UPEM - ESIEE PARIS

M2 SIS - Projet Patchwork

NACER Tabib et TOFFANIN Marc

16 avril 2016

TABLE DES MATIÈRES

1 INTRODUCTION

1.1 CAHIER DES CHARGES

Le projet PATCHWORK est un projet proposé en 2016 en cours de c++ consistant à simuler des interactions entre une maîtresse et ses élèves. Ce projet s'effectue en language C++11. Les interactions sont simulées par des clients (les élèves) qui se connectent à un server (la maîtresse). Le but des élèves et des créer un dessin constitué de formes géométriques simples et de l'envoyer à leurs maitresse. Si elle trouve les dessins suffisants elle créer un patchwork, c'est-à-dire qu'elle les assemble. Dans le cas contraire, elle peut renvoyer un dessin à un élève avec une annotation pour qu'il fasse les modifications nécessaires. [1]

Le projet se feras en mode console, avec la possibilité d'afficher les dessins en créant une fenêtre. De plus l'application doit répondre à un certain nombre de contraintes qui sont les suivantes :

- Les dessins sont constitués de formes géométriques élémentaires et colorées : lignes, polygones, cercles, ellipses.
- Les calculs du périmètre et de l'aire sont requis. L'aire d'un polygone quelconque peut être calculée par triangulation.
- Les formes géométriques peuvent se transformer par homothétie, se déplacer par translation, par rotation, par symétries centrale et axiale.
- Une image peut également contenir des images de plus petite taille (aire).
- On souhaite pouvoir ordonner les formes selon plusieurs critères : leur périmètre, leur aire et leur distance à l'origine. Les relations d'ordre devront utiliser la notation <=.
- Aucun doublon de forme géométrique ne peut exister
- Le nombre total de chaque forme géométrique de la grande fresque murale doit pouvoir être déterminé afin de procéder à des statistiques (histogramme, fréquence). Il en est de même pour les couleurs.

Une suite de test devras être mise en place pour pouvoir effectuer des tests de régression facilement.

1.2 CHOIX ET DÉCISIONS

Dans tout projet il faut parfois prendre des décisions même sans en référer au client. Cette section va détailler les différents choix qui ont été effectués, sans pour autant rentrer dans les détails d'implémentation.

Tout d'abord la partie réseau de l'application, c'est-à-dire la communication entre le server et les clients se fait en TCP (Transmission Control Protocol) ce qui permet de faire des connections persistentes et de ne pas avoir à gérer le découpages et l'assemblage des paquets que provoquerais l'utilisation du protocol UDP. L'outil retenu pour faire celà est $boost :: asio$ [2] . Ce choix a été fait pour deux raisons principales :

- Cette bibliothèque est totalement crossplateforme (elle peut se compiler sous diffé-

rents environnements). Elle permet donc de faire tourner le projet sous Windows (via Visual Studio) ou sous Linux (testé sous Ubuntu 14.0 LTS).
— Cette bibliothèque va peut-être rentrer dans le standard C++x0, c'est en pourparler. Celà en fait une bibliothèque intéressante à apprendre, de plus elle propose le pattern Proactor pour la gestion asynchrone du réseau de base.

Dans les spécifications, un affichage de l'image n'est pas requis cependant il a été décidé de rajouter cet affichage grâce à la bibliothèque *SDL2* (Simple DirectMedia Layer) [3], qui de la même manière que $boost :: asio$ est crossplateforme et facile d'utilisation. Une décision qui pourrait être discutée à été de stocker les vecteurs et de faire les opérations en float. Ceci pose des soucis au niveau de l'affichage, quand on discrétise les formes. Cette branche est appellé $DigitalGeometry$ et s'étend au delà du scope du projet.

De la même manière, un choix qui peut être discuté a été de ne pas définir la taille de l'image. Laissant ainsi l'utilisateur utiliser l'infinité de R^2 pour placer ses figures. C'est seulement au moment d'afficher que l'image est adaptée pour correspondre à la taille de la fenêtre d'affichage.

La composante mathématique du projet (à savoir les opérations vectorielles en deux dimensions) ont été implémentées à la main, sans utiliser de bibliothèques.

Enfin, pour la gestion des fichiers, l'ensemble des définitions des formes géométrique est dans un header, rendant cette "bibliothèque" header only. Il est donc simple de l'ajouter à un projet. De plus le projet Client et Server n'ont qu'un cpp par préférences personnelles.

1.3 Limites du projet

Le projet a dans l'ensemble été implémenté dans son entiereté, comme nous allons le montrer dans la section suivante. Cependant certaines limites aparaissent, par exemple la rotation des ellipses est impossible dû à notre modèle d'ellipse. Les tests ont été effectués en locale il n'y as donc aucune garantie de bon fonctionnement en utilisation réelle.

Certaines contraintes n'ont pas été respéctées, notamment celles sur les doublons et les relations d'ordres.

1.4 Mode d'emploi

L'application est simple à utiliser. Il faut lancer le serveur qui va ensuite attendre la connection des clients. Quand un client se lance, il essayes de se connecter au serveur, cependant il est possible de créer une image en mode hors-connection. Au sein du client ou du serveur, un ensemble de commandes sont proposées et permettent d'effectuer diverses opérations. Ces commandes peuvent être affichées avec la commande $help$. Un exemple d'utilisation de

l'application peut être trouvé en ligne. [4]

Pour compiler l'application, deux options sont possibles :

— Sous Windows, il faut utiliser Visual Studio 2013 car les DLL fournies ont été compilées pour cette version. Il est possible d'utiliser l'application avec d'autres versions, mais la compilation de $boost :: system$ est nécessaire et laissée au lecteur.
— Sous Linux, il faut installer les bibliothèques SDL2 et Boost sur son sytème et ensuite éxecuter le Makefile via la commande $make$.

2 Architecture du code

Le code est composé de trois projets : le Server, le Client et Shapes_tests. Le premier définis le serveur, le second le client et le dernier est une suite de tests sur l'utilisation des formes géométriques. Chaque projet est détaillé dans cette section. Chaque classe sera détaillées dans la prochaine section 3 $Détaild'implémentation$, ici seulement le diagramme des classes est présenté.

2.1 Réseau asynchrone

Avant de parler des projets en eux-mêmes, une petite explication sur les réseaux asynchrones s'impose [5].

Les réseaux synchrones de bases effectuent les opérations séquentiellement, c'est-à-dire qu'une opération a la forme :

— Le programme appelle le socket
— Le socket fait passer l'opération au service
— Le service demande au système d'exploitation d'effectuer l'opération
— Le système d'exploitation renvois le résultat de l'opération
— Le résultat est renvoyé jusqu'au socket.

