DURAND Clovis
MAGNAN Camille

12 Mai 2017

PG208

PG208
Dessin vectoriel

V1

ENSEIRB-Matmeca
Bordeaux - Talence

Table des matières

Table des figures

1 Introduction

L'objectif de ce projet est de mettre en œuvre les notions de base de la programmation orientée objets appliquées au langage C++ que l'on a appréhendées durant l'enseignement de PG208. Pour ce faire nous avons décidé de nous lancer dans le second sujet de projet : **Dessin vectoriel**. L'objectif est de générer une image Bitmap en fonction d'un fichier de description vectoriel de formes. L'enjeu de ce sujet est de comprendre l'intérêt de l'utilisation de classes et d'apprendre à faire interagir les différentes classes entre elles.

Cette application est appelé `dessin_vect`

1.1 Gestion du cahier des charges

Nous avons donc tenté de réaliser une application permettant de suivre au mieux le cahier des charges qui nous a été soumis. En effet, nous avons développé un programme permettant de générer des images à l'aide d'une description vectorielle de ces images. Ces images seront créées à partir de la descriptions de plusieurs formes géométriques. Nous avons donc programmé le dessin de 10 formes différentes afin d'avoir un large panel de possibilités graphiques : la ligne, le triangle vide, le point, le cercle (vide et plein), le rectangle (vide et plein) et enfin le carré (vide et plein).

Le cahier des charges stipule que nous devons gérer un facteur d'échelle qui modifie la résolution de notre image de sortie. Un facteur de transparence doit aussi être intégré lorsqu'on dessine une forme.

Voici le diagramme de cas d'utilisation qui représente le mode d'opération du point de vue du client :

