_colored_line*() copie d'abord la ligne lue dans un tampon temporaire et la parcourt caractère par caractère. Lorsqu'elle rencontre une accolade ouvrante ou fermante ({ ou }), elle l'affiche en bleu, puis continue. Dans les autres cas, elle construit un token (suite de caractères jusqu'à un espace ou une accolade) et vérifie s'il correspond à un mot-clé : s'il figure dans le tableau de mots-clés dynamiques, correspondant aux nom de commandes stockées dans le dictionnaire dédié, ou dans la liste supplémentaire (p. ex. WITH, IN RANGE, etc.), il est affiché en rouge. Sinon, il est affiché en vert et correspondant donc à une variable.

Cet affichage coloré facilite ainse la relecture et le repérage d'éléments importants (commandes, structures de bloc, etc.) par l'utilisateur.

```
Enter the name of the file to open (without extension): test

=== Content of file test ===
SET VARIABLE x = 42
SET VARIABLE y = x
SET VARIABLE z = x * 2 + 3
CURSOR 1 y z TRUE
IF CURSOR WITH ID = 1 IS IN POSITION X > 10 {
    COLOR 1 red,
    CURSOR 2 -100 -100 TRUE,
    CURSOR 3 100 100 TRUE,
    COLOR 2 green
} ELSE {
    COLOR 1 green
}
CIRCLE 1 100 2
FOR I IN RANGE (1,10) {
    MOVE 1 I*10,
    PASS
}
```

1.2.5 Modification du contenu d'un fichier *.draw*

Après l'ouverture et l'affichage du contenu d'un fichier, l'éditeur propose de le modifier. S'il le désire, deux choix sont alors possibles pour l'utilisateur. D'une part, il peut ajouter du contenu à ce qui est préexistant. Le fichier est alors ouvert en mode "*append*" et l'utilisateur complète les lignes déjà existantes, en saisissant à nouveau du texte jusqu'à écrire "*END"*

D'autre part, il peut décider de remplacer tout le contenu du fichier. Celui-ci est alors rouvert en mode "*write*", ce qui permet l'écrasement de toutes les lignes précédentes, qui sont remplacées par les nouvelles instructions entrées par l'utilisateur, de la même façon que lors de la création d'un nouveau fichier.

Après chaque modification, l'éditeur réaffiche le contenu, avec le code couleurs décrit précédemment, afin que l'utilisateur puisse vérifier les changements. L'utilisateur peut ensuite valider ou poursuivre la modification. Ce processus itératif assure une grande flexibilité : on peut éditer autant de fois que nécessaire avant de valider définitivement le contenu.

1.2.6 Interprétation d'un fichier *.draw* et exécution du fichier *.py* résultant

Pour finir, la troisième option du menu « Compile a draw file » déclenche la fonction compile(), qui :

1. Demande le nom du fichier .draw à interpréter.
2. Génère un chemin de sortie vers un fichier .py qui portera le même nom que le fichier .draw (ex. ../py_files_directory/<nom_fichier>.py).
3. Appelle la fonction interpret_draw_file(...) , définie dans interpreter.c et chargée de parser le fichier *.draw*. Le fonctionnement de cette fonction est décrit ultérieurement.
4. Informe l'utilisateur que le fichier Python correspondant vient d'être créé.

Suite à cet enchaînement d'actions et à l'appel du module traducteur du projet, une fenêtre turtle s'ouvre et affiche ce qui a été demandé par l'utilisateur.

1.2.6 Assistance sur les commandes fournies par la commande *help*

Afin d'aider l'utilisateur en cas d'oubli de la syntaxe de sa part, nous avons implémenté une commande *help*, que celui-ci peut taper à tout moment dans le terminal, même pendant l'écriture d'un fichier *.draw*, qui sera alors mis en pause.

