Tangram

Rapport de projet

Réalisé par :

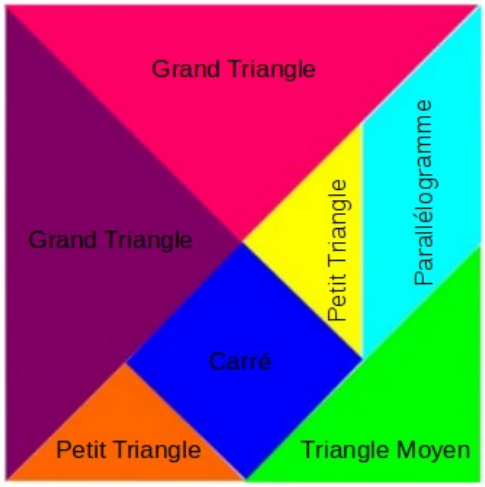
BEN HAMOUDA Amel
DURAND Aurélien

Université Paris-Est Marne-la-Vallée
Master 2 Science de l'image
2019/2020

Table des matières

I. Description du projet

Notre projet Tangram, consiste à développer un jeu d'origine chinoise composé de 7 pièces.



Ce casse-tête, a pour but de reproduire une forme donnée. Les règles sont simples :

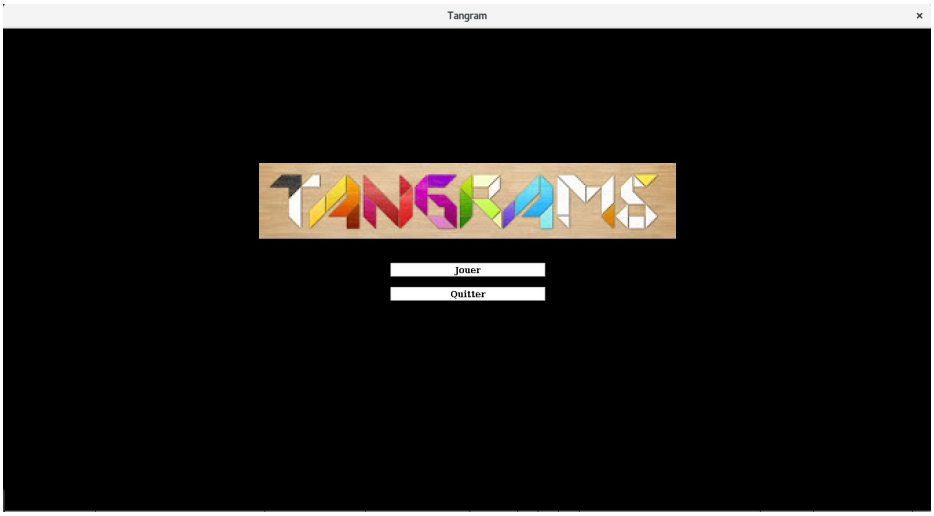
→ on utilise toujours la totalité des pièces qui doivent être posées à plat,
→ et ne pas superposer les pièces.

Notre projet est principalement découpé en deux parties.
La première est la partie "calcul" dans laquelle se trouve toute la partie de création et de mouvement des formes.
La seconde est la partie "Interface" dans laquelle se trouve toute la partie nécessaire à l'affichage du jeu (autre que les figures) et les fonctionnalités telles que les différents boutons.

Ce rapport a pour but d'expliquer à l'utilisateur le mode de fonctionnement du jeu et de décrire l'architecture que nous allons développer pour notre projet.

II. Mode d’emploi

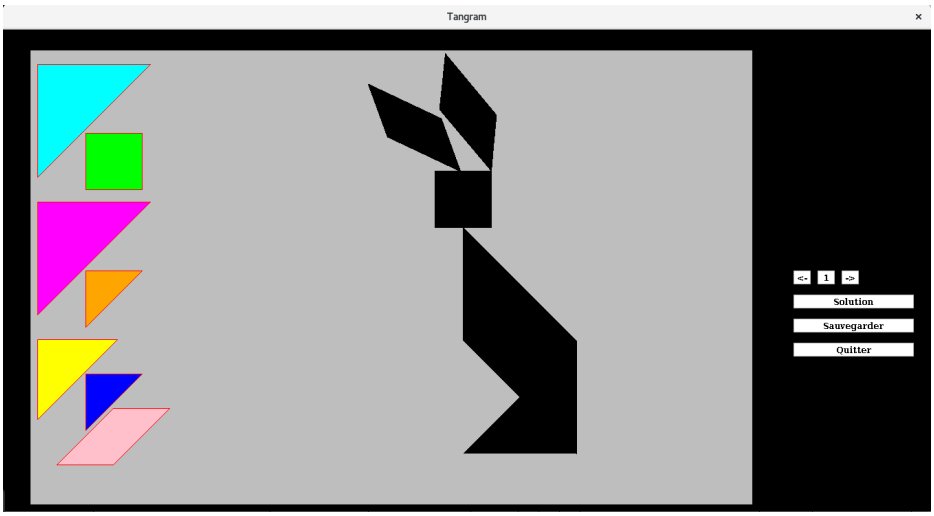
Des qu’on lance le programme on obtient :



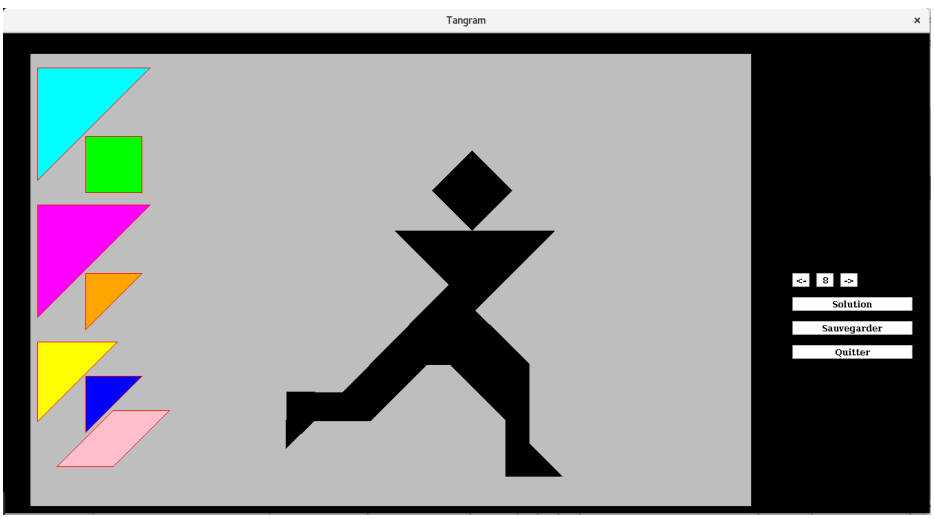
On a alors deux choix :