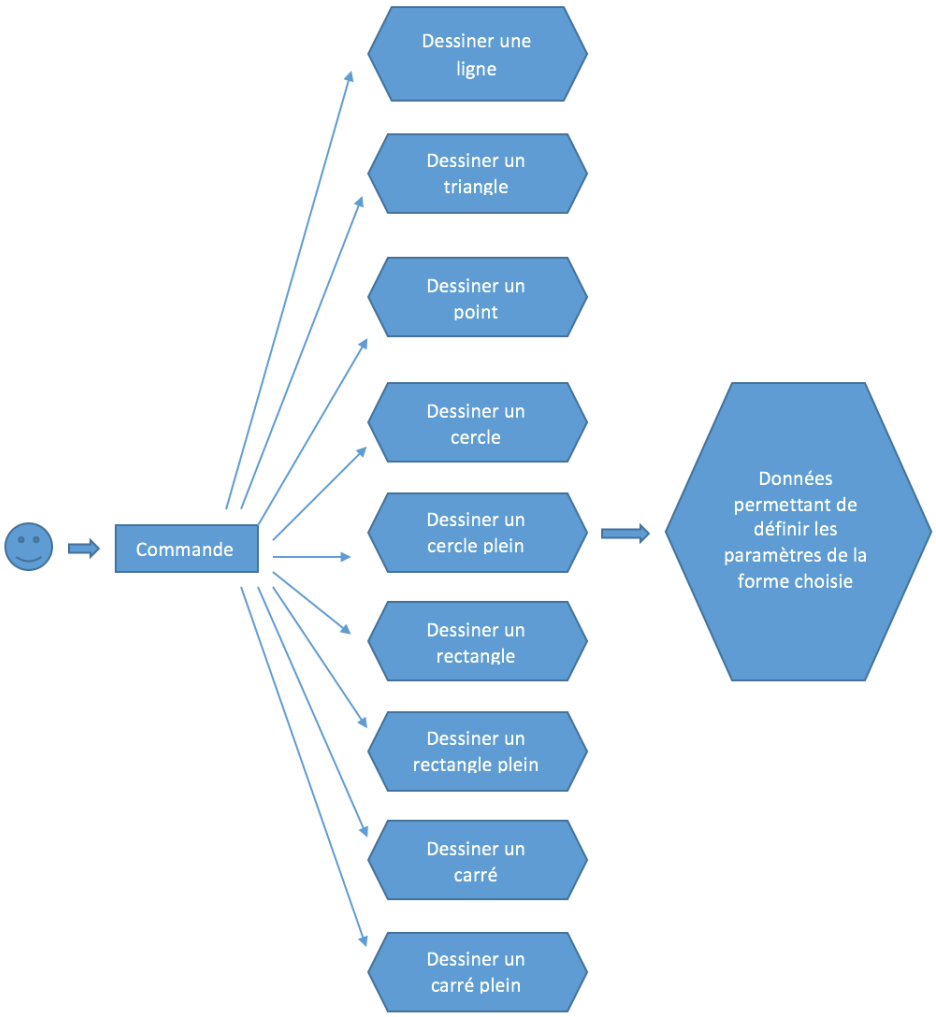

FIGURE 1 – Diagramme de cas d'utilisation de notre programme `dessin_vect`

1.2 Diagramme UML

Le langage UML est un langage de modélisation graphique à base de pictogrammes conçu pour fournir une méthode simple pour visualiser la conception d'un système. Nous avons donc utilisé ce langage afin de nous de nous permettre de suivre un cheminement clair et détaillé de notre projet. Ce diagramme permet aussi de comprendre comment les différentes classes ont été construites les unes par rapport aux autres, comme les rapports d'héritage.