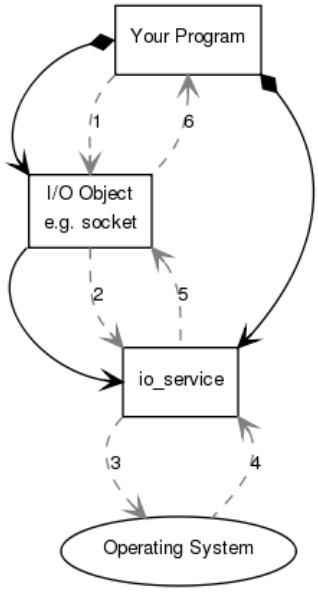

FIGURE 2.1 – Diagramme d'une opération synchrone

La demande d'opération, l'opération et le résultat se font donc séquentiellement. Cepdendant, dans le cas asynchrone c'est un peu différent.

— Le programme demande au socket de faire une opération asynchrone, et lui donne une fonction à éxécuter à la complétion
— Le socket fait passer l'opération au service
— Le service demande au système d'exploitation d'effectuer l'opération de manière asynchrone
— Une fois les opérations demandées, le système d'exploitation les effectues à sa guise, et place le résultat dans une queue que le service récupère
— Le service va en permanence regarder la queue pour voir si des opérations ont été finies quand ont l'appel
— Enfin le service dequeue les résultats et nous les renvois

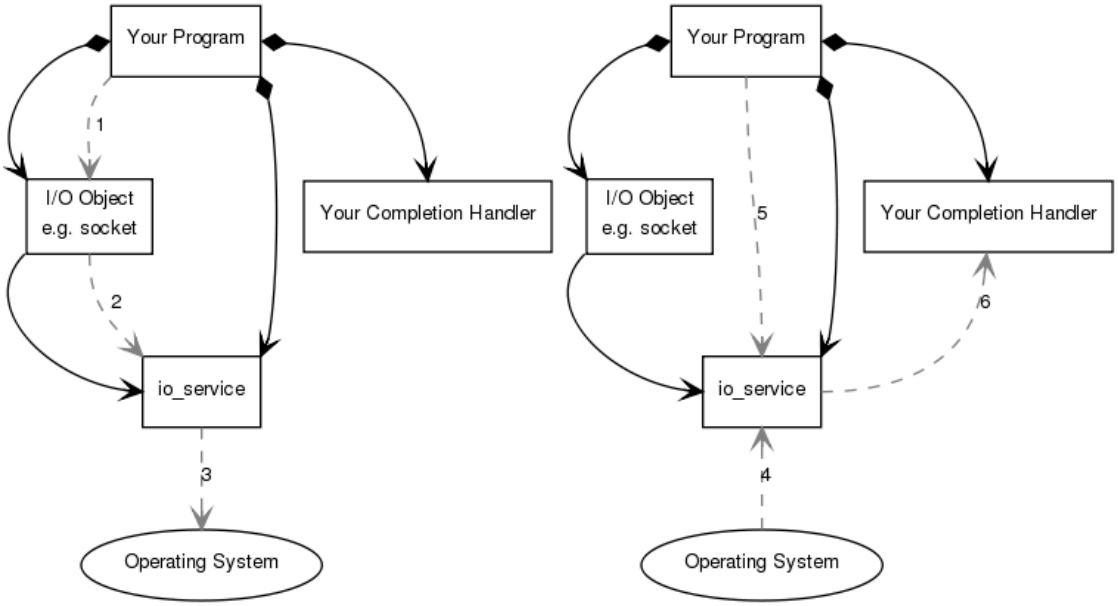

FIGURE 2.2 – A gauche le diagramme d'une demande d'opération asynchrone et à droite la manière dont le résultat est renvoyé.

La demande d'opération, l'opération et le résultat sont donc fait séparément au cours du temps. Il peut théoriquement y avoir un temps assez long entre la demande et le résultat d'une opération. Ce design est puissant, mais il est compliqué de mettre en place une suite de tests dessus étant donné que les opérations ne sont pas sequentielles, on ne peut donc pas tester de manière robuste à un instant T donné un résultat.

Le serveur et le client utilise ce design pour effectuer leurs opération réseau.

Il est important de noté que la partie boost : :asio est très inspirée de l'example en ligne de chat asynchrone [6].

2.2 LE SERVER

Le projet Server est responsable pour la création de la connection serveur, ainsi que le polling des commandes de l'utilisateur (la maîtresse). Ce projet est donc constitué de deux threads : un pour les commandes et un pour la partie réseau.

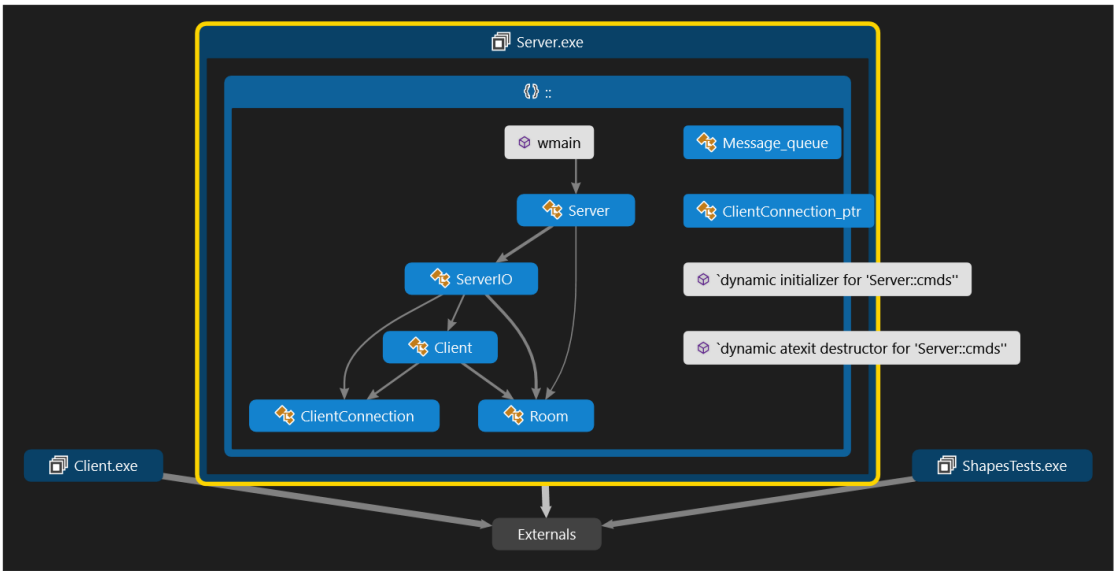

FIGURE 2.3 – Diagramme des dépendances du projet Server

2.3 LE CLIENT

Le projet Client est responsable pour la création du client, c'est à dire la connection au serveur ainsi que le polling des commandes de l'utilisateur (un élève). Ce projet est donc constitué de deux threads : un pour les commandes et un pour la partie réseau.