L'utilisateur a deux options. D'une part, il peut taper seulement "*help*", et alors une liste de toutes les commandes possibles ainsi qu'un petit descriptif de chacune apparaît. D'autre part, il peut taper *"help <command_name>"*, pour obtenir des informations sur une commande en particulier, notamment sa description et les paramètres nécessaires à cette commande, avec leurs types.

```
=== MAIN MENU ===
1. Create a .draw file
2. Open a .draw file
3. Compile a file
4. Quit
Enter a number for your choice or type 'help' for assistance: help

=== HELP: AVAILABLE COMMANDS ===
- CURSOR : Create a cursor at a given position.
- COLOR : Change the color of a cursor.
- THICKNESS : Change the line thickness for a cursor.
- MOVE : Move a cursor forward by a certain distance.
- GOTO : Move a cursor to specific coordinates.
- ROTATE : Rotate a cursor by a given angle.
- LINE : Draw a line forward from the actual cursor position to a certain point according to the distance given.
- CIRCLE : Draw a circle.
- SQUARE : Draw a square.
- RECTANGLE : Draw a rectangle.
- POINT : Draw a point at the current cursor position.
- ARC : Draw an arc.
- ANIME : Animate a figure's movement : move it in a certain direction according to a given angle.
- SET : Assign a value to a variable.
- IF : Conditional statement: executes code if a condition is true.
- FOR : FOR loop: repeats an action a defined number of times.
- WHILE : WHILE loop: repeats an action while a condition is true.
- BREAK : Break out of the current loop.
- CONTINUE : Skip the rest of the current iteration in a loop.
- PASS : No operation placeholder.
- END : Ends the command input.
- help : Displays the list of available commands and their descriptions.
```

1.2.7 Récapitulatif

L'éditeur constitue la première étape de la chaîne de traduction du langage *draw++*. Il propose à l'utilisateur un moyen simple et intuitif de créer de nouveaux scripts .draw, les lire et les modifier avec une coloration syntaxique simple et pratique et de lancer la traduction en python.

La suite du rapport détaillera les mécanismes de parsing du fichier .draw qui s'effectuent au moment où l'utilisateur choisit l'option "Compile" du menu.

2. L'interpréteur et traducteur C

2.1 La structure globale du code

Le programme a été pensé et écrit de sorte à ce que sa structure soit la plus modulaire, claire et concise possible, pour faciliter la navigation parmi les différents composants de l'interpréteur de langage *draw++*. Les logiques des fonctions sont stockées dans les fichiers .c, tandis que leurs prototypes sont déclarés dans des fichiers de type *header* (.h). Ainsi, leur accès par d'autres fichiers est nettement facilité.

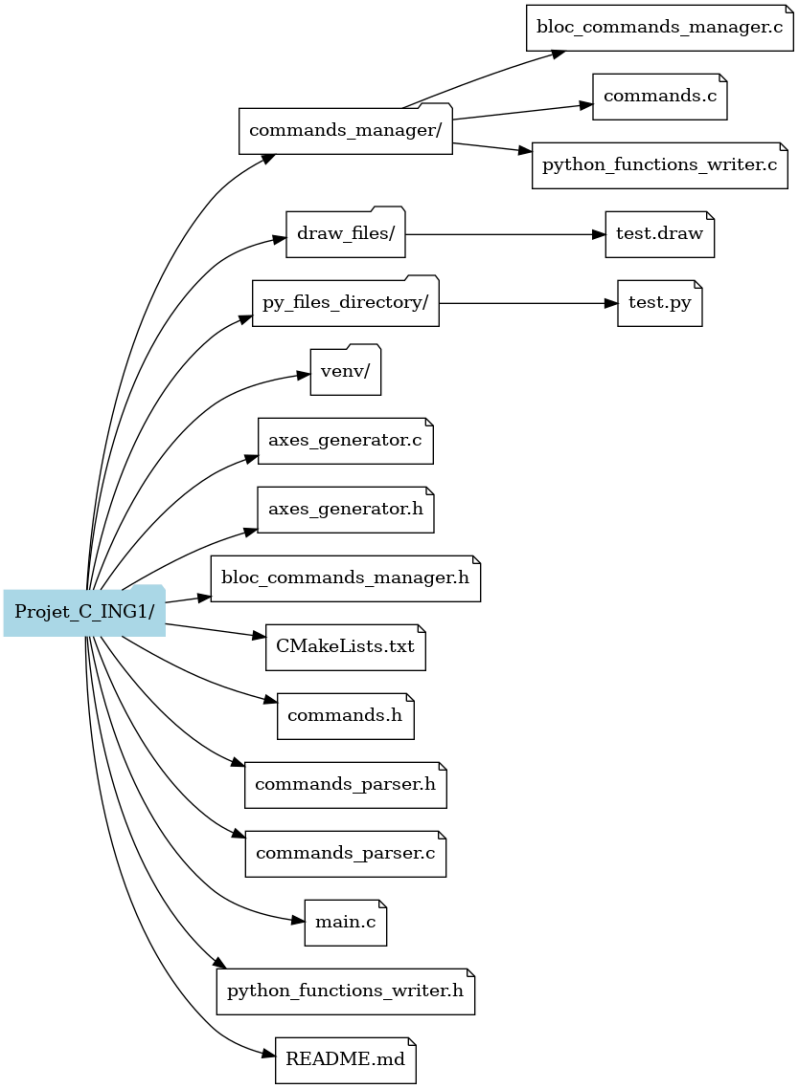
Lorsqu'on exécute le programme, c'est le fichier *main.c* qui est lancé et qui stocke toute la logique de l'éditeur et interface du projet. Tout fichier *.draw* créé à l'aide de l'éditeur est stocké dans le dossier *draw_files*.