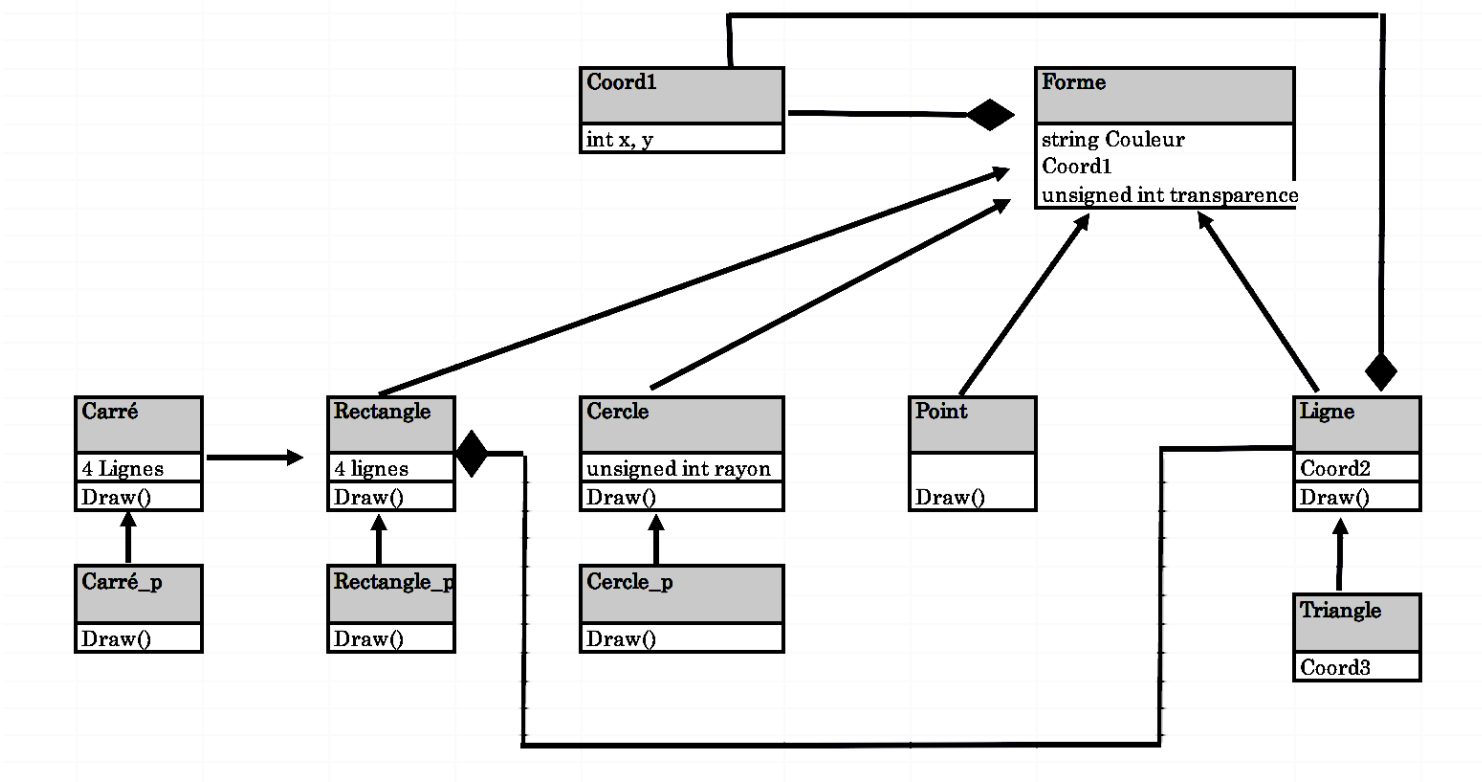

FIGURE 2 – diagramme UML

2 Les classes principales

Grâce au diagramme UML vu précédemment, nous avons pu commencer à écrire le code de notre projet par les classes principales. En effet certaines des classes à programmer sont plus importantes que les autres. Nous avons fait le choix de créer une classe de coordonnées Coord (non indispensable mais qui nous a permis de nous approprier la notion de position d'un objet dans l'image dès le début) qui rassemble les coordonnées x et y de chaque forme.

Nous avons commencé par concevoir la classe forme contient tous les attributs communs aux futures classes des différentes formes. Entre autres :

— Coord m_c1 : Cette coordonnée définit la position dans l'image de l'objet. Ce point sert de base pour construire le reste de la forme.

— string m_couleur : La couleur de l'objet. Tous les objets à dessiner doivent avoir une couleur définie, de manière à écrire cette couleur sur un pixel de l'image. Pour ce faire un format RGB est utilisé (0 à 255 pour chacune des trois couleur primordiales).

— unsigned int m_transparence : Ce facteur permet de définir le niveau de transparence de la forme sur le dessin.

Ensuite, nous avons écrit le reste des classes de formes, en commençant par la classe Ligne. Celle-ci est à la base de beaucoup de formes. Par exemple, la méthode `drawLigne()` est utilisée dans toutes les formes (mis à part la classe Cercle et la classe Cercle_p).

Enfin une classe Dessin permet de réunir toutes les formes en un seul objet, de définir les bornes de l'image, et de fournir les informations au dessin de l'image.

2.1 Classe de coordonnées `Coord`

La classe `Coord` est une classe pratique. Celle-ci ne génère pas une forme, mais permet simplement de généraliser les 2 coordonnées d'un point en un seul objet. Cet objet servira d'attribut pour la classe Forme, ainsi que pour d'autres formes.

2.2 Classe forme `Forme`

La classe mère `forme` regroupe l'ensemble des attributs communs à toutes les classes de formes :

— Les coordonnées (x,y) `Coord const c1`
— La couleur `string const c`
— La transparence `string const c`

Elle regroupe également les méthodes utilisées dans les différentes formes, c'est à dire la méthode générant la couleur des pixels en fonction des valeurs de R, G et B. De cette façon il nous est possible de récupérer la valeur de R, G et B indépendamment afin de déterminer la couleur de notre pixel. Nous avons rajouté une fonction `DrawPixel` dans la classe CImage qui nous permet de dessiner chaque pixel grâce à un simple appel à cette fonction. Cette classe nous permet également de déterminer le réglage de couleur du pixel en fonction du niveau de transparence retenu grâce à la formule suivante :

$$Pixel'(x,y) = \frac{(100 - transp) \times Pixel(x,y) + transp \times CouleurForme}{100}$$

2.3 Classe Dessin `Dessin`

A REMPLIRRRE

2.4 Classe Ligne `Ligne`

Nous avons décidé de tracer nos lignes à l'aide de la méthode de Bresenham. Cette méthode permet d'identifier le sens du tracé de la ligne et de déterminer les pixels que l'on doit dessiner afin de s'approcher le plus près possible d'une ligne droite (avec un taux d'erreurs le plus faible possible). Pour celà, nous séparons l'écran en octants. Nous partons d'un point central, puis nous prenons le deuxième point de la ligne et nous calculons les deltas de la droite reliant ces deux points. Grâce à ceux-ci nous determinons dans quel octant nous nous trouvons et appliquons ainsi l'algorithme de Bresenham dans cet octant.

