RAPPORT DU PROJET TANGRAM C++

Table des matières

Introduction

Ce projet consiste à l'implémentation du jeu du Tangram en C++. L'objectif étant d'utiliser les notions vues en cours.

Pour ce projet j'ai fait le choix d'utiliser la bibliothèque graphique MLV, un outil que je connais bien depuis quelques années désormais, c'était donc un choix logique.

Les objectifs fixés pour ce projet sont, en plus du jeu, un menu pour jouer/créer un puzzle, une navigation de pages dans le choix des puzzles, un mode permettant de créer des puzzles.

Les limites du projet sont :

- Pas de fonctionnalités pour supprimer un puzzle crée via l'interface graphique.
- De paramétrer l'affichage précisément. Il n'y a qu'une configuration par défaut qui correspond à un affichage « responsive » en fonction de la taille de l'écran de l'utilisateur.
- Pas de possibilité de renommer un puzzle via l'interface graphique.

J'entends par limite le fait de n'avoir pas implémenté certaines fonctionnalités. Dans ce cas précis aucune n'était demandée à l'origine.

L'analyse fonctionnelle générale

Le programme peut être généralisé ainsi :

- Un module graphique dit Drawabl, en charge de l'affichage des éléments liés à l'expérience utilisateurs. (Boutons, menus, pièces etc)
- Un module de gestion des pièces, nommé Shape, permettant ainsi d'effectuer toutes les opérations nécessaires pour le déroulement du jeu.
- Un module Game qui est en fait le cœur du jeu, qui définit les commandes autorisées et qui lie ces dernières à l'opération souhaitée de la figure. Il définit également le comportement des pièces entre elles (collision), et certains ajustements de jouabilité tel que le « stick » permettant de coller une pièce à un endroit afin d'être sûr que le jeu ne se joue pas au pixel près.
- Un module Parser permettant de sauvegarder un puzzle une fois édité (mode création de puzzle) ou de charger un puzzle existant afin de le jouer.
- Un module utility ne contenant qu'une classe template Point.

UML

L'UML est accessible en ligne sur le github de mon projet :

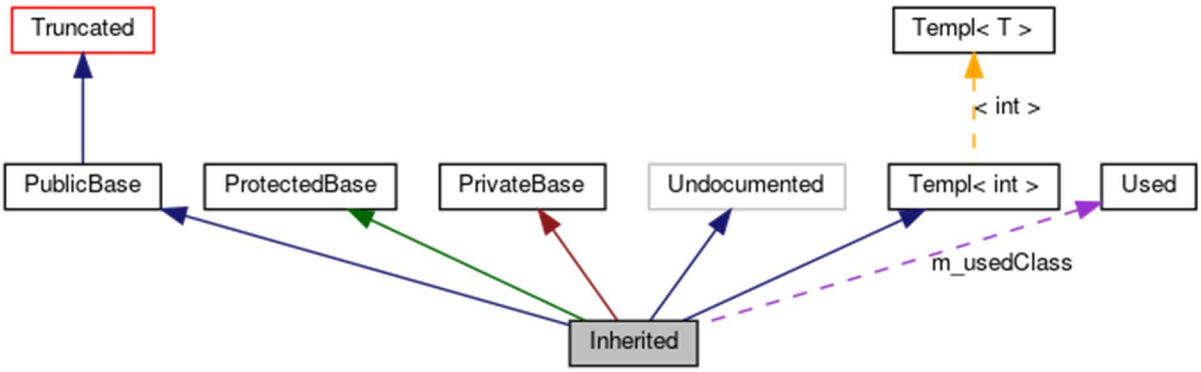
- https://github.com/jlebas01/Tangram/blob/master/Software%20Modeling.pdf

Analyse fonctionnelle détaillée

Graphe d'inclusion et dépendances des classes :