Si par la suite l'utilisateur choisit de compiler un fichier *.draw* pour visualiser le résultat de ses commandes, ce sont les fonctions du fichier *commands_parser.c* qui prennent la main, pour analyser et parser les commandes dans le fichier *.draw*.

Une fois les commandes différenciées les unes des autres, ce sont les fonctions contenues dans les fichiers du dossier *commands_manager* qui sont appelées.

Pour créer les fonctions correspondantes aux commandes nécessaires dans le script Python, le code exécuté est stocké dans le fichier *python_function_writer.c*.

Pour ce qui est des appels à ces mêmes fonctions python, le code est réparti dans deux fichiers : commands.c pour les commandes simples, en une ligne, et *bloc_commands_manager.c*, pour les commandes en blocs telles que les boucles *for* et *while*. Tout fichier python écrit est finalement stocké dans le dossier py_files_directory.



2.2 La lecture et le parsing des fichiers *.draw*

Dans cette section, nous décrivons l'algorithme mis en place pour analyser (ou *parser*) le fichier .draw, délimiter les commandes les unes des autres, que ce soit une commande en ligne ou en bloc, et finalement exécuter le fichier python final une fois toutes les traductions de commandes effectuées.

Le fichier interpret.c contient l'essentiel de cette logique. Il s'articule principalement autour de trois fonctions majeures qui sont : *interpret_draw_file(), process_command()*, et *execute_command()*.

2.2.1 Première lecture du fichier .draw : *interpret_draw_file*()

La lecture et l'analyse du contenu du fichier .draw débute par l'exécution de la fonction *interpret_draw_file*, appelée au sein du fichier main.c. Cette fonction prend en paramètres deux chaînes de caractères, le nom du fichier .draw et le nom du fichier .py, qui a été créé au sein du code de l'éditeur.

La fonction débute par l'ouverture du fichier .draw en lecture et celle du fichier Python .py en écriture. Les pointeurs de fichier sont stockés dans des variables locales (*draw_file* et *python_file*), pour faciliter leur accès à toutes les autres fonctions du fichier

commands_parser.c. La fonction *set_python_file* est également exécutée pour écrire les imports nécessaires dans le nouveau fichier python et créer les variables globales, telles que le dictionnaire de curseur qui les gardera tous en mémoire.

Par la suite, nous parcourons le contenu du fichier *.draw* ligne par ligne grâce à *fgets*(), afin de repérer les commandes du langage *draw++* écrites par l'utilisateur et qui figurent dans le tableau commands[]. Chaque instruction reconnue est alors mémorisée dans la liste *detected_commands*[]. Cette première analyse permet d'associer immédiatement à chaque commande détectée dans le contenu du fichier *.draw* la fonction chargée de générer, dans le script Python, la définition d'une routine utilisant la bibliothèque *turtle* et dédiée à l'accomplissement de l'action associée à la commande. Le tableau commands[] associe en effet à chaque nom de commande une fonction de création Python, accessible par l'appel commands[name].create(python_file).