→ soit "Quitter" qui ferme le jeu,

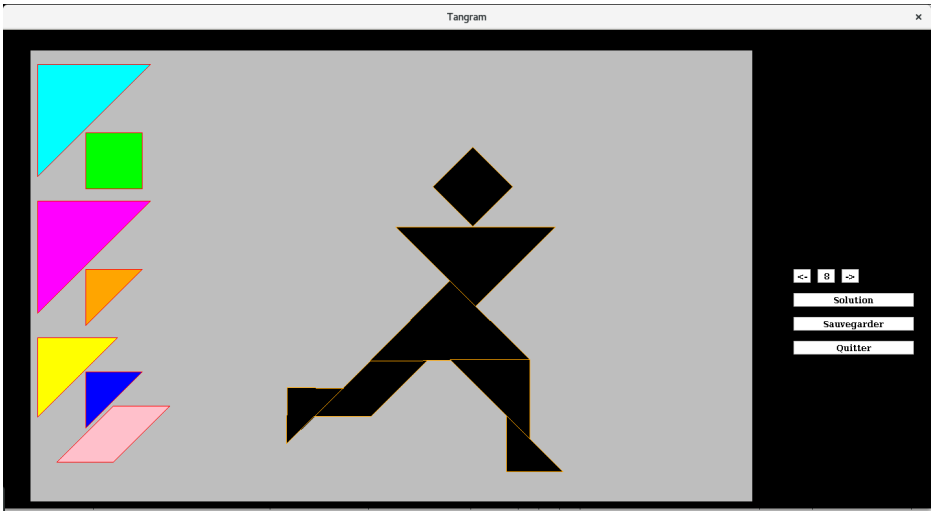
→ soit "Jouer" et on arrive sur cette fenêtre :



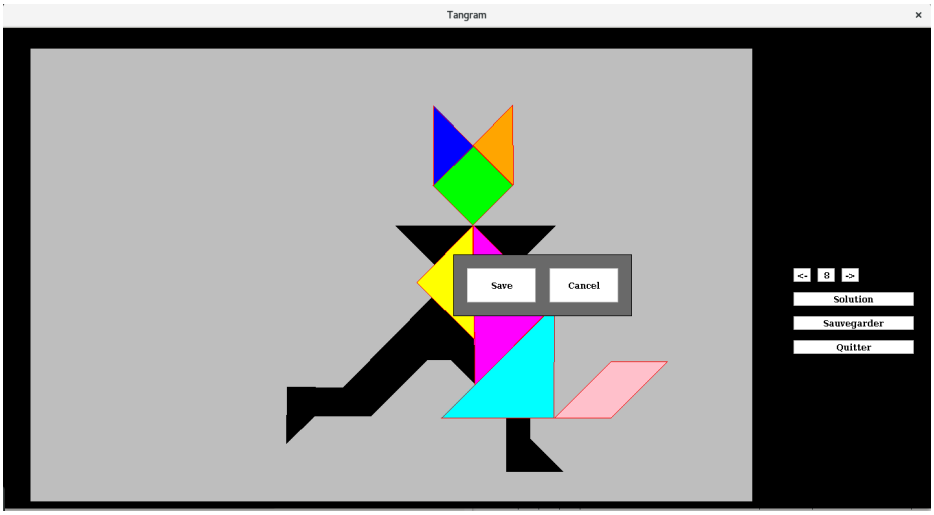
Et là on commence le jeu, on peut changer de figure avec les flèches (droite / gauche) :



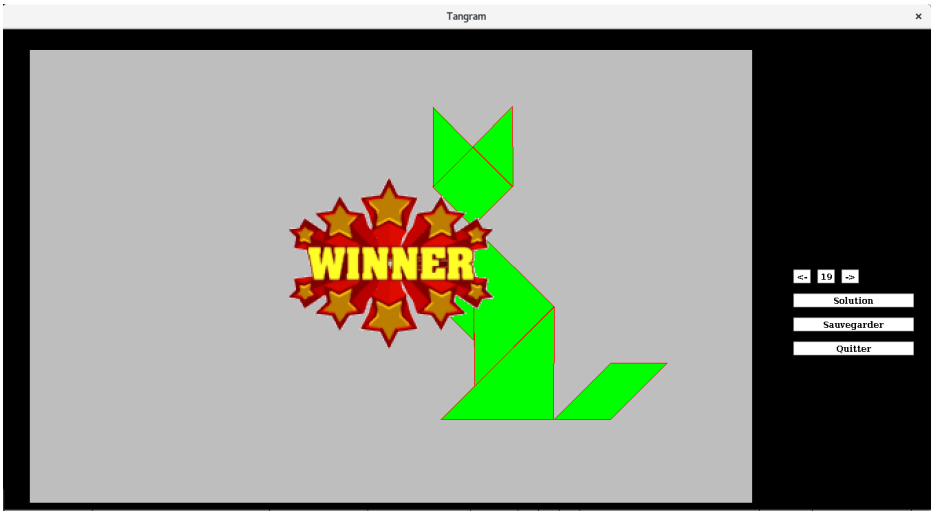
On peut afficher la solution avec le bouton "Solution" :



On peut créer une nouvelle figure, la dessiner au-dessus du jeu et en appuyant sur le bouton "Sauvegarder" on obtient le choix suivant:



Et du coup on retrouvera cette figure à la fin du fichier "savefig.txt" qui contient l'ensemble des figures du jeu.
Quand on réussi à résoudre le jeu il y a un message qui apparaît :



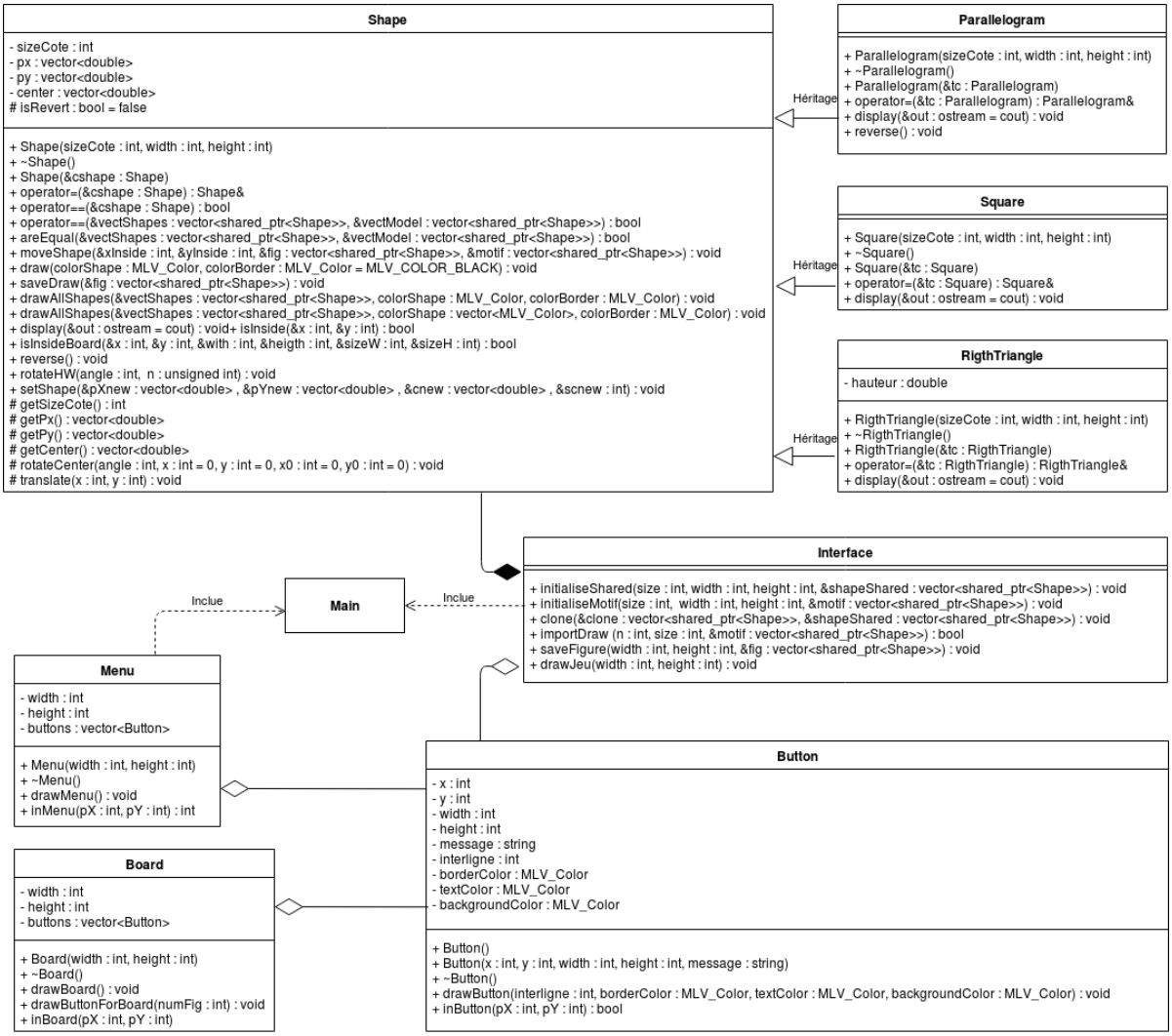
Et comme dans l’interface de départ on a un bouton "Quitter" qui ferme le jeu.