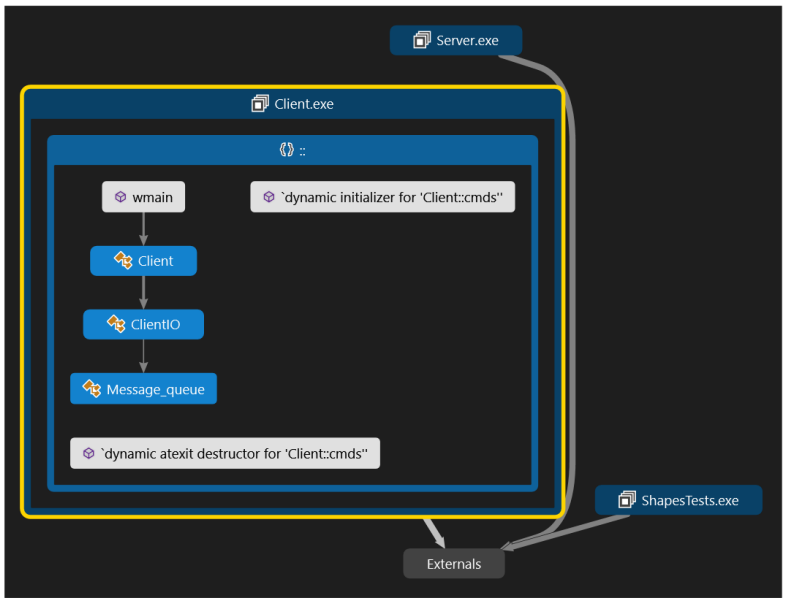

FIGURE 2.4 – Diagramme des dépendances du projet Client

2.4 Les tests

Le projet de tests va juste effectuer des tests sur les formes géométriques. Il affiche au fur et à mesure les résultats. Il n'est pas constitué de classes mais uniquement de fonctions qui vont tester chaque forme indépendemment.

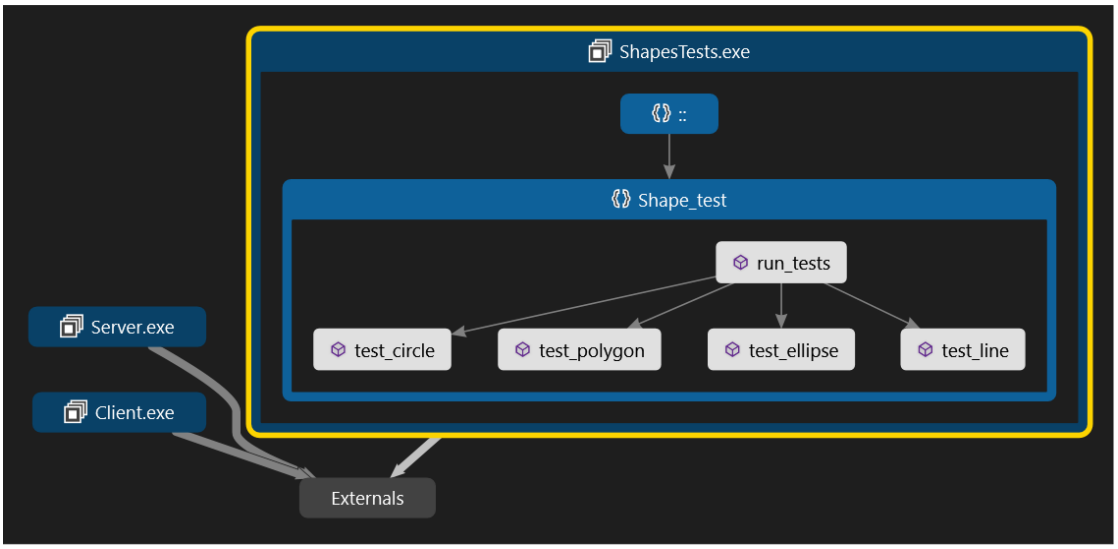

FIGURE 2.5 – Diagramme des dépendances du projet Shapes_tests

3 Détails d'implémentation

Cette section explique en détails les parties importantes de chaque projet. Elle explique en particulier les détails d'implémentation comme les choix de structures de données ou encore les héritages.

3.1 Mathématiques

Pour pouvoir définir des vecteurs et points dans un espace 2D on a besoin de créer différentes fonctions mathématiques, contenues dans *Maths.h*.

Tout d'abord, on définit une structure *Vec2* qui nous permet de stocker un couple de *float*, pouvant contenir ainsi un point ou un vecteur. De la même manière on définie une structure *Color* comme un ensemble de trois *integer*.

```
struct Vec2
{
        Vec2(float x, float y) : x(x), y(y) {}
        Vec2() : x(0.f), y(0.f) {}
```

```
        float x; /*!< x coordinate */
        float y; /*!< y coordinate */

        friend std::ostream& operator << (std::ostream& out, const Vec2& v);
};
Vec2 operator- (const Vec2& a, const Vec2& b) { return (Vec2(a.x - b.x, a.y - b.y)); }
Vec2 operator+ (const Vec2& a, const Vec2& b) { return (Vec2(a.x + b.x, a.y + b.y)); }
Vec2 operator* (int a, const Vec2& b) { return (Vec2(a*b.x, a*b.y)); }
Vec2 operator* (const Vec2& a, int b) { return (Vec2(b*a.x, b*a.y)); }
Vec2 operator* (float a, const Vec2& b) { return (Vec2(a*b.x, a*b.y)); }
Vec2 operator* (const Vec2& a, float b) { return (Vec2(b*a.x, b*a.y)); }
bool operator== (const Vec2& a, const Vec2& b) { if (a.x == b.x && a.y == b.y)
                return true; return false; }
bool operator== (const Vec2& a, Vec2& b) { if (a.x == b.x && a.y == b.y)
                return true; return false; }
std::ostream& operator << (std::ostream& out,const Vec2& v){
                out << "(" << v.x << "␣,␣" << v.y << ")"; return out; }
```

Ainsi on peut définir le dot product et la norm d'un *Vec*2, mais pour pouvoir calculer des résultats trigonométriques en 2D nous avons besoin du sinus, cosinus ainsi que de la fonction racine carré. Ces fonctions sont implémentées par appel aux fonctions assembleur directement (uniquement sous Windows, sous Linux les fonctions définies dans <cmath> sont utilisées), ce qui permet d'avoir un résultat plus précis et plus rapide que les fonctions de base.

```
//Exemple du sinus
double inline __declspec (naked) __fastcall fast_sqrt(double n)
{
        _asm fld qword ptr[esp + 4]
                _asm fsqrt
        _asm ret 8
}


float dot(Vec2& a, Vec2& b)
{
        return(a.x * b.x + a.y * b.y);
}


float norm(Vec2& a)
{
        return (fast_sqrt((a.x * a.x) + (a.y * a.y)));
}
```

3.2 Les formes géométriques

Le coeur du projet est de pouvoir créer et afficher des images constituées de formes géométriques. La première chose à laquelle on pense c'est donc de créer une classe mère qui définie une forme, de laquelle chaque forme géométrique spécifique (cercle, polygon, line, ellipse et image) va hériter.