Nous avons fait le choix de définir une fonction Python distincte pour chaque commande demandée au moins une fois par l'utilisateur, pour limiter les redondances dans le code C et Python. À l'exécution du script python généré, il suffit alors de faire appel à la fonction correspondante avec les paramètres donnés en *draw++* par l'utilisateur pour réaliser l'opération attendue. Ainsi, lorsqu'un mot-clé comme "CURSOR" est rencontré, il est stocké dans la liste *detected_commands*, puis on appelle et exécute la fonction "*create_cursor_func_py*", qu'on a récupéré dans le dictionnaire *commands*[]. Cette fonction écrira dans le fichier Python le script de la fonction *handle_cursor* capable de créer des curseurs en fonction des paramètres donnés.

Après cette phase de mise en place du fichier Python, la fonction libère la mémoire attribuée au stockage des commandes détectées dans la liste *detected_commands[]*. De plus, elle referme le fichier .draw, car celui-ci ayant été parcouru entièrement, le curseur de lecture est arrivé à la fin du contenu et ne peut plus remonter aux lignes supérieures. Elle transmet toutefois son nom, ainsi que le fichier python qui lui est toujours en écriture, à la fonction *process_command*().

2.2.2 Traitement commande par commande : process_command()

La fonction *process_command()* est invoquée par *interpret_draw_file*() une fois les fonctions python nécessaires à l'exécution des commandes voulues par l'utilisateur créées. Elle commence par ouvrir le fichier .draw en mode lecture, pour pouvoir le lire de nouveau dès le début. Grâce à une boucle *while* et à *fgets()*, elle va alors séparer chaque commande et leurs paramètres respectifs les unes des autres.

Il existe deux types de commandes possibles, que la fonction doit différencier. Pour ce faire, elle cherche d'abord à détecter de potentielles instructions en bloc, dont le début est marqué par une accolade ouvrante {. Si ce signe distinctif est repéré, la variable inside block prend alors la valeur 1. Cela permet de garder en mémoire si l'on se trouve à l'intérieur d'un bloc (par exemple après IF { ... }). Un tampon (buffer) accumule ensuite toutes les lignes

jusqu'à la détection d'une accolade fermante }. Si c'est le cas, le contenu du tampon est finalement transmis d'un seul coup à *execute_command*().

En dehors de ces blocs, chaque ligne est immédiatement envoyée à *execute_command*(), ce qui permet de traiter séparément les commandes simples (une seule ligne) et celles regroupées sous forme de blocs.

2.2.3 Compilation des commandes : execute_command()

Une fois qu'une commande et ses paramètres ont été isolés, la chaîne de caractères contenant ces informations est transmise en tant que paramètre à la fonction *execute_command*(). Celle-ci commence par extraire séparément le nom de la commande et ses paramètres en utilisant la fonction *sscanf*(). Ensuite, une fois stockés dans des variables, le programme compare le nom de la commande détectée avec la liste des commandes contenue dans le tableau commands[]. S'il trouve une correspondance, il récupère la fonction chargée d'écrire l'appel aux méthodes créées préalablement dans le fichier python et l'exécute, générant ainsi le code Python équivalent à la commande draw++.

2.2.4 Finalisation : *finalize_execution*()

Une fois toutes les commandes traduites en script Python, la fonction *finalize_execution*() procède au lancement de la chaîne d'actions *turtle* puis à la fermeture du fichier *.draw* ainsi que du fichier Python *.py*. Elle libère exécute enfin le fichier Python généré à l'aide de *system(command),* qui cette commande prend généralement la forme suivante : *py python_filename*. Une fois cette étape achevée, l'utilisateur peut immédiatement visualiser le dessin ou l'animation résultant de la traduction de son fichier *.draw.*