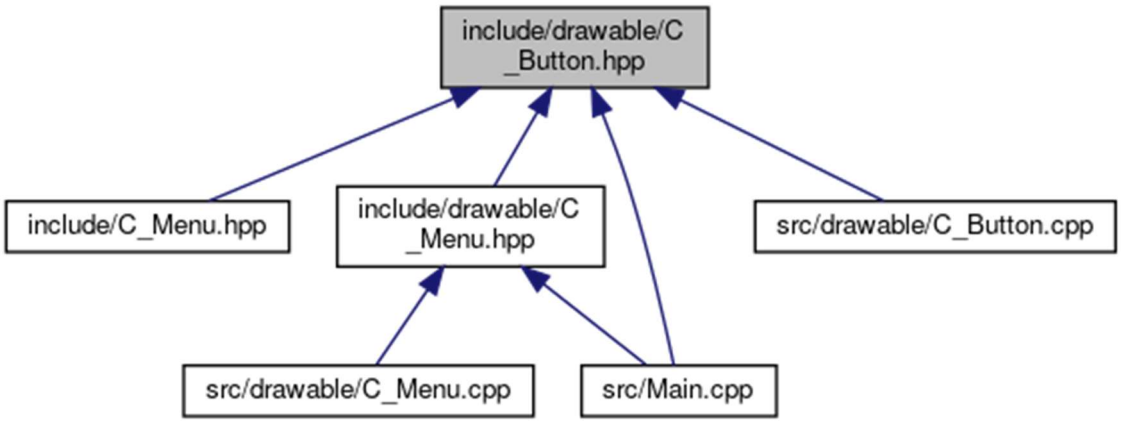
Avant de détailler les différents modules, leurs usages et leurs classes, voici les graphes des différentes classes / interfaces afin d'avoir une vue d'ensemble du projet.

Légende :

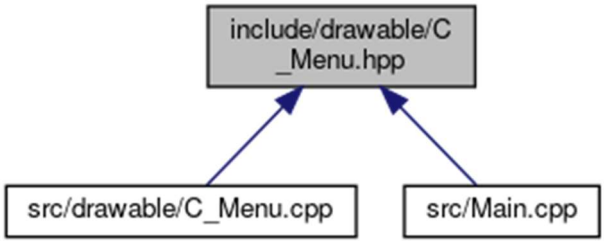


Module Drawable :

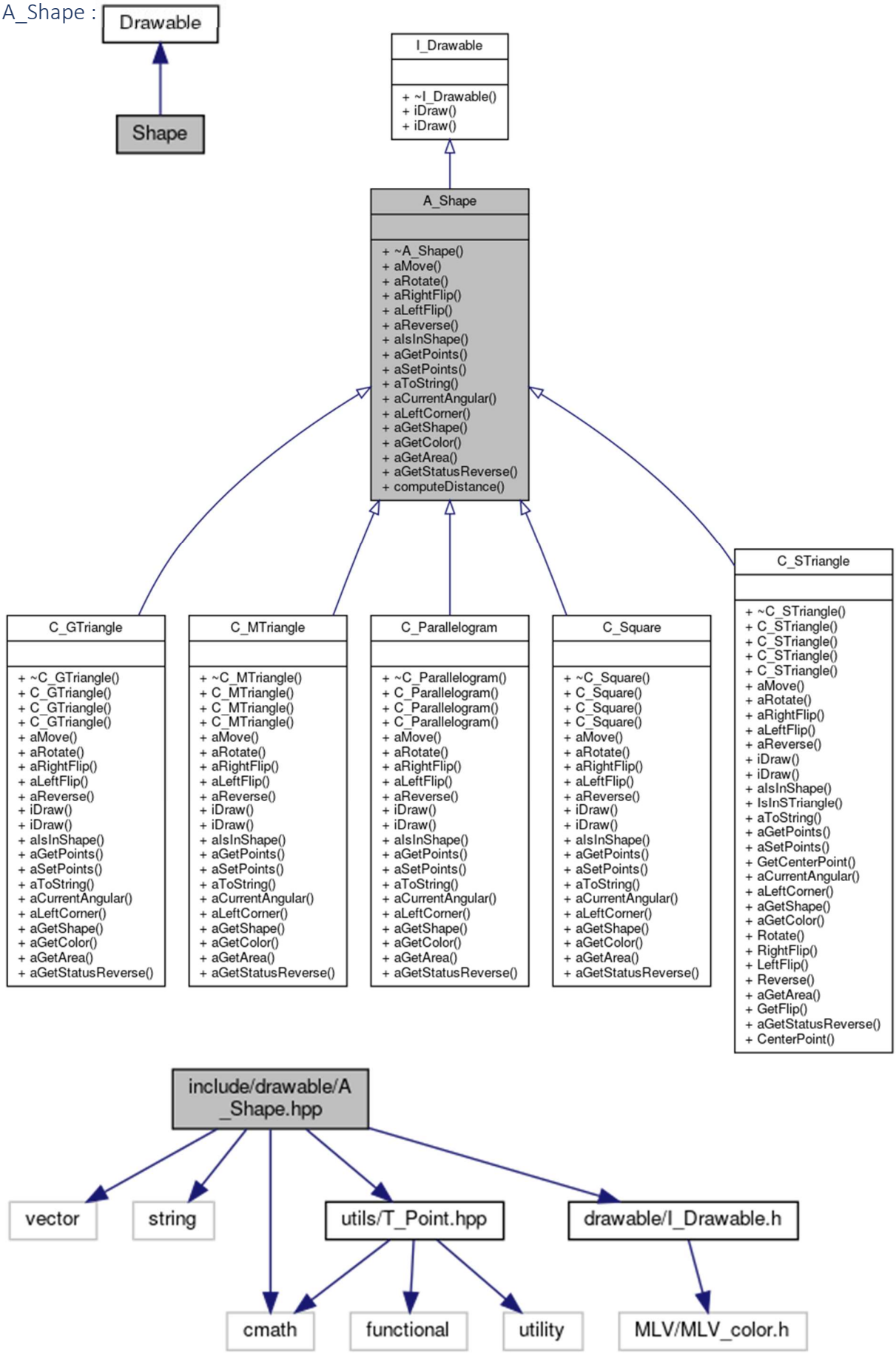
C_Button :