3.2.1 La classe Shape

La classe *Shape* est une classe virtuelle pure (abstraite) qui demande aux classe héritantes d'implémenter les fonctions de transformations géométriques, d'aire, de périmètre, de serialisation, d'affichage et deux attributs. Les deux attributs sont le type de la classe qui hérite, celà permet de tester en temps constant le type d'un enfant et de $static\_cast$ pour retrouver le bon type d'objet, et une couleur. Le type est définit par un énumération, un tableau de string est associé à l'énumération pour pouvoir associer une string (commande) à une énumération et pouvoir switch dessus. De la même manière une énumération est faite pour les fonctions. On utilise le fait qu'une énumération s'incrémente par 1 pour créer le vecteur de string dans le même ordre que l'énumération, permettant ainsi de dire que $vecteur[i] = (Enumeration)i$;, d'où la définition du début de l'énumération à 0.

```
class Shape
        {

    ....

enum Derivedtype { CIRCLE=0, POLYGON, LINE, ELLIPSE, IMAGE, END_ENUM };
enum Functions { ROTATION = 0, HOMOTHETY, TRANSLATE,
        AXIAL_SYMETRY, CENTRAL_SYMETRY, UNKNOWN };
static const std::vector<std::string> transforms;
static const std::vector<std::string> shapes;

     ....

Shape(Derivedtype type, Color color) : m_type(type), m_color(color){};
Derivedtype type() const { return(m_type); }
const Color color() const { return(m_color); }
virtual float area() = 0;
virtual float perimeter() = 0;
virtual void translate(const Vec2& v) = 0;
virtual void homothety(float ratio) = 0;
virtual void homothety(const Vec2& p, float ratio) = 0;
virtual void rotate(float angle) = 0;
virtual void rotate(const Vec2& p, double angle) = 0;
virtual void centralSym(const Vec2& p) = 0;
virtual void axialSym(const Vec2& p, const Vec2& v) = 0;
virtual void display(SDL_Renderer* renderer, float ratio) = 0;
virtual void serialize( std::string& serial ) = 0;
virtual BoundingBox bounding_box() = 0;

        ....

Derivedtype m_type;
Color m_color;
}
```

Une fois la classe mère définie on peut définir les classes filles grâce à l'héritage. Tout d'abord nous avons décidé de définir les formes comme suit :

— Circle : Un point d'origine et un rayon.

— Polygon : Un ensemble de points ordonnés. On pourrait éviter de devoir ordonner les points pour être plus robuste en calculant la coque convexe avant de calculer le périmètre.
— Line : Un point par lequelle elle passe, ainsi qu'un vecteur direction. En fait une ligne est un segment car elle n'est pas infinie en implémentation mais sa longueur est définie par le vecteur direction qui n'est donc pas normalisé.
— Ellipse : Un point d'origine et un rayon en abscisses et ordonnées. Ainsi on peux voir qu'il n'est pas possible d'implémenter la rotation de l'Ellipse car les rayons sont alignés sur les axes.
— Image : Un point d'origine et une liste de composants de type *Shape* en plus d'une annotation.

La plupart des fonctions ne sont pas intéressantes à détailler et calculent de manière directe les différentes transformations, la boîte englobante non orientée ainsi que l'aire et le périmètre. L'aire ainsi que l'affichage du polygone sont effectués par triangulation. Le périmètre de l'ellipse est calculé par l'approximation de Ramanujan [6].

3.2.2 La classe Image

En revanche la classe *Image* est intéressante à détailler. En effet, une Image contient des composants qui peuvent eux-même être des Images. De plus cette classe est thread safe avec l'utilisation d'un mutex. En plus de la fonction d'affichage héritée de *Shape*, une Image a une deuxième fonction d'affichage. Cette fonction particulière calcule un ratio entre la taille de l'Image et la taille de la fenêtre d'affichage, afin de pouvoir afficher l'Image dans sa globalité dans la fenêtre. Quand un composant est appellé avec un ratio différent de 1, une homothétie avec comme centre l'origine (0,0) est opérée sur le composant avant affichage. De plus une Image peut se créer à partir d'une *string* qu'elle desérialise et qui montre un exemple de l'utilisation de l'énumération des types dérivés de la classe *Shape* (c.f. Listing suivant).

```
class Image : public Shape
        {

    ....

  Image(Vec2 o = { 0, 0 }) : Shape(Shape::IMAGE, Color(0, 0, 0)), annotation(std::string()),
  components_(std::vector<Shape *>()), origin_(o){}
  ~Image()
  {
        components_.clear();
  }
  //Due to the use of mutex which is not copyable, Image is not copyable either
  Image(const Image&) = delete;
  Image& operator=(Image const&) = delete;

  ....

  //Exemple fonction de rotation
```

```
  void rotate(float angle)
  {
        //Lock mutex to be thread safe
        std::lock_guard<std::mutex> guard(mutex);
    //Apply needed transform to all components
        for (auto component : components_)
        {
                component->rotate(angle);
        }
  }

  ...

//Base display function from Shape
  void display(SDL_Renderer* renderer, float ratio)
  {
        std::lock_guard<std::mutex> guard(mutex);
        for (auto component : components_)
        {
                component->display(renderer, ratio);
        }
  }
//Special display function
  void display(SDL_Renderer* renderer)
  {
        BoundingBox bb = bounding_box();
        std::lock_guard<std::mutex> guard(mutex);
        //Get window size
        int w, h;
        SDL_GetRendererOutputSize(renderer, &w, &h);
        Vec2 center((w / 2), (h / 2));
        bb.x_max = bb.x_max + center.x;
        bb.x_min = bb.x_min + center.x;
        bb.y_max = bb.y_max + center.y;
        bb.y_min = bb.y_min + center.y;
        Vec2 v1 = (center - Vec2( bb.x_max, bb.y_max ));
        Vec2 v2 = (center - Vec2(bb.x_min, bb.y_min));
        int im_w, im_h;
        float n1 = norm(v1);
        float n2 = norm(v2);
        //Compute ratio as biggest diagonal
        if (norm(v1) > norm(v2))
        {
                im_w = n1 * 2;
                im_h = im_w;
        }
        else
        {
                im_w = n2 * 2;
                im_h = im_w;
        }
        float w_ratio = (float) w / im_w;
        float h_ratio = (float) h / im_h;
        float final_ratio = 1.f;
```

```
        if (w_ratio < 1.f || h_ratio < 1.f)
         {
                if (w_ratio <= h_ratio)
                 {
                        final_ratio = w_ratio;
                }
                 else
                {
                        final_ratio = h_ratio;
                }
        }
        //Display all component with desired ratio
        for (auto component : components_)
        {
                component->display(renderer, final_ratio);
        }
}

....

void deserialize(std::string s)
{
        components_.clear();
        std::istringstream buf(s);
        for (std::string word; buf >> word;)
        {
                switch (Shape::ShapeStringToEnum(word))
                {
                        case Shape::CIRCLE:
                        {
                                ...Read all infos from buffer...
                                add_component(new Circle(Vec2(x, y), rad, Color(r, g, b)));
                }break;

            case Shape::POLYGON:
            {
                ...Read all infos from buffer...
                add_component(new Polygon(points, Color(r, g, b)));
            }break;

            case Shape::LINE:
            {
                ...Read all infos from buffer...
                add_component(new Line(Vec2(x, y), Vec2(dir_x, dir_y), Color(r, g, b)));
            }break;

            case Shape::ELLIPSE:
            {
              ...Read all infos from buffer...
              add_component(new Ellipse(Vec2(x, y), Vec2(rad_x, rad_y), Color(r, g, b)));
            }break;

            case Shape::UNKNOWN:
            {
```

```
                //Assume only annotation cast the unknown shape enum
                annotate(annotation);
            }break;
}
}
}

        ....
}
```