2.3 Les scripts Python résultant de l'interprétation des fichiers *.draw*

La génération des scripts Python s'appuie sur trois composants principaux : le fichier *python_functions_writer*.c, qui définit dans le code Python les fonctions nécessaires à l'exécution des commandes, le fichier *commands*.c, qui gère la traduction des commandes *draw++* simples en appels à ces scripts Python ,et enfin le fichier *bloc_commands_manager*.c, qui gère les instructions plus complexes telles que les conditions (IF), les boucles (FOR, WHILE) et les commandes de contrôle de celles-ci (BREAK, CONTINUE, PASS). L'ensemble de ces modules produit un script Python autonome, capable de tracer des formes avec la bibliothèque turtle en fonction de ce qui a été indiqué par l'utilisateur.

2.3.1 commands.c

Le fichier commands.c stocke un tableau *commands*[], où est stockée la liste des commandes acceptées par le programme. Chaque élément associe un nom de commande (par

exemple "MOVE", "LINE", "IF", etc.) à deux fonctions : une fonction de compilation (*call_...*) et, s'il y a lieu, une fonction de création (*create_...*). Lorsqu'une ligne du fichier *.draw* est reconnue comme correspondant à l'une de ces commandes, la fonction *create_* associée dans le dictionnaire est appelée pour créer le script Python global, via des appels *fprintf()*, puis la fonction *call_* est invoquée et écrit directement, la ou les appels Python à insérer dans le script. Ainsi, *call_move_func_py* analyse les paramètres de la commande donnés en arguments (identifiant du curseur, distance à parcourir) puis génère un appel à *handle_move*(...) avec les résultats de ces analyses du côté Python. Il en va de même pour *call_color_func_py, call_circle_func_py,* etc., chacun récupérant au préalable les paramètres de la commande dans la ligne du fichier *.draw* (par exemple la couleur, le rayon d'un cercle, ou les coordonnées de destination).

Pour la plupart des commandes de dessin (lignes, cercles, rectangles, points), le code se contente d'appeler les fonctions préalablement créées dans le script Python, comme *handle_line*(...) ou *handle_circle*(...). En revanche, la commande ANIME (gérée par *call_animation_func_py*) insère un bloc de code Python plus étoffé, comprenant une boucle *for i in range(...)*, des appels à *time.sleep(...)* et un usage avancé de *turtle.tracer(...)* pour gérer l'animation en temps réel. Cette particularité illustre la flexibilité du mécanisme : selon la complexité de la commande *draw++*, la portion de script Python générée peut être très simple (une unique ligne) ou nettement plus élaborée.

2.3.2 python_functions_writer.c : définition des fonctions Python

Si *commands.c* inscrit dans le script l'appel de la fonction Python adéquate, c'est dans *python_functions_writer.c* que sont réellement définies ces fonctions. La fonction *set_python_file(...)* initialise d'abord le contexte Python, en important turtle, time et math, puis en configurant la fenêtre graphique (taille, titre), le dictionnaire *cursors* qui stocke tous les curseurs *turtle*, et les listes *shapes* et *animation_shapes* où sont mémorisés les figures (lignes, cercles, etc.). En fin de traitement, *end_python_file*(...) ajoute l'instruction *turtle.done()*, qui garantit que la fenêtre reste affichée une fois le dessin terminé.

Chaque commande *create_...* (par exemple *create_move_func_py*, *create_circle_func_py* ou *create_rectangle_func_py*) sert à déclarer une fonction Python correspondant à l'instruction *draw++* voulue. Ainsi, *create_cursor_func_py* définit en python *handle_cursor(...)*, qui crée un nouveau curseur à l'aide de *turtle.Turtle()*, lui associe un identifiant et le place en coordonnées *(x, y)*. De la même manière, *create_line_func_py* introduit *handle_line(...)*, chargée de tracer une ligne à partir de la position courante du curseur, puis d'enregistrer cette forme dans la liste *shapes*. Chaque fonction Python ainsi générée reçoit l'identifiant du curseur à utiliser, les éventuels paramètres nécessaires (longueur, rayon, angle…), et s'occupe du tracé avec *turtle*. Il devient alors facile pour le code C, dans *commands.c*, d'implémenter la commande demandée en une simple instruction *fprintf()*.