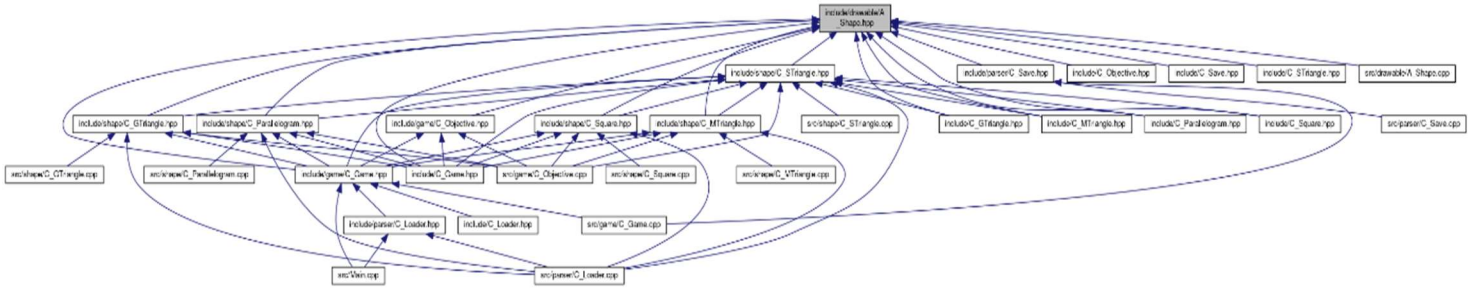


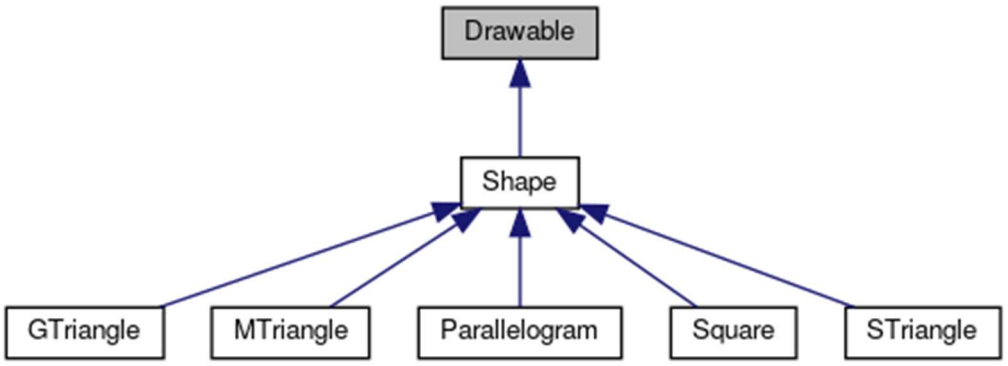
C_Menu :