Les commandes du jeu sont les suivantes :

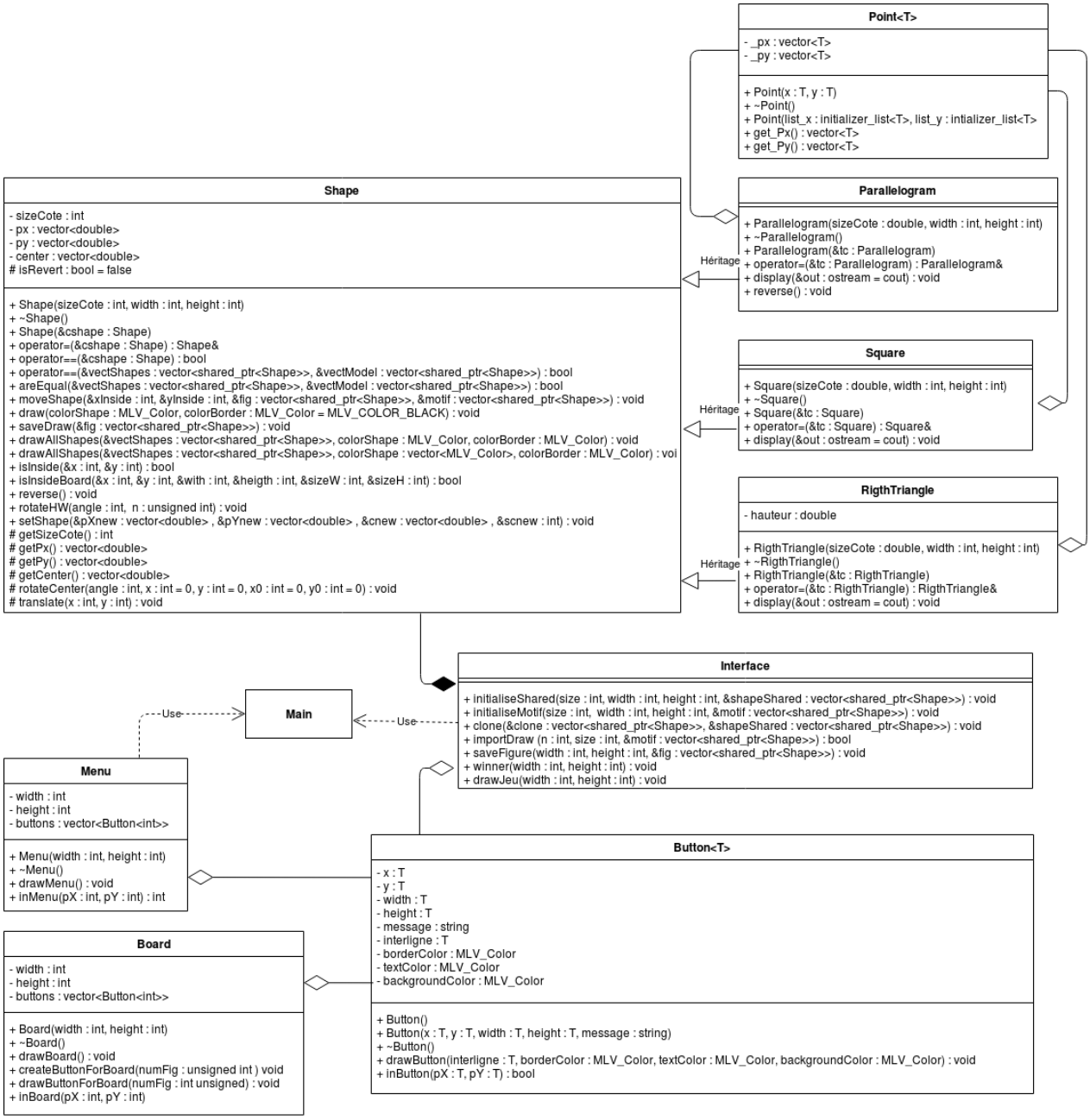
→ clic gauche pour les événement liés aux boutons.
→ clic gauche sur une figure pour la sélectionner.
→ clic gauche en dehors de la figure pour la désélectionner.
→ quand la figure est sélectionnée :
* clic droit pour la rotation.
* clic gauche + droit pour la rotation lente.
* clic molette symétrie.
→ ctrl+s pour sauvegarder une figure (en plus du bouton).