Plus complexe que les autres commandes, *create_animation_func_py* définit la fonction *animation(...),* qui permet de manipuler plusieurs figures à la fois et calcule leur déplacement

via des transformations géométriques (par exemple une translation en *dx* et *dy*, dans une direction donnée par un certain angle, combinée à l'utilisation de *tracer*() pour masquer le tracé temporaire). Cette approche, qui repose sur le stockage de chaque forme dans shapes avec un identifiant, une position et un type (ligne, cercle, carré…), permet d'animer, et plus particulièrement de déplacer efficacement le contenu de la fenêtre *turtle*.

2.3.3 bloc_commands_manager.c : logique conditionnelle et boucles

Certaines commandes *draw++* forment des structures plus avancées et complexes, ce sont des commandes en bloc comme IF, FOR ou WHILE. Dans *bloc_commands_manager.c,* chaque fonction analyse la syntaxe du bloc transmis, détecte d'éventuelles conditions sur les coordonnées du curseur , puis écrit la syntaxe Python adéquate (if … :, else :, for … in range(...) :, while … :) dans le fichier final. Les blocs compris entre accolades { … } sont traités en réutilisant la fonction *execute_command(...)* pour chaque sous-commande, qui traitera la sous-commande comme étant une simple commande en ligne isolée. Il faut toutefois veiller à séparer les sous-commandes par des virgules pour que le parseur sache correctement les identifier.

Les fonctions handle_break_python(...), handle_end_python(...), handle_continue_python(...) se chargent, quant à elles, d'insérer la ligne break, continue ou pass au bon endroit si écrite par l'utilisateur dans le fichier .draw.

2.4 La gestion des erreurs implémentée

2.4.1 Dans l'éditeur

Le fichier *main.c* intègre une gestion des erreurs systématique afin d'assurer la robustesse de l'application. Ces vérifications se manifestent principalement lors de l'accès aux fichiers et répertoires, ainsi que dans le traitement des entrées utilisateur. Par exemple, avant de créer un fichier *.draw*, le code vérifie si le fichier peut être créé et ouvert en écriture ; en cas d'échec, un message d'erreur explicite est affiché. De même, l'ouverture d'un fichier ou la lecture d'un répertoire est accompagnée de précautions : si le répertoire ne peut être accédé ou si le fichier est introuvable, l'utilisateur est informé immédiatement et l'action est interrompue. Par ailleurs, le programme s'assure que les choix dans le menu principal sont valides, en signalant toute option incorrecte tout en guidant l'utilisateur vers les commandes disponibles.

2.4.2 Dans le fichier parseur de commandes.

La gestion des erreurs dans ce fichier couvre principalement les aspects liés à la manipulation des fichiers, à l'exécution des commandes et à la gestion des blocs structurés. Ainsi, lorsqu'un fichier *.draw* ou Python doit être ouvert, le code vérifie systématiquement si l'opération est possible. Si l'ouverture échoue, un message d'erreur clair est affiché directement dans le terminal, qui constitue l'interface de l'utilisateur, et l'exécution de la

fonction concernée est interrompue. L'utilisateur retourne donc à l'étape du choix d'option dans le menu principal.

Pour l'étape de détection des commandes dans le fichier, que ce soit pour la création ou pour l'appel des fonctions Python dédiées, si une commande est inconnue ou mal formée, elle est signalée à l'utilisateur via un message dans le terminal et son exécution est ignorée. Par ailleurs, les blocs de commandes délimités par des accolades font l'objet d'une attention particulière : une allocation ou une réallocation de mémoire défaillante lors de la création du tampon (*buffer*) interrompt immédiatement l'opération, et un avertissement est affiché dans l'interface lorsque qu'un bloc ouvert n'est pas correctement fermé.