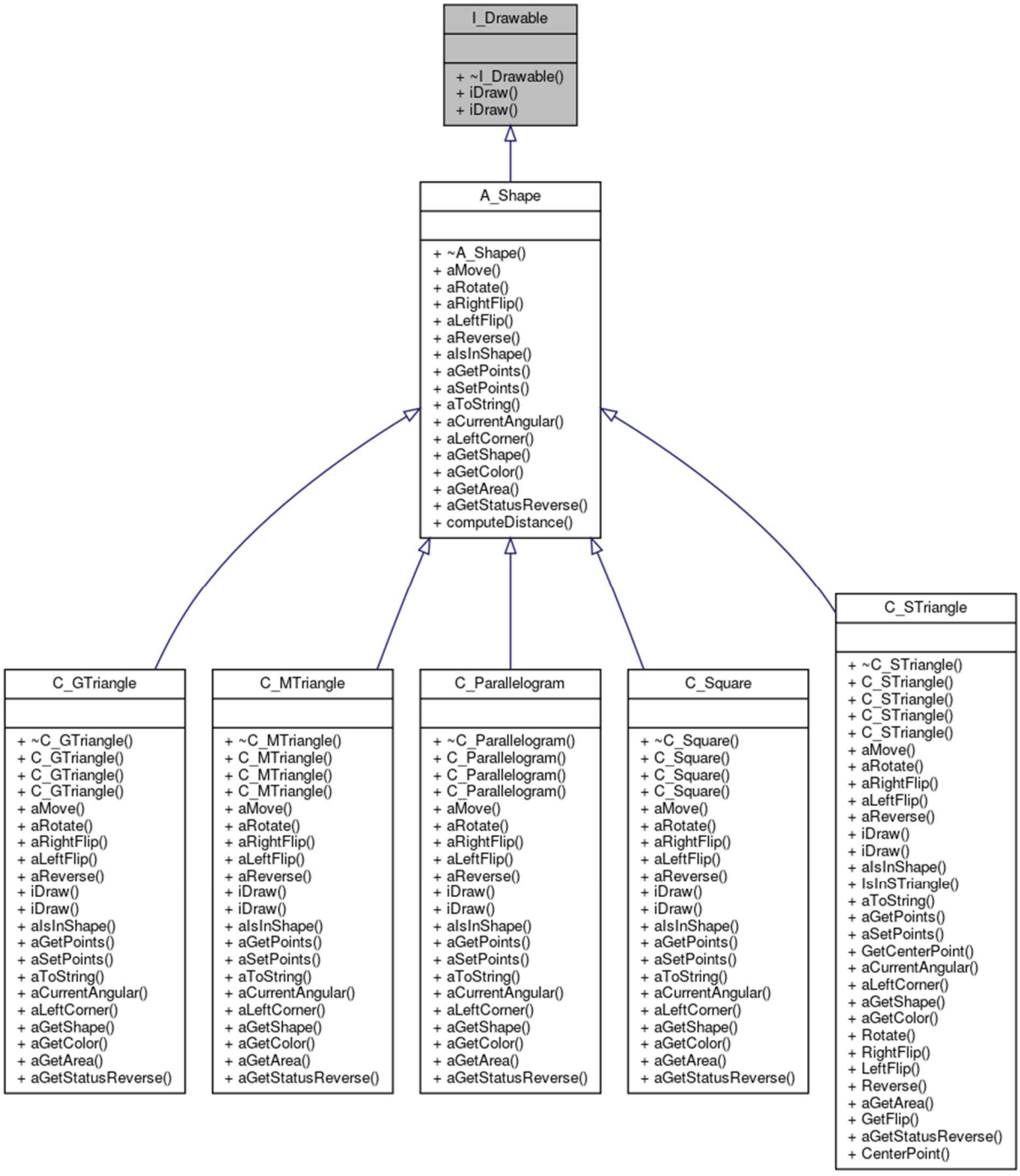
A_Shape :