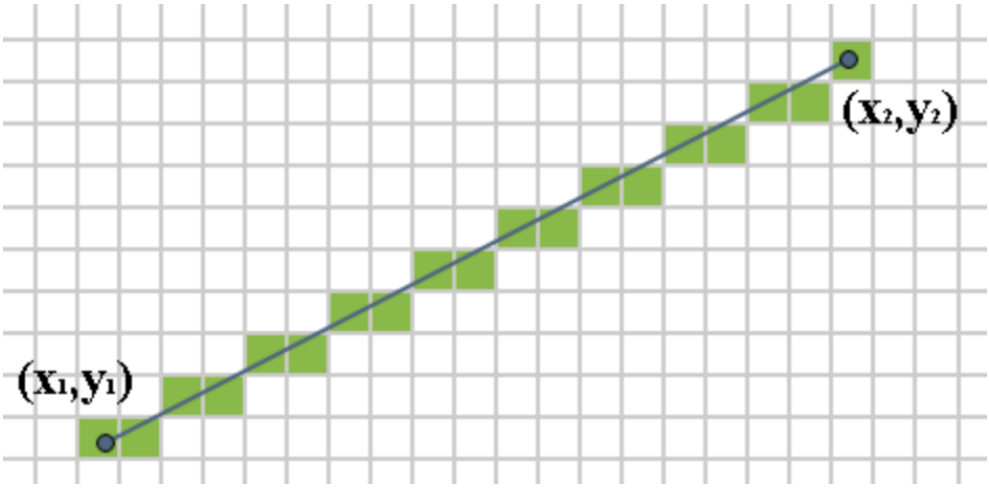

FIGURE 3 – Schéma explicatif de la méthode de Nreenham

3 Classes filles `Formes diverses`

3.1 Le Point `Point`

3.2 Le Cercle `Cercle & Cercle_p`

3.2.1 Cercle vide

De son côté le cercle est défini de la façon suivante : CERCLE (C, RAYON, COULEUR, TRANSPARENCE). Nous utilisons ensuite une condition de longueur, telle que si l'on se trouve à une distance `RAYON` du centre, alors le pixel est dessiné. Afin d'avoir un tracé suffisement large à l'oeil humain, nous avons ajouté un "epsilon" de sorte que le pixel soit dessiné si il se trouve dans l'intervale `RAYON +- EPSILON`

3.2.2 Cercle plein

Le cercle plein est dessiné d'une façon similaire, sauf que la condition ne porte pas sur un intervale mais seulement sur une infériorité. En effet, on dessine le pixel si il est à une distance inferieure à celle du rayon du centre du cercle.

3.3 Le Rectangle `Rectangle & Rectangle_p`

3.3.1 Rectangle vide

Le rectangle vide est tout simplement créé à partir de 4 lignes. Ces lignes trouvent leurs coordonnées en fonction du point de départ (coin en bas à gauche du rectangle) et des longueurs et hauteurs.

3.3.2 Rectangle plein

Le rectangle plein fonctionne à partir de lignes également. En effet, le remplissage se fait en balayant le rectangle dans la hauteur et en dessinant une ligne à chaque incrémentation.

3.4 Le Carré `Carre & Carre_p`

3.5 Le Triangle `Triangle`

Tout comme le rectangle vide, le triangle est créé à partir de 3 lignes.

3.6 Aboutissement

4 Dimensions graphiques

JE COMPRENDS RIEN