La classe *Image* a été rendu thread safe, car c'est en fait le seul composant à devoir, l'être au sein du serveur. Ainsi, dès que Image a été thread safe, il n'a pas été besoin de modifier le serveur de quelque façon que ce soit.

3.3 Le Client

Le Client va donc utiliser $boost :: asio$ pour créer une connection avec le serveur. Quand un client se lance, une image vide est créée, ainsi que les variables nécèssaires pour boost : :asio. De plus, les variables servant à l'affichage sont créees, comme un pointeur sur SDL_Window, etc.

3.3.1 La classe Client

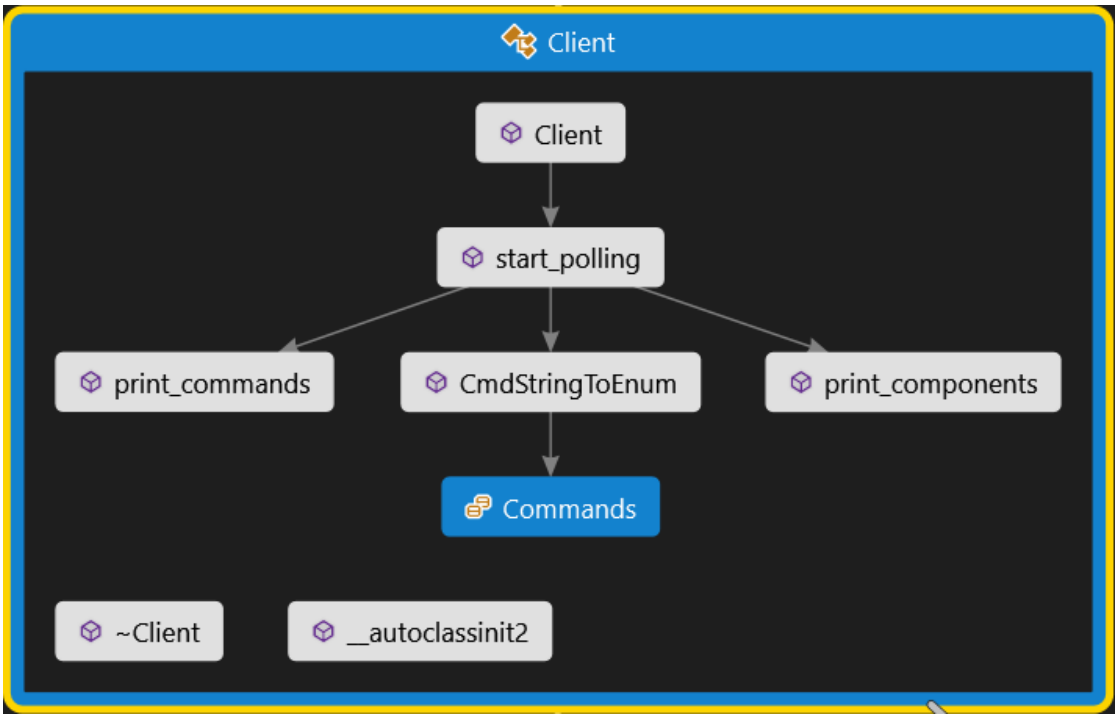

Figure 3.1 – Diagramme de fonctions de la classe Client

La classe *Client* va initialiser les différentes variables et créer un objet *ClientIO* ainsi qu'un thread dans lequel le *io_service* de *boost* :: *asio* va en permanence poll la queue des résultats (c.f. Section 2.1). Ensuite, il démarre le polling des commandes. De la même manière que pour la classe *Shape* une enumération et un vecteur de *string* associées est créer pour définir les commandes valides. Ensuite l'entrée utilisateur est lue en permanence et si une commande est bonne elle est executée. Une gestion d'exception est mise en place au cas où l'entrée utilisateur ne corresponde pas au type attendu.