III. Diagramme des classes

1. Version 1



2. Version 2


Point<T>
- _px : vector<T>
- _py : vector<T>
+ Point(x : T, y : T)
+ ~Point()
+ Point(list_x : initializer_list<T>, list_y : intializer_list<T>
+ get_Px() : vector<T>
+ get_Py() : vector<T>
Shape
- sizeCote : int
- px : vector<double>
- py : vector<double>
- center : vector<double>
isRevert : bool = false
+ Shape(sizeCote : int, width : int, height : int)
+ ~Shape()
+ Shape(&cshape : Shape)
+ operator=(&cshape : Shape) : Shape&
+ operator==(&cshape : Shape) : bool
+ operator==(&vectShapes : vector<shared_ptr<Shape>>, &vectModel : vector<shared_ptr<Shape>>) : bool
+ areEqual(&vectShapes : vector<shared_ptr<Shape>>, &vectModel : vector<shared_ptr<Shape>>) : bool
+ moveShape(&xInside : int, &yInside : int, &fig : vector<shared_ptr<Shape>>, &motif : vector<shared_ptr<Shape>>) : void
+ draw(colorShape : MLV_Color, colorBorder : MLV_Color = MLV_COLOR_BLACK) : void
+ saveDraw(&fig : vector<shared_ptr<Shape>>) : void
+ drawAllShapes(&vectShapes : vector<shared_ptr<Shape>>, colorShape : MLV_Color, colorBorder : MLV_Color) : void
+ drawAllShapes(&vectShapes : vector<shared_ptr<Shape>>, colorShape : vector<MLV_Color>, colorBorder : MLV_Color) : voi
+ isInside(&x : int, &y : int) : bool
+ isInsideBoard(&x : int, &y : int, &with : int, &heigth : int, &sizeW : int, &sizeH : int) : bool
+ reverse() : void
+ rotateHW(angle : int, n : unsigned int) : void
+ setShape(&pXnew : vector<double> , &pYnew : vector<double> , &cnew : vector<double> , &scnew : int) : void
# getSizeCote() : int
# getPx() : vector<double>
# getPy() : vector<double>
# getCenter() : vector<double>
# rotateCenter(angle : int, x : int = 0, y : int = 0, x0 : int = 0, y0 : int = 0) : void
# translate(x : int, y : int) : void
Parallelogram
+ Parallelogram(sizeCote : double, width : int, height : int)
+ ~Parallelogram()
+ Parallelogram(&tc : Parallelogram)
+ operator=(&tc : Parallelogram) : Parallelogram&
+ display(&out : ostream = cout) : void
+ reverse() : void
Héritage
Square
+ Square(sizeCote : double, width : int, height : int)
+ ~Square()
+ Square(&tc : Square)
+ operator=(&tc : Square) : Square&
+ display(&out : ostream = cout) : void
Héritage
RigthTriangle
- hauteur : double
+ RigthTriangle(sizeCote : double, width : int, height : int)
+ ~RigthTriangle()
+ RigthTriangle(&tc : RigthTriangle)
+ operator=(&tc : RigthTriangle) : RigthTriangle&
+ display(&out : ostream = cout) : void
Héritage
Interface
+ initialiseShared(size : int, width : int, height : int, &shapeShared : vector<shared_ptr<Shape>>) : void
+ initialiseMotif(size : int, width : int, height : int, &motif : vector<shared_ptr<Shape>>) : void
+ clone(&clone : vector<shared_ptr<Shape>>, &shapeShared : vector<shared_ptr<Shape>>) : void
+ importDraw (n : int, size : int, &motif : vector<shared_ptr<Shape>>) : bool
+ saveFigure(width : int, height : int, &fig : vector<shared_ptr<Shape>>) : void
+ winner(width : int, height : int) : void
+ drawJeu(width : int, height : int) : void
Use
Main
Use
Menu
- width : int
- height : int
- buttons : vector<Button<int>>
+ Menu(width : int, height : int)
+ ~Menu()
+ drawMenu() : void
+ inMenu(pX : int, pY : int) : int
Button<T>
- x : T
- y : T
- width : T
- height : T
- message : string
- interligne : T
- borderColor : MLV_Color
- textColor : MLV_Color
- backgroundColor : MLV_Color
+ Button()
+ Button(x : T, y : T, width : T, height : T, message : string)
+ ~Button()
+ drawButton(interligne : T, borderColor : MLV_Color, textColor : MLV_Color, backgroundColor : MLV_Color) : void
+ inButton(pX : T, pY : T) : bool
Board
- width : int
- height : int
- buttons : vector<Button<int>>
+ Board(width : int, height : int)
+ ~Board()
+ drawBoard() : void
+ createButtonForBoard(numFig : unsigned int) void
+ drawButtonForBoard(numFig : int unsigned) : void
+ inBoard(pX : int, pY : int)

IV. Description de l’architecture

Au niveau de la partie "calcul", on a choisi de distinguer quatre classes différentes.

La classe mère qui est *Shape* qui va comporter la majorité des méthodes permettant de travailler avec les pièces du Tangram. Ensuite on prend trois classes filles héritant de *Shape* et ayant des constructeurs qui leur sont propres. On a alors : *Parallelogram*, *Square* et *RightTriangle*.
Ces trois classes filles permettent donc de créer chacune des pièces du Tangram séparément et héritent des méthodes de *Shape*. Elles redéfinissent les méthodes virtuelles de la classe *Shape* si cela est nécessaire et utilisent la classe *Point* pour manipuler les coordonnées des figures.

Alors qu’au niveau "Interface", on a choisi de distinguer trois classes différentes.
Les classes *Menu* et *Board* qui contiennent des boutons (classe *Button*), ainsi que le plateau de jeu dans la classe *Board*.

Ces différentes classes sont rassemblées dans la classe *Interface,* qui permet la "fusion" des deux parties différentes.

Nous allons expliquer plus en détail les différentes classes dans la suite du rapport.

1. Partie calcul

A – Classe Point

C’est une classe template qui permet la gestion des coordonnées de points.

Ces attributs sont :

→ un vecteur de point X qui permet le stockage des points X,
→ un vecteur de point Y qui permet le stockage des points Y.

Ces méthodes sont :

→ un constructeur à deux paramètres qui prend deux objets T,
→ un destructeur virtuel,
→ un constructeur à deux paramètres (les attributs) qui permet d’avoir une liste d’initialisateur afin de créer les vecteur x et y,
→ une méthode "get_Px" qui permet de renvoyer la liste des points X,
→ une méthode "get_Py" qui permet de renvoyer la liste des points Y.

B – Classe Shape

C’est la classe mère qui comprend l’ensemble des méthodes et attributs nécessaires à la fabrication et au déplacement des pièces du Tangram.

Ces attributs sont :

→ deux vecteurs de coordonnées (x,y) délimitant un polygone,
→ la taille du côté de construction, par exemple le côté d’un carré,
→ le centre de masse du polygone calculé à partir des coordonnées,
→ et un booléen pour savoir si on a effectué la symétrie de l’objet.

Détails des méthodes et de leur implémentation :

→ constructeur : celui-ci à 3 paramètres recueillant la position sur la fenêtre du premier point de construction du polygone ainsi que la taille d’un côté de construction. Il est déclaré en implicite car les paramètres doivent être en double pour la taille d’un coté et la position dans la fenêtre est une position entière. Il constitue un constructeur générique de forme et est appeler dans les classes filles pour initialiser les attributs de la classe *Shape*.

→ destructeur : il est déclaré en virtuel car la classe *Shape* doit être héritée. En déclarant le destructeur en virtuel cela permet de ne pas avoir d’ambiguïté à la destruction des objets filles de la classe *Shape* et d’assurer la hiérarchie de destruction. L’effet du destructeur étant implicite et la classe *Shape* ne disposant pas de pointeur, donc pas de mémoire allouée, il n’y a pas besoin de réaliser sont implémentation.

→ constructeur de copie : celui-ci est normalement implicite car il n’y a pas de données utilisant des pointeurs. On l’explicite tout de même en venant recopier chaque attribut d’un objet *Shape* mis en paramètre vers l’instance courante.

→ surcharge de l’opérateur de copie : ici on code le constructeur de copie de la classe *Shape* pour pouvoir faire une utilisation de *std::swap* dans le projet. En effet comme pour le constructeur de copie, la classe *Shape* ne disposant pas de valeur pointée la copie implicite est normalement suffisante. On utilise une instance "copy", utilisant le constructeur de copie de *Shape*, de l’objet mis en paramètre et avec *std::swap* on assure le déplacement des données de l’objet en paramètre vers l’instance courante. Si il y avait eu des pointeurs dans la classe *Shape*, l’utilisation de *std::swap* et de l’objet temporaire "copy" aurait ainsi permis d'éviter les potentielles fuites mémoire.