Drawable
Shape
I_Drawable
+ ~I_Drawable()
+ iDraw()
+ iDraw()
A_Shape
+ ~A_Shape()
+ aMove()
+ aRotate()
+ aRightFlip()
+ aLeftFlip()
+ aReverse()
+ aIsInShape()
+ aGetPoints()
+ aSetPoints()
+ aToString()
+ aCurrentAngular()
+ aLeftCorner()
+ aGetShape()
+ aGetColor()
+ aGetArea()
+ aGetStatusReverse()
+ computeDistance()
C_GTriangle
+ ~C_GTriangle()
+ C_GTriangle()
+ C_GTriangle()
+ C_GTriangle()
+ aMove()
+ aRotate()
+ aRightFlip()
+ aLeftFlip()
+ aReverse()
+ iDraw()
+ iDraw()
+ aIsInShape()
+ aGetPoints()
+ aSetPoints()
+ aToString()
+ aCurrentAngular()
+ aLeftCorner()
+ aGetShape()
+ aGetColor()
+ aGetArea()
+ aGetStatusReverse()
C_MTriangle
+ ~C_MTriangle()
+ C_MTriangle()
+ C_MTriangle()
+ C_MTriangle()
+ aMove()
+ aRotate()
+ aRightFlip()
+ aLeftFlip()
+ aReverse()
+ iDraw()
+ iDraw()
+ aIsInShape()
+ aGetPoints()
+ aSetPoints()
+ aToString()
+ aCurrentAngular()
+ aLeftCorner()
+ aGetShape()
+ aGetColor()
+ aGetArea()
+ aGetStatusReverse()
C_Parallelogram
+ ~C_Parallelogram()
+ C_Parallelogram()
+ C_Parallelogram()
+ C_Parallelogram()
+ aMove()
+ aRotate()
+ aRightFlip()
+ aLeftFlip()
+ aReverse()
+ iDraw()
+ iDraw()
+ aIsInShape()
+ aGetPoints()
+ aSetPoints()
+ aToString()
+ aCurrentAngular()
+ aLeftCorner()
+ aGetShape()
+ aGetColor()
+ aGetArea()
+ aGetStatusReverse()
C_Square
+ ~C_Square()
+ C_Square()
+ C_Square()
+ C_Square()
+ aMove()
+ aRotate()
+ aRightFlip()
+ aLeftFlip()
+ aReverse()
+ iDraw()
+ iDraw()
+ aIsInShape()
+ aGetPoints()
+ aSetPoints()
+ aToString()
+ aCurrentAngular()
+ aLeftCorner()
+ aGetShape()
+ aGetColor()
+ aGetArea()
+ aGetStatusReverse()
C_STriangle
+ ~C_STriangle()
+ C_STriangle()
+ C_STriangle()
+ C_STriangle()
+ C_STriangle()
+ aMove()
+ aRotate()
+ aRightFlip()
+ aLeftFlip()
+ aReverse()
+ iDraw()
+ iDraw()
+ aIsInShape()
+ IsInSTriangle()
+ aToString()
+ aGetPoints()
+ aSetPoints()
+ GetCenterPoint()
+ aCurrentAngular()
+ aLeftCorner()
+ aGetShape()
+ aGetColor()
+ Rotate()
+ RightFlip()
+ LeftFlip()
+ Reverse()
+ aGetArea()
+ GetFlip()
+ aGetStatusReverse()
+ CenterPoint()
include/drawable/A_Shape.hpp
vector
string
utils/T_Point.hpp
drawable/I_Drawable.h
cmath
functional
utility
MLV/MLV_color.h


include/drawable/A Shape.hpp
include/shape/C_STriangle.hpp
include/parser/C_Save.hpp
include/C_Objective.hpp
include/C_Save.hpp
include/C_STriangle.hpp
src/drawable/A_Shape.cpp
include/shape/C_GTriangle.hpp
include/shape/C_Parallelogram.hpp
include/game/C_Objective.hpp
include/shape/C_Square.hpp
include/shape/C_MTriangle.hpp
src/shape/C_STriangle.cpp
include/C_GTriangle.hpp
include/C_MTriangle.hpp
include/C_Parallelogram.hpp
include/C_Square.hpp
src/parser/C_Save.cpp
src/shape/C_GTriangle.cpp
src/shape/C_Parallelogram.cpp
include/game/C_Game.hpp
include/C_Game.hpp
src/game/C_Objective.cpp
src/shape/C_Square.cpp
src/shape/C_MTriangle.cpp
include/parser/C_Loader.hpp
include/C_Loader.hpp
src/game/C_Game.cpp
src/Main.cpp
src/parser/C_Loader.cpp