Les commandes disponibles sont :

- "display" pour afficher l'image tant que l'utilisateur ne ferme pas la fenêtre
- "help" pour afficher les commandes disponibles
- "print" pour afficher l'ensemble des composants de l'image
- "delete" pour supprimer un composant
- "make" pour créer un composant (Exemple dans le listing suivant)
- "transform" pour effectuer une transformation sur un composant
- "send" pour envoyer l'image au serveur

```
class Client
{
  Client(std::string ip, std::string port, boost::asio::io_service& service) :
    io_service(service)
  {
    img = new Image();
    //Initiliaze connection
    resolver = new tcp::resolver(io_service);
    auto endpoint_iterator = resolver->resolve({ "127.0.0.1", "8080" });
    c = new ClientIO(io_service, endpoint_iterator, *img);
    std::thread* t = new std::thread([&](){ io_service.run(); });
    SDL_Init(SDL_INIT_VIDEO);
    start_polling();
  };


  void start_polling()
  {
    const unsigned int LINE_MAX_SIZE = 256;
    // Start polling for commands
    char line[LINE_MAX_SIZE];
    std::string cmd;
    //    While the users is entering commands we react to it
    std::cout << "Available␣commands␣:␣";
    print_commands();
    std::cout << std::endl << "Command␣:␣";
    while (std::cin.getline(line, LINE_MAX_SIZE))
    {
       cmd = std::string(line);
       // Convert string to a command enum we can switch on
      switch (CmdStringToEnum(cmd))
      {
        case Commands::DISPLAY:
        {
```

```
        //Display img
    }break;

    case Commands::HELP:
    {
        //Print available commands
    }break;

    ....

case Commands::MAKE:
{
    //Print available shapes
    std::cout << "Available shapes : ";
    Shape::print_shapes();
    std::cout << std::endl;
    std::cout << "Enter Shape Type : ";
    std::cin >> cmd;
    //Switch on user entered shapes
    switch (Shape::ShapeStringToEnum(cmd))
    {
        case Shape::Derivedtype::CIRCLE:
        {
          try
          {
             float x, y, radius;
            int r, g, b;
            std::cout << "Origin x : ";
            std::cin >> x;
            std::cout << "Origin y : ";
            std::cin >> y;
            std::cout << "Radius : ";
            std::cin >> radius;
            std::cout << "Color R : ";
            std::cin >> r;
            std::cout << "Color G : ";
            std::cin >> g;
            std::cout << "Color B : ";
            std::cin >> b;
            if (std::cin.fail())
            {
            std::cin.clear();
            throw std::domain_error("Bad input");
            }
          img->add_component(new Circle(Vec2(x, y), radius, Color(r, g, b)));
          std::cout << "Circle created" << std::endl;
          }
          catch (std::exception& e)
          {
          //Catch error of types when cin trying to convert to desired type
          std::cout << std::endl << " Problem : " << e.what() << std::endl;
          }
                    }break;
  ....
```

```
            }
          ....
           }break;
        .....
       }
     }
   }

   ....
}
```

3.3.2 La classe ClientIO

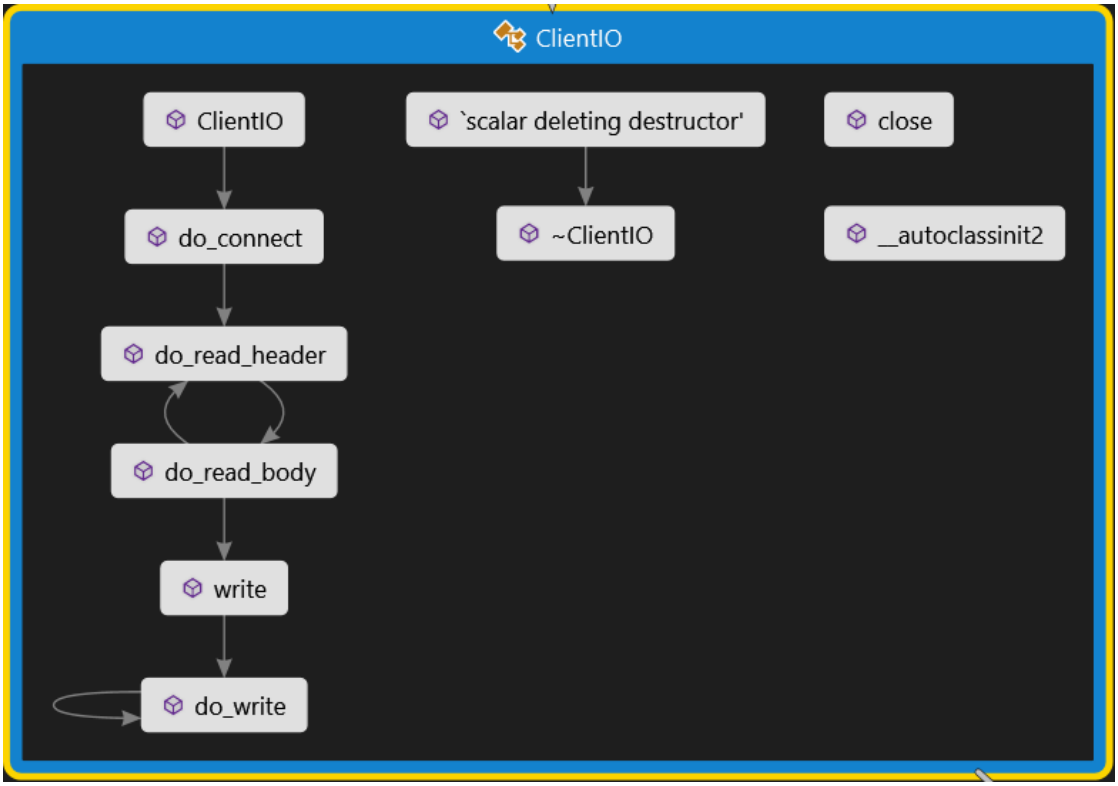

FIGURE 3.2 – Diagramme de fonctions de la classe ClientIO

La classe ClientIO est celle qui s'occupe de demander les opérations asynchrone. Les fonctions de handler étant des lambdas elles n'apparaissent pas dans le diagramme. Les opérations disponibles sont d'écrire sur le socket, de lire sur le socket (si le message reçu est "GET" alors on envoit l'image sinon on deserialise l'image reçue) et la fonction de connection bien entendu. Un exemple de la fonction de lecture est donné dans le listing suivant.

```
void do_read_body()
{
```

```
//Ask OS to read from socket
  boost::asio::async_read(socket_,
      boost::asio::buffer(read_msg_.body(), read_msg_.body_length()),
      //Lambda handler function called on operation completion
      [this](boost::system::error_code ec, std::size_t /*length*/)
      {
        if (!ec)
        {
                        if (std::string(read_msg_.body()) == "GET")
                        {
                                //Send image
                                Message msg;
                                std::string s;
                                img.serialize(s);
                                msg.body_length(s.length());
                                std::memcpy(msg.body(), s.c_str(), msg.body_length());
                                msg.encode_header();
                                write(msg);
                        }
                        else
                        {
                                //Get image
                                img.deserialize(std::string(read_msg_.body()));
                        }
          do_read_header();
        }
        else
        {
          socket_.close();
        }
      });
}
```

3.4 Le Server

Le Server va donc utiliser *boost* :: *asio* pour gérer les connection avec les clients. Quand un client se connecte, il rejoint une *Room* qui contient l'ensemble des participants à jour, un objet *Client* est créer qui contient une image ainsi qu'un identifiant unique. La classe *Server* s'occupe d'initialiser les différentes variables (*boost* :: *asioetSDL*) ainsi que de poll les commandes et d'effectuer les opérations.