→ surcharge de l’opérateur de comparaison : afin de pouvoir comparer deux objets de type *Shape* dans le jeu on se doit de surcharger l’opérateur de comparaison. En effet dans le cadre du Tangram on accepte une certaine tolérance sur la condition de victoire qui implique que deux objets soient

considérés comme équivalents à plus ou moins un seuil choisi. La condition de seuil que nous avons choisie est telle que deux objets soient considérés comme égaux si leur vecteurs de positions sont de même taille et que l'on trouve pour ces deux figures N points égaux à ± 20%. Donc pour un carré par exemple on regarde les quatre sommets de chaque objet et si les positions en x et en y concordent à ± 20% on renvoie alors la valeur booléenne "true" en sortie du test.

→ "areEqual" : c'est une méthode amie de la classe *Shape* qui reçoit en entrée deux vecteurs de type *Shape* contenant donc deux ensembles de figures à comparer. On utilise alors la surcharge de l'opérateur== pour comparer les deux ensembles et renvoyer en sortie la concordance ou non des deux ensembles.

→ "magnetisme" : cette méthode permet de magnétiser les deux pièces (celle du tracé (noir) et celle du joueur (couleur)) pour un meilleur rendu visuel dans le cas ou l'utilisateur ne positionne pas correctement une pièce du Tangram. Cette méthode utilise la surcharge de l'opérateur== pour renvoyer un booléen et si la condition est validée alors l'instance courante copie les valeurs de l'objet en paramètre, ce qui donne l'effet de magnétisme en jeu.

→ getters de *Shape* : pour chaque attribut on associe un getter. Il y a donc quatre getters, un pour la taille d'un côté, deux getters récupérant les positions en x et en y de l'objet et enfin un getter pour récupérer le point situé au centre de l'objet.

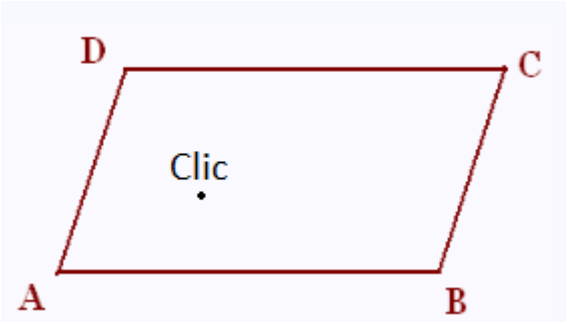
→ setter : on a choisi de faire un setter permettant d'attribuer des valeurs à l'ensemble des attributs de *Shape*. On se sert de ce setter notamment dans les constructeurs des classes filles dérivant de *Shape*.

→ "draw" et "drawAllShapes" : ces méthodes de dessins utilisant la bibliothèque MLV et notamment les méthodes MLV_draw. La méthodes "draw" permet de dessiner un objet *Shape* et fait partie de la classe *Shape* tandis que les deux méthodes "drawAllShapes" sont des méthodes amies utilisant la méthode "draw" pour dessiner des ensembles d'objets de type *Shape*. Il y a deux méthodes "drawAllShapes" car on souhaite pouvoir attribuer un vecteur de couleur ou bien une couleur unique à un ensemble d'objets *Shape*. On réalise les deux possibilités en surchargeant donc cette méthode. Ces méthodes sont déclarées en void et reçoivent en entrées des références des objets *Shape* dans un *std::vector,* puis la couleur, soit dans un *std::vector* si on veut attribuer des couleurs différentes à chaque objet, soit une couleur unique. Les couleurs étant des couleurs déclarées de type MLV_Color.

→ "display" et surcharge de l'opérateur<< : la méthode "display" permet d'afficher sur un flux de sortie les coordonnées d'un objet de type *Shape*. Elle

est déclarée en virtuel car on souhaite pouvoir personnaliser l'affichage sur le flux de sortie en fonction de l'objet fille considéré. La surcharge de l'opérateur de flux sortie fait alors appel à la méthode "display" pour afficher les informations.

→ "inside" : cette méthode reçoit en entrée une position x et y et retourne un booléen qui permet de savoir si le point de coordonnée x et y se trouve ou non dans l'objet *Shape*. On calcule ici les déterminants entre un point donné en entrée de la méthode et l'ensemble des vecteurs px et py de l'objet *Shape* courant. Afin d'assurer qu'un clic de la souris est bien dans l'objet *Shape* il faut alors que l'ensemble des déterminants soit positif.



Sur la figure on calculera alors le déterminant de la façon suivante. On pose x et y comme étant les clics de la souris et on déroule le calcul :

Coordonnées du vecteur AB : $Dx = Bx - Ax$

$Dy = By - Ay$

Coordonnées du vecteur entre A et le clic de la souris:

$Dx = x - Ax$

$Dy = y - Ay$

Déterminants associées : $Det = Dx * Ty - Dy * Tx$

Si l'ensemble des déterminants est positif alors on peut sélectionner l'objet et lui appliquer des changements.

→ "isInsideBoard" : c'est une méthode amie de la classe *Shape*. Cette méthode utilise le même principe et calcul de déterminant que la méthode "inside" de la classe *Shape*. Elle sert alors à vérifier lors d'un changement (translation/rotation) que l'objet modifié reste dans le cadre du jeu. Cette méthode est donc une méthode booléenne qui reçoit en entrée une position x et y ainsi que les données de la fenêtre de jeu. Elle renvoie en sortie un booléen sur l'appartenance ou non à la fenêtre du point(x,y). On utilise par exemple cette méthode dans les méthodes "rotateCenter" et "Translate" pour assurer que l'objet ne quittera donc pas la fenêtre, si un point est en dehors de la fenêtre, la méthode effectue un *break* pour sortir de la boucle for permettant le changement.

→ "reverse" : elle permet de faire la symétrie d'un objet. Cette méthode est déclarée virtuelle et de base n'effectue qu'une rotation de 180° pour des objets comme le carré/triangle dont la symétrie n'est qu'une rotation. Elle doit être redéfinie pour des figures plus complexes, par exemple pour le parallélogramme du Tangram. On a décidé d'implémenter la symétrie complète que dans les objets filles de *Shape* pour éviter d'effectuer des calculs non nécessaires et surtout car l'axe de symétrie dépend de l'objet considéré.

→ "rotateHW" : elle permet la rotation d'un objet autour d'une de ses coordonnées. On utilise cette méthode déclarée en public pour créer la figure de base du Tangram, mais dans le jeu la rotation est réalisée par la méthode "rotateCenter".

→ "rotateCenter" : elle permet la rotation d'un objet par rapport à son centre. Deux utilisations sont alors possibles en fonction des données en entrées.

- Soit seul un angle de rotation est donné auquel cas la rotation est faite dans le sens antihoraire en fonction de l'angle fourni.
- Soit quatre coordonnées sont données en entrées correspondant à un déplacement relative de la souris (coordonnées correspondant à deux positions de la souris espacées de quelques millisecondes).

Dans ce second cas on vient alors calculer les vecteurs U et V entre le centre de l'objet *Shape* et les deux positions relatives. On calcule alors le déterminant et le produit scalaire des vecteurs U et V pour connaître l'angle de rotation et sa direction (positive ou négative). On applique ensuite la rotation à l'ensemble des vecteurs px et py de l'objet *Shape* courant. Avec la formule de rotation en 2D :

$$R(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$

On vérifie à chaque rotation que l'objet est contenu dans la fenêtre de jeu avec la méthode "isInsideBoard" et si ce n'est pas le cas on annule la rotation.