Enfin, après la génération du fichier Python, une tentative d'exécution est réalisée via une commande système. Toute erreur lors de cette étape est détectée, et un message explicite informe l'utilisateur dans le terminal de l'échec ou, au contraire, du succès de l'exécution. Ces différentes mesures garantissent un traitement rapide et fiable des fichiers et des commandes, dès les premières étapes du processus tout en prévenant et en informant l'utilisateur de problèmes potentiels. Il n'a donc pas été nécessaire d'implémenter de la gestion d'erreurs dans les fichiers du dossier *commands_manager*.

3. Exemples d'exécution

3.1 Exemple 1 : Ouverture et modification d'un fichier *.draw*

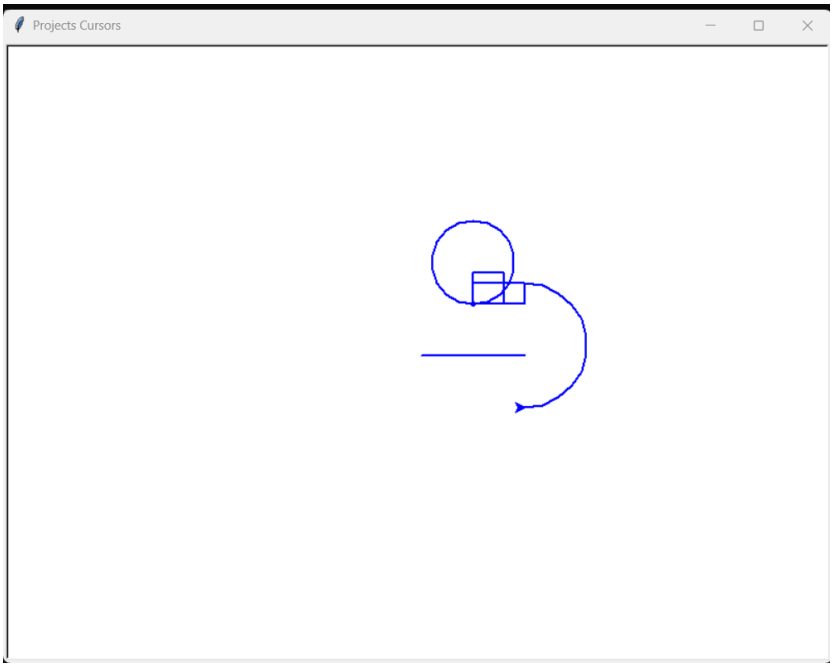
```
=== MAIN MENU ===
1. Create a .draw file
2. Open a .draw file
3. Compile a file
4. Quit
Enter a number for your choice or type 'help' for assistance: 2
=== List of available files ===
Files in directory 'draw_files':
- test.draw
- test1.draw
- test2.draw
- test3.draw
- test4.draw

Enter the name of the file to open (without extension): test

=== Content of file test ===
CURSOR  1  100  50  TRUE


Would you like to modify this file? (y/n): y

=== Modifying file test ===
Options:
1. Append content
2. Replace all content
Your choice: 1
Enter the content to append (type 'END' to finish, or 'help' for assistance):
> COLOR 1 RED
> END

=== Current content of file test ===
CURSOR 1 100 50 TRUE

COLOR 1 RED

Are you satisfied with this modification? (y to confirm, n to continue): y
Modifications confirmed.

=== MAIN MENU ===
1. Create a .draw file
2. Open a .draw file
3. Compile a file
4. Quit
Enter a number for your choice or type 'help' for assistance:
```

3.2 Exemple 2

Fichier .draw :

```
CURSOR 1 0 0 TRUE
COLOR 1 blue
THICKNESS 1 2.5
LINE 1 100 1    # Ligne de 100 pixels
GOTO 1 50 50    # Déplacement à la position (50, 50)
CIRCLE 1 40 2   # Cercle de rayon 40
SQUARE 1 30 3   # Carré de côté 30
RECTANGLE 1 50 20 4  # Rectangle de largeur 50 et hauteur 20
POINT 1 5       # Point

SET VARIABLE VARX = 100
SET VARIABLE VARY = -50
GOTO 1 VARX VARY  # Déplacement à une position calculée avec des variables

ARC 1 60 6       # Arc de rayon 60
```

Résultat de l'exécution :