3.4.1 La classe Server

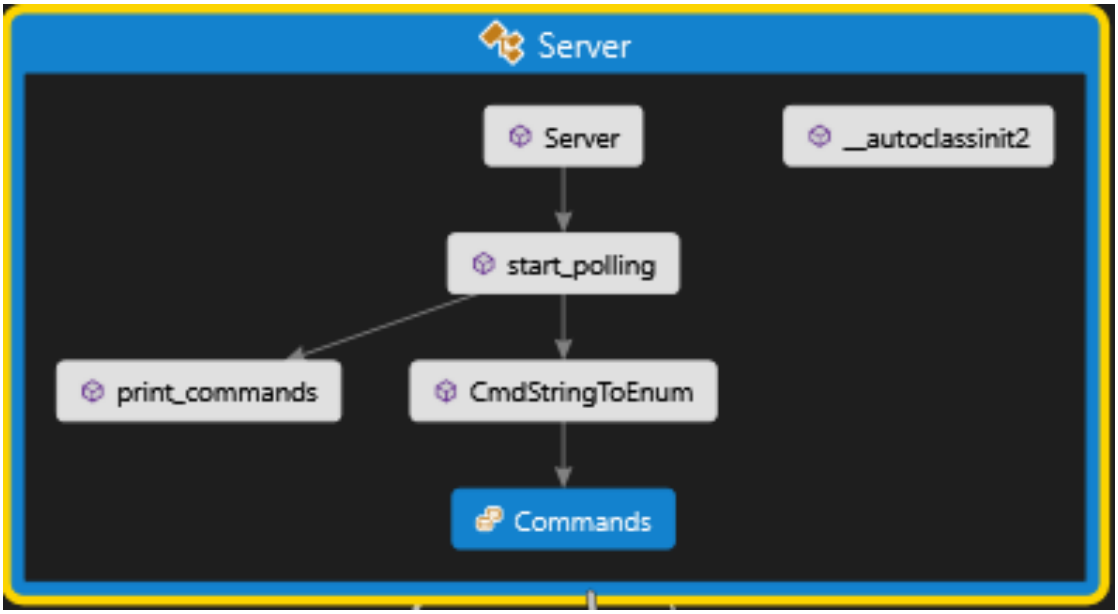

FIGURE 3.3 – Diagramme de fonctions de la classe Server

Cette classe est la principale du projet qui crée les différents objets nécessaires comme la SDL_window, les variables nécèssaires à $boost :: asio$ et ensuite poll les commandes utilisateurs. De la même manière que pour la classe $Shape$ une enumération et un vecteur de $string$ associées est créer pour définir les commandes valides. Ensuite l'entrée est lue en permanence et si une commande est bonne elle est executée. Une gestion d'exception est mise en place au cas où l'entrée utilisateur ne correspondent pas au type attendu.

Les commandes disponibles sont :

- "display" pour afficher l'image associée à un client tant que l'utilisateur ne ferme pas la fenêtre, attention l'image affichée est locale !
- "help" pour afficher les commandes disponibles
- "get" pour demander à tous les clients connectés d'envoyer leurs Images
- "send" pour envoyer à tous les clients l'image locale qui leurs est associée (souvent avec une nouvelle annotation)
- "annotate" pour annoter une image associée à un client
- "print" pour afficher l'identifiant unique des cleints connectés
- "stats" pour afficher des statistiques sur le nombre de formes géométriques et les couleurs
- "patchwork" pour créer et afficher le patchwork résultat

La même stratégie étant appliquée au projet client, pour un exemple voir le Listing dans la section 3.3.1.

3.4.2 LA CLASSE SERVERIO

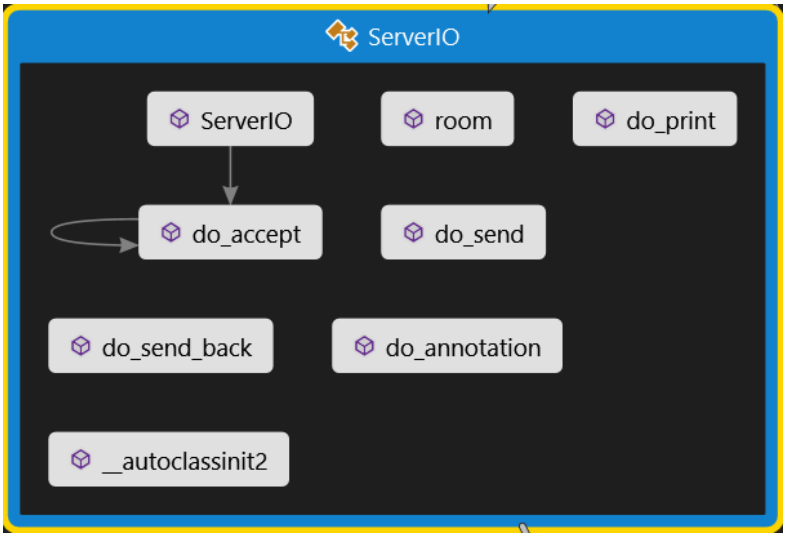

FIGURE 3.4 – Diagramme de fonctions de la classe ServerIO

Cette clase rassemble les fonctions d'éxecution d'opérations appellées par la classe Server. Par exemple quand la commande de demande de réception des images est entrée, la fonction $do_send()$ est en fait appellée. Cette fonction va ensuite demander au client d'envoyer et recevoir les messages.