→ "translate" : qui réalise la translation d'un polygone par les coordonnées de la souris (x, y) données en entrée. Comme pour la méthode "rotateCenter" les données x et y sont des entiers car la méthode pour récupérer la position de la souris que l'on utilise est la méthode MLV_get_mouse_position de la bibliothèque MLV qui renvoie des positions entières.

On applique donc la translation par x et y à l'ensemble des vecteurs px et py ainsi qu'au centre de l'objet et on vérifie que l'objet est dans le cadre du jeu avec la méthode "isInsideBoard".

→ foncteur pour "translate" et "operator()" : on a décidé de faire un foncteur sur la méthode translate en surchargeant l'opérateur (). Même si l'on ne s'en sert pas concrètement dans notre projet, le foncteur permet alors grâce à la surcharge de l'opérateur() d'agir comme une fonction et d'être passé en argument d'une méthode ou bien d'une autre fonction.

→ "moveShape" : c'est une méthode permettant le déroulement du jeu et composée d'une boucle while qui s'arrête quand un objet est désélectionné, elle reçoit en entrée :

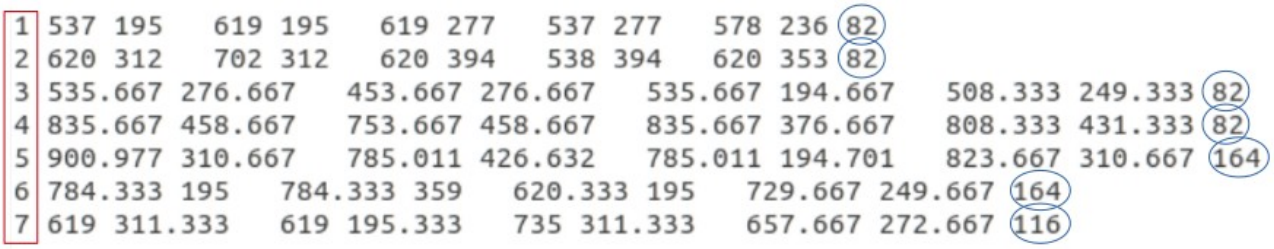
- une position x et y de la souris,
- un *std::vector* de pointeurs intelligents d'une figure motif à reproduire par le second *std::vector* étant les objets que l'utilisateur peut bouger dans le jeu,
- les couleurs des objets sous formes de *std::list*,
- un itérateur de *std::list* permettant d'obtenir la couleur de l'objet à modifier afin d'assombrir celui-ci pour savoir si un objets est sélectionné ou non,
- un booléen sur la condition de victoire pour connaître l'état actuel du jeu et dessiner la figure avant d'effectuer un changement.

Cette méthode permet donc grâce à l'ensemble des méthodes de la classe *Shape* de dérouler le déplacement des pièces du jeu en fonction des actions de l'utilisateur.
Elle permet ainsi d'associer les clics de la souris à différentes méthodes :

- bouton gauche de la souris : permet d'effectuer la translation de l'objet en appelant la méthode translate.
- bouton gauche + droit de la souris : permet en plus de la translation d'appeler la méthode "rotateCenter" et d'effectuer une rotation fixe de 1 degré continuellement pour placer une pièce.
- bouton droit de la souris : permet d'appeler la méthode "rotateCenter" mais avec le déplacement relatif de la souris.
- bouton milieu (clic molette de la souris) : fait appel à la méthode "reverse" où à sa surcharge dans les méthodes filles pour effectuer la symétrie d'un objet.

La méthode "moveShape" fait alors appel aux méthodes "drawAllShapes" / "draw" pour redessiner les figures (le motif, la figure déplacée ainsi que les autres pièces de l'utilisateur) à chaque fois qu'un déplacement est effectué. De plus on vérifie à chaque tour de boucle si un "magnétisme" doit être effectué. Enfin en sortie de la boucle si la pièce est désélectionnée, on lui réattribue sa couleur d'origine.

→ "saveDraw" : qui permet de sauvegarder une figure contenue dans un *std::vector*. Cette méthode permet d'ouvrir le fichier texte "savefig.txt" et d'enregistrer une figure en écrivant le n° de l'objet associé puis les coordonnées de chaque objets composant la figure.
Un exemple de figure sauvegardée aura alors la forme suivante dans le fichier :



On y retrouve les numéros de figures associées (rouge) puis ensuite les coordonnées de chaque point en x puis en y, ensuite vient la position en x et y du centre de l'objet et le dernier chiffre d'une ligne correspond à la taille du coté de construction de l'objet (bleu).

C – Classe Parallelogram, Square et RigthTriangle

Ces trois classes héritent de *Shape* et permettent de dessiner un parallélogramme, un carré ou un triangle rectangle, étant les figures du Tangram. Ces trois méthodes créent donc des objets filles de *Shape et* sont alors relativement similaires et disposent quasiment des mêmes méthodes.

Ces méthodes sont :

→ constructeur à trois paramètres : on commence par initialiser les objets en appelant le constructeur de la classe *Shape*. Ensuite pour chaque figure on remplit les vecteurs de coordonnées de façon à les construire. Pour initialiser ces vecteurs on appelle la classe template *Point* qui permet avec deux listes d'initialisation des vecteurs de points. On attribue ensuite le centre correspondant à chaque objets et enfin on utilise la méthode "setShape" de *Shape* pour construire chaque objets avec les données calculées.

→ destructeur virtuel : chaque objet dispose d'un destructeur déclaré en virtuel.

→ constructeur de copie et opérateur= : ici on utilise la conversion implicite pour le constructeur de copie et on y fait appel pour surcharger l'opérateur=.

→ "display": ici on redéfini la méthode virtuelle "display" de la classe *Shape.* Chaque classe dispose d'une redéfinition de la méthode virtuelle "display" pour afficher le type d'objet que l'on regarde. Cette méthode fait appel à la méthode "display" de la classe *Shape* pour afficher les coordonnées de position de l'objet.

→ "reverse" : ici on redéfinit la méthode virtuelle "reverse" de la classe *Shape.* Si le symétrique d'un objet est plus complexe qu'une rotation par 180° alors les objets filles de la classe *Shape* peuvent redéfinir la méthode virtuelle "reverse". Pour le Tangram la seule classe d'objet nécessitant de redéfinir cette méthode est la classe *Parallelogram*.
Il faut, pour le parallélogramme, réaliser la symétrie par rapport à un axe de symétrie car une rotation de 180° n'est donc pas une symétrie de parallélogramme.
On utilise alors un booléen pour savoir si une symétrie a déjà été effectuée de façon à alors effectuer la symétrie inverse. Une fois le calcul effectué on réaffecte les valeurs de l'objet avec la méthode "setShape" et on inverse l'état du booléen "isRevert".

2. Interface du jeu

A – Classe Button

C'est une classe template qui permet la création de bouton pour les fonctionnalités de l'Interface.

Ces attributs sont :

→ la coordonnée nord ouest du bouton (x, y),
→ la longueur et la hauteur de la fenêtre,
→ le texte sur le bouton,
→ l'espacement des boutons (si nécessaire),
→ les couleurs du texte, du bouton et de la bordure du bouton.