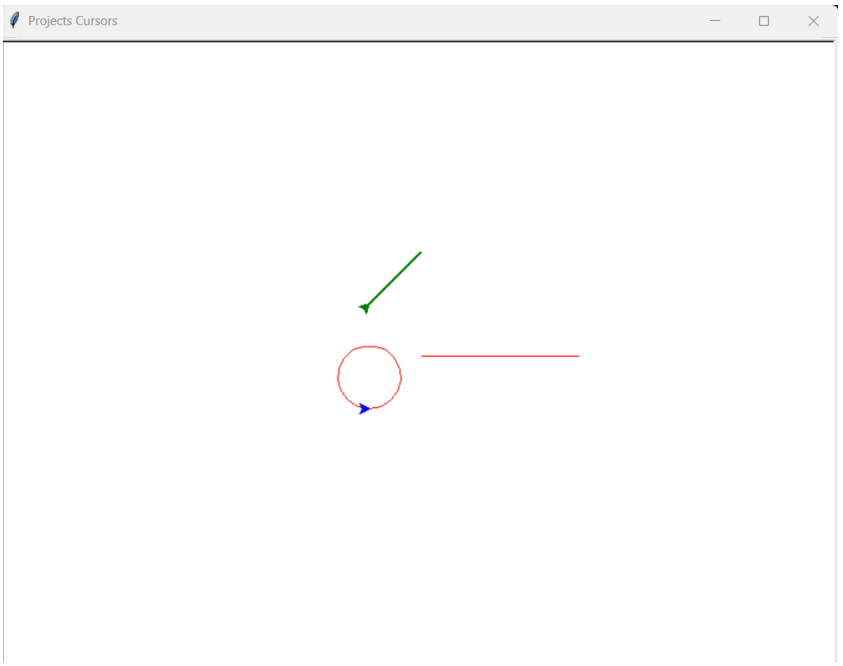
Notez que le centre du repère de turtle est le centre de la fenêtre

3.3 Exemple 3

Fichier .draw :

```
CURSOR 1 0 0 TRUE
CURSOR 2 -100 50 TRUE

COLOR 1 red
COLOR 2 green
THICKNESS 1 1.0
THICKNESS 2 2.0

LINE 1 150 1
GOTO 1 -50 -50
CIRCLE 1 30 2

MOVE 2 50
ROTATE 2 45
LINE 2 70 3

IF CURSOR WITH ID = 1 IS IN POSITION X > 0 {
    COLOR 1 yellow,
} ELSE {
    COLOR 1 blue,
}
```

Résultat de l'exécution :



Conclusion

Ce projet a permis de concevoir et de développer un langage de programmation *draw++*, offrant des possibilités simples pour la création de dessins et d'animations à l'aide de commandes structurées. L'ensemble des objectifs initiaux a été atteint, incluant la mise en place d'un éditeur fonctionnel, d'un compilateur robuste et d'un traducteur générant des scripts Python exploitables. Les fonctionnalités implémentées, telles que les commandes conditionnelles, les boucles et les animations, mettent en évidence la flexibilité et l'intégration cohérente des concepts de programmation dans ce langage adapté à des utilisateurs variés.

Cependant, plusieurs pistes d'amélioration pourraient être envisagées pour enrichir encore davantage *draw++* et répondre à des besoins plus avancés. Parmi elles :

- Rotation des figures avec la commande ANIME : cette fonctionnalité permettrait d'étendre les possibilités d'animation en autorisant des animations plsu avancées.
- Ajout de nouvelles conditions (basées sur la couleur, l'épaisseur, etc.) dans les blocs IF et WHILE : cela offrirait plus de contrôle à l'utilisateur
- Gestion des boucles imbriquées : cette amélioration accroîtrait la complexité des scénarios possibles, en permettant la combinaison de plusieurs niveaux de boucles pour des scénarios plus avancés.

Ces évolutions permettraient d'étendre les capacités du langage draw++ pour les rapprocher de ce qu'il est possible de faire en Python, tout en offrant à ses utilisateurs une expérience encore plus personnalisée dans la création de contenus graphiques.