3.4.3 La classe Client

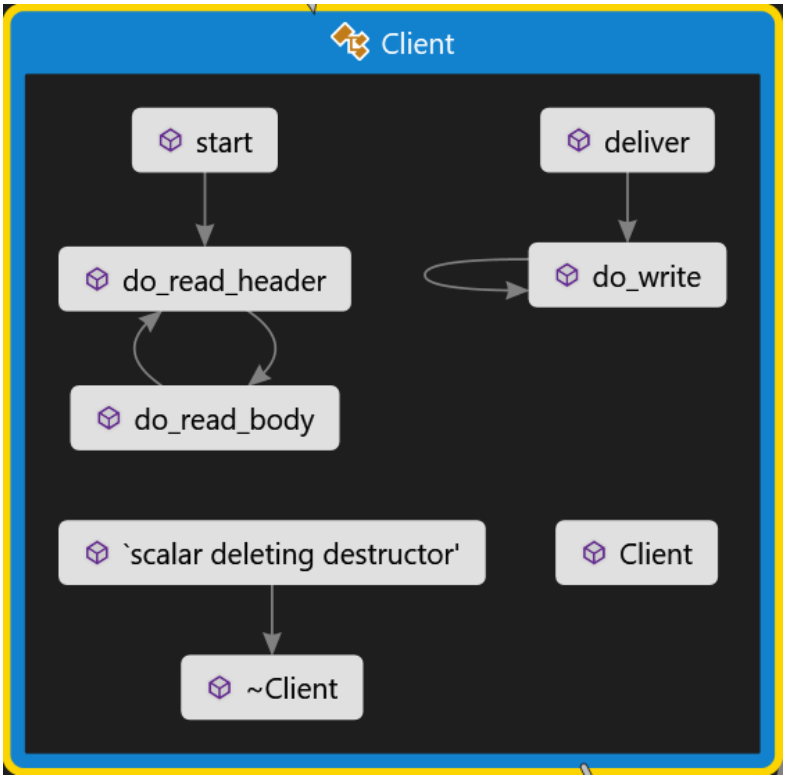

FIGURE 3.5 – Diagramme de fonctions de la classe Client

C'est cette classe qui implémente la partie $boost :: asio$ de manière similaire à la classe $ClientIO$ du projet Client. Pour plus de détails, voir la section 3.3.2.

3.4.4 Les tests

Du à la nature des opérations asynchrones, aucun test autre qu'empirique n'a été effectué sur le serveur et le client. En revanche le bon fonctionnement des formes géométriques, notamment des transformations ont été testé. Ceci se retrouve dans le projet $Shape\_test$. Ce projet lance les différents tests dont un exemple est donné dans le listing suivant. De plus une méthode $test_assert$ est définie dans le fichier $Asserts.h$. Cette méthode prend en entrée une condition qu'elle va tester et un message. Le message suivit de "OK" sera affiché si la condition est vrai, sinon le message sera suivit de "KO".

Du fait que tout les résultats sont en $float$, les tests sont compliqués puisque les tests d'égalité entre float sont trop précis, ce qui peut amener a des tests qui ne passe pas alors qu'ils sont bon. Par exemple, la valeur peut être très proche de zéro, et on veut tester si elle est égale à zéro, ce qui nous retourne $false$.

3.5 Les contraintes du projet

On rappel que dans la section 1.1, le cahier des charges du projet a été défini, avec un certain nombre de contraintes à respecter. La plupart des contraintes ont été respectées hormis deux qui n'ont pas été implémentées à cause d'une incompréhension.

3.5.1 La contrainte d'ordering

Une des contraintes est de pouvoir ordonner les formes selon plusieurs critères, à l'aide de l'opérateur <=. Cependant, il n'est pas possible de surcharger un même opérateur plusieurs fois. De plus faire un opérateur <= unique n'est même pas possible, en effet, deux formes peuvent avoir le même périmètre alors qu'elles ont une aire différentes. Il est difficile de créer un opérateur qui a les propriétés du $strictweakordering$.

Cependant, bien que non implémenté, il est très simple d'ordonner les formes par aire par exemple, puisqu'elle ont toute une fonction area() qui calcul l'air, on a donc : $shapeA <= shapeB <=> shapeA.area() <= shapeB.area()$

3.5.2 La contrainte de non-duplicata

Cette contrainte demande à ce qu'aucun doublon de forme géométrique n'existe. Ce qui peut être interpreté de plusieurs façons.

La première est de dire que l'on ne peut avoir qu'une seule instance de chaque type de forme dans une image. Si c'est cette définition qui est retenue, alors ont peut l'implémenter en utilisant le pattern singleton.

La seconde consiste à dire qu'aucune forme ne peut être clonée ou égale à une autre. Ceci peut être implémenter en utilisant la fonction $delete$ sur le constructeur par copie et assignement (comme effectué dans la classe Image) ou encore en vérifiant à la création d'une figure qu'elle n'est pas égale à une autre. Ce qui pourrait s'implémenter comme suit :

```
bool operator==(const Shape& a, const Shape& b)
{
    if( a.type() != b.type() )
        return false;

    switch ( a.type() )
    {
        case Derivedtype::CIRCLE :
        {
            Circle* c_a = static_cast<Circle>(a);
            Circle* c_b = static_cast<Circle>(b);
            if ( c_a == c_b )
                return true;
```

```
            return false;
        }break;

        ... Implements for others type ...
    }
}
```

4 Améliorations

L'impémentation peut étre améliorée en appliquant les principes de l'inversion de Controle. Un Factory peut etre défini afin de déleguer la reponsabilité de création des differentes formes à une classe tierce et utiliser juste une interface pour appeller le factory. Ainsi, on aura une meilleurs modularité ainsi qu'un changement agile de l'implémentation des formes sans affecter l'implémentation d'origine.

Le design pattern Composite peut aussi être utilisé au niveau de l'inclusion d'une image par une autre du fait que la structure est la même.

Par ailleurs, la testabilité peut aussi etre améliorée par le principe d'injection de l'indépendance en utlisant une abstraction de la forme qu'on passe en paramétre au niveau du constructeur (afin d'injecter des mocks avec differentes variations possibles), améliorant ainsi les tests unitaires.

5 Conclusion

En conclusion, le projet a été mené à bien en assez peu de temps compte tenu de l'ensemble des projets demandés sur l'année. La solution proposée réponds à la plupart des définitions du cahier des charges et nous a permis d'étudier d'un peu plus prêt la bibliothèque *boost*, ainsi que d'améliorer notre vision et programmation en c++11.

Comme vu dans les améliorations, le projet est loin d'être irréprochable ce qui laisse la place encore à de nombreuses fun heures de programmations pour le finir.

6 Bibliographie / Sitographie

[1] Patchwork subject, https ://elearning.u-pem.fr/pluginfile.php/45760/mod_resource/content/2/Projet%20Patchwork.pdf
[2] Boost ASIO, http ://www.boost.org/doc/libs/1_60_0/doc/html/boost_asio.html
[3] SDL, https ://www.libsdl.org/
[4] Vidéo de présentation, https ://youtu.be/vJRDc06tM80
[5] Asynchronous network, http ://www.boost.org/doc/libs/1_60_0/doc/html/boost_asio/overview/

core/basics.html
[6] Ramanujan approx, http ://www.johndcook.com/blog/2013/05/05/ramanujan-circumference-ellipse/
[7] Example boost, http ://www.boost.org/doc/libs/1_55_0/doc/html/boost_asio/examples/cpp11_examples.html