Ces méthodes sont :

→ un constructeur sans aucun paramètre,
→ un constructeur à cinq paramètres (les cinq premiers attributs),
→ un destructeur virtuel,
→ une méthode "drawButton" qui permet de dessiner un bouton en appelant la méthode MLV_draw_text_box de la libMLV,
→ une méthode "inButton" qui permet de savoir si le clic de la souris est sur le bouton en fonction du clic de la souris et de la taille de la fenêtre.

B – Classe Menu

C'est la classe qui permet l'affichage du menu de départ.

Ces attributs sont :

→ la longueur et la hauteur de la fenêtre,
→ un vecteur de boutons qui permet le stockage des boutons nécessaires.

Ces méthodes sont :

→ un constructeur à deux paramètres (les deux premiers attributs),

→ un destructeur virtuel,

→ une méthode "drawMenu" qui permet de dessiner le menu, en créant les boutons nécessaires et les ajouter dans un *std::vector* pour pouvoir ensuite les parcourir pour tous les dessiner en même temps,

→ une méthode "inMenu" qui permet de connaître le choix de l'utilisateur en fonction du clic de la souris.

C – Classe Board

C'est la classe qui permet l'affichage du plateau de jeu.

Ces attributs sont :

→ la longueur et la hauteur de la fenêtre,

→ un vecteur de boutons qui permet le stockage des boutons nécessaires.

Ces méthodes sont :

→ un constructeur à deux paramètres (les deux premiers attributs),

→ un destructeur virtuel,

→ une méthode "drawBoard" qui permet de dessiner le plateau de jeu avec la méthode MLV_draw_filled_rectangle de la libMLV,

→ une méthode "drawButtonForBoard" qui permet de dessiner les boutons nécessaires et les ajouter dans un *std::map* pour pouvoir ensuite les parcourir, pour tous les dessiner en même temps,

→ une méthode "createButtonForBoard" qui permet de remplir un objet *std::map* avec un index et un objet de la classe temple *Button*, afin de classer nos boutons sur la fenêtre.

→ une méthode "inBoard" qui permet de connaître le choix de l'utilisateur pour les boutons utilisés en fonction du clic de la souris.

D - Classe Interface

C'est la classe qui permet la "fusion" des deux grandes parties de ce projet.

Ces méthodes sont :

→ une méthode "initialiseShared" qui permet de créer les pièces du Tangram à utiliser pour construire une figure à l'aide de la classe *Shape* et des enfants *RigthTriangle*, S*quare* et P*arallelogram*. Cette méthode renvoie un vecteur de S*hape* de pointeur sur chaque objet. On utilise des pointeurs car nos classes utilisent de l'héritage et des méthodes virtuelles.

→ une méthode "initialiseMotif" qui est similaire à la méthode "initialiseShared" et permet en fait de créer un motif de base pour le Tangram.

→ une méthode "clone" qui permet de réaliser la copie d'un vecteur de S*hape.*

→ une méthode "move" qui permet de réaliser le déplacement sémantique d'un vecteur de pointeur de Shape.

→ une méthode "importDraw" qui permet d'importer des figures d'un fichier texte regroupant les coordonnées et attributs d'un motif de Tangram.

→ une méthode "saveFigure" qui permet d'effectuer une sauvegarde d'un motif qu'un utilisateur pourrait faire dans le jeu.

→ une méthode "drawJeu" qui permet d'afficher l'ensemble des figures, motifs et pièces, et d'appeler les différentes méthodes de la classe *Shape* et de ces enfants. Cette méthode est le cœur de l'interface et c'est dedans que l'on vient créer/afficher les objets ainsi que l'on appelle les différentes méthodes de déplacement d'objets et d'égalité entre deux ensembles de figures. De plus on vient programmer les différents boutons nécessaire au jeu grâce à la classe *Button*, des méthodes de l'interfaces et de la classe *Shape*.

V. Conclusion

Lors de ce projet nous avons pu mettre en application les connaissances de programmation orientée objets liées au C++. En développant un jeu tel qu'un Tangram nous avons choisi une conception avec une classe mère regroupant les autres éléments du Tangram. On est ainsi amené à faire de l'héritage sur ses enfants, ce qui nous à permis de mettre en place un certain nombre des notions du cours.
Nous avons commencé le développement de ce projet assez tôt et certaines contraintes du sujets sont arrivées après une bonne partie de notre développement.
On a ainsi dû, à plusieurs reprises, reprendre le code pour ajouter les différentes fonctionnalités (contraintes) qui ont été données au milieu du processus de développement. Ce qui nous a poussés à rajouter ou modifier l'architecture fonctionnelle, d'où le fait d'avoir deux versions d'UML dans la partie Diagramme des classes.
Notre code actuel est en couplage fort avec la librairie graphique MLV.
Une des solutions possibles pour éviter ce couplage fort est que toutes les classes qui s'occupent des créations et mouvements de forme renvoient les valeurs numériques associées à d'autres classes qui s'occuperaient exclusivement de l'affichage sur l'interface.
Une autre amélioration possible vient sur la conception des pièces du Tangram. En effet nous construisons chaque objet avec un nombre de points qui lui est propre, par exemple un carré avec 4 points. Cela pose un léger problème dans la condition de validation du Tangram, nous pensons qu'une « meilleure » solution pour valider un dessin aurait été de découper chaque figure en un ensemble de plus petits triangles. En transformant chaque objet du Tangram en un ensemble de plus petits triangles on aurait pu par exemple interchanger le parallélogramme et les deux petits triangles du Tangram afin de réaliser la même solution, ce qui n'est pas possible actuellement.

VI. Bibliographie

→ Tangram :

https://fr.wikipedia.org/wiki/Tangram

→ Librairie MLV :

http://www-igm.univ-mlv.fr/~boussica/mlv/api/French/html/index.html

VII. Annexe

1. Instructions de compilation

Pour compiler le programme avec les modules il suffit de lancer le makefile par la commande **make** dans le terminal.
Attention quand vous récupérez le zip (le rendu) vérifiez bien que vous avez un dossier "bin" au même niveau que les dossiers "src" et "include" car sinon le programme ne compilera pas.

2. Documentations / Instructions d'utilisation

Pour lancer le programme il suffit de taper ./Tangram dans le terminal.
Ensuite une fenêtre s'ouvre avec une image et deux boutons, voir Mode d'emploi et le README.md pour la suite.

3. Utilisation des notions du cours

A – Chapitre 1

→ **Surcharge de fonction**

Fichier / Numéros de ligne : shape.hpp / l.285-286

Exemple(s) de code :

```
void drawAllShapes(const std::vector<std::shared_ptr<geometricShape::Shape>>
&vectShapes, MLV_Color colorShape, MLV_Color colorBorder);
            */
void drawAllShapes(const std::vector<std::shared_ptr<geometricShape::Shape>>
&vectShapes, std::list<MLV_Color> colorShapes, MLV_Color colorBorder);
```

B – Chapitre 2

→ **Liste d'initialisations**

Fichier / Numéros de ligne : shape.cpp / l.78

Exemple(s) de code :

```
geometricShape::Shape::Shape(const Shape & tc) : sizeCote(tc.sizeCote),
px(tc.px), py(tc.py), center(tc.center) {}
```

→ ***Mot-clé mutable***

Fichier / Numéros de ligne :

Exemple(s) de code

→ **Attribut static**

Fichier / Numéros de ligne : shape.hpp / l.39

Exemple(s) de code :

```
static constexpr float PI = 3.14159265358979323846264338327 9;
```

→ ***Méthode static***

Fichier / Numéros de ligne :

Exemple(s) de code

→ ***R.A.I.I.***

Fichier / Numéros de ligne :

Exemple(s) de code

C – Chapitre 3 Surcharge d'opérateur

→ **Surcharge d'opérateur (hors << et >>)**

Fichier / Numéros de ligne : shape.hpp / l.72 et l.79

Exemple(s) de code :

```
Shape & operator=(const Shape &cshape);
bool operator==(const Shape &cshape) const;
```

D – Chapitre 4 Héritage, Polymorphisme

→ **fonctions virtuelles**

Fichier / Numéros de ligne : shape.hpp / l.162 et l.187

Exemple(s) de code

```
virtual void display(std::ostream & out = std::cout) const;
virtual void reverse();
```

→ **override**

Fichier / Numéros de ligne : parallelogram.hpp / l.61 et 69

Exemple(s) de code

```
void display(std::ostream & out = std::cout) const override;
void reverse() override;
```

→ ***final (classe, méthode)***

Fichier / Numéros de ligne :

Exemple(s) de code

E – Chapitre Template, STL

→ **Template**

Fichier / Numéros de ligne : point.hpp et button.hpp

Exemple(s) de code : Tout le fichier.

→ **Les classes de la STL**

~ classe Map

Fichier / Numéros de ligne : board.hpp / l.31

Exemple(s) de code :

```
std::map<int, Button<int>> buttons;
```

~ classe List

Fichier / Numéros de ligne : interface.cpp / l.19

Exemple(s) de code

```
std::list<MLV_Color> figcolor() {
      std::list<MLV_Color> figloc;
      figloc.push_back(MLV_COLOR_GREEN);
      figloc.push_back(MLV_COLOR_PINK);
      figloc.push_back(MLV_COLOR_ORANGE);
      figloc.push_back(MLV_COLOR_BLUE);
      figloc.push_back(MLV_COLOR_CYAN);
      figloc.push_back(MLV_COLOR_MAGENTA);
      figloc.push_back(MLV_COLOR_YELLOW);
      return figloc;
}
```

~ classe Vector

Fichier / Numéros de ligne : interface.cpp / l.118 à 120

Exemple(s) de code :

```
std::vector<double> pxloc;
std::vector<double> pyloc;
std::vector<double> centloc;
```

F – Chapitre Foncteur

→ **Foncteur**

Fichier / Numéros de ligne : shape.hpp / l.261

Exemple(s) de code

```
 void operator()(int x, int y);
```

G – Chapitre divers (dont auto, lambda)

→ ***énumération***

Fichier / Numéros de ligne :

Exemple(s) de code

→ ***static_assert***

Fichier / Numéros de ligne :

Exemple(s) de code

→ **assert**

Fichier / Numéros de ligne : point.hpp / l.80

Exemple(s) de code :

```
assert(list_y.size() == list_x.size());
```

→ **délégation de constructeurs**

Fichier / Numéros de ligne : rigth_triangle.cpp / l.19

Exemple(s) de code

```
geometricShape::RigthTriangle::RigthTriangle(double sizeCote, int width, int
height) : Shape(sizeCote, width, height), hauteur(0) {
```

→ ***héritage de constructeurs avec mot-clé using***

Fichier / Numéros de ligne :

Exemple(s) de code

→ **inférence de type : mot-clé auto**

Fichier / Numéros de ligne : interface.cpp / l.269

Exemple(s) de code

```
auto fig_i = colorfig.begin();
```

→ ***inférence de type : mot-clé decltype***

Fichier / Numéros de ligne :

Exemple(s) de code

→ **itérateur : begin, end**

Fichier / Numéros de ligne : interface.cpp / l.270

Exemple(s) de code

```
std::for_each(fig.begin(), fig.end(), [&](std::shared_ptr<geometricShape::Shape>
it){
	it-> moveShape(xInside, yInside, fig, motif, motifShape, motifBorder ,
	colorfig,fig_i, board,wincondi);
	drawAllShapes(motif, motifShape, motifBorder);
	drawAllShapes(fig, colorfig, MLV_COLOR_RED);
	fig_i++;
});
```

→ ***itérateur : cbegin, cend***

Fichier / Numéros de ligne :

Exemple(s) de code

→ **boucle foreach**

Fichier / Numéros de ligne : board.cpp / l.52

Exemple(s) de code

```
unsigned int count = 0;
for (auto button : buttons) {
	bool ibout = button.second.inButton(pX, pY);
	if (ibout == true) {
		break;
	}
	count++;
}
```

→ **liste d'initialisateurs**

Fichier / Numéros de ligne : point.hpp / l.78

Exemple(s) de code :

```
template <class T>
Point<T>::Point(std::initializer_list<T> list_x, std::initializer_list<T>
list_y) : _px(list_x), _py(list_y)
```

→ **mot-clé explicit pour un constructeur**

Fichier / Numéros de ligne : shape.hpp / l.48

Exemple(s) de code

```
explicit Shape(double sizeCote, int width, int height);
```

→ ***mot-clé explicit pour un opérateur***

Fichier / Numéros de ligne :

Exemple(s) de code

→ **lambda**

Fichier / Numéros de ligne : interface.cpp / l.270

Exemple(s) de code

```
std::for_each(fig.begin(), fig.end(), [&](std::shared_ptr<geometricShape::Shape>
it){
      it-> moveShape(xInside, yInside, fig, motif, motifShape, motifBorder ,
      colorfig,fig_i, board,wincondi);
      drawAllShapes(motif, motifShape, motifBorder);
      drawAllShapes(fig, colorfig, MLV_COLOR_RED);
      fig_i++;
});
```

→ ***std::sort***

Fichier / Numéros de ligne :

Exemple(s) de code

H – Chapitre constexpr

→ **constexpr**

Fichier / Numéros de ligne : shape.hpp / l.39 et button.hpp / l.48

Exemple(s) de code :

```
static constexpr float PI = 3.14159265358979323846264338327 9;
constexpr Button();
```

I – Chapitre pointeurs intelligents

→ ***unique_ptr***

Fichier / Numéros de ligne :

Exemple(s) de code

→ **shared_ptr**

Fichier / Numéros de ligne : shape.cpp / l.49

Exemple(s) de code :

```
shapeShared.push_back(std::make_shared<geometricShape::Square>(size, allWidth + 70, allHeigth + 100));
```

→ ***weak_ptr***

Fichier / Numéros de ligne :

Exemple(s) de code

J – Chapitre Exceptions

→ **Exceptions**

Fichier / Numéros de ligne : inerface.cpp / l.48 à 62

Exemple(s) de code

```
try {
        ...
} catch (std::bad_alloc & e) {
      std::cerr << "bad_alloc caught: " << e.what() << '\n';
      exit(EXIT_FAILURE);
}
```

K – Chapitre réf universelles, sémantique de déplacement

→ **fonction move**

Fichier / Numéros de ligne : interface.cpp / l.104

Exemple(s) de code

```
void Interface::move(std::vector<std::shared_ptr<geometricShape::Shape>> &move, std::vector<std::shared_ptr<geometricShape::Shape>> &shapeShared) {
    move = std::move(shapeShared);
}
```

→ ***constructeur de déplacement***

Fichier / Numéros de ligne :

Exemple(s) de code

4. Utilisation de design pattern

→ **Design Pattern Composite**

fichier square, parllelogram et rigth_triangle qui héritent de shape.