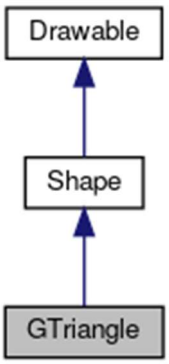

Drawable
Shape
GTriangle
MTriangle
Parallelogram
Square
STriangle

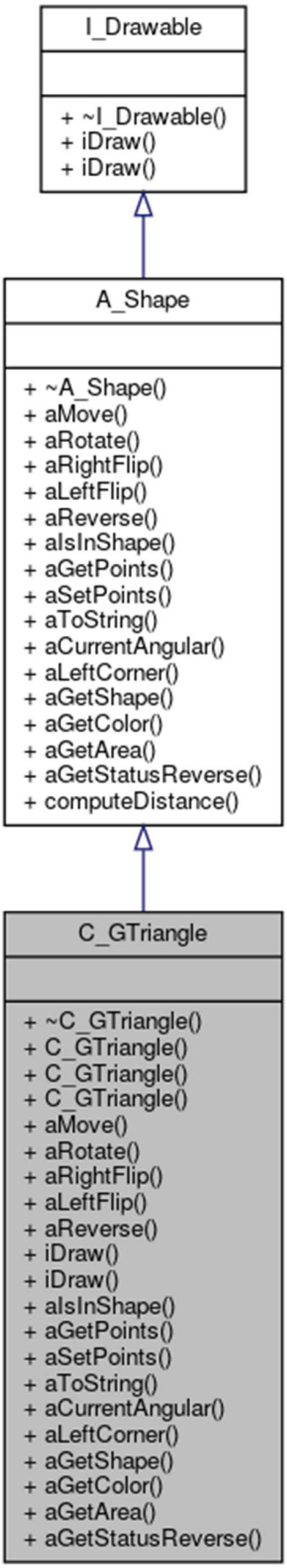
I_Drawable :

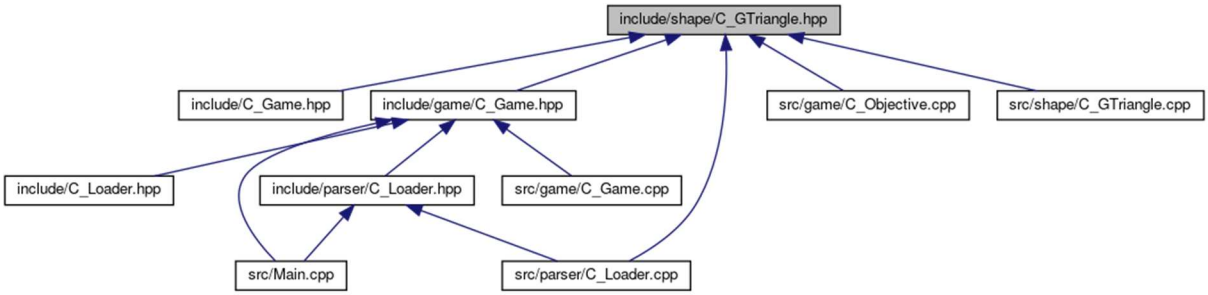

I_Drawable
+ ~I_Drawable()
+ iDraw()
+ iDraw()
A_Shape
+ ~A_Shape()
+ aMove()
+ aRotate()
+ aRightFlip()
+ aLeftFlip()
+ aReverse()
+ aIsInShape()
+ aGetPoints()
+ aSetPoints()
+ aToString()
+ aCurrentAngular()
+ aLeftCorner()
+ aGetShape()
+ aGetColor()
+ aGetArea()
+ aGetStatusReverse()
+ computeDistance()
C_GTriangle
+ ~C_GTriangle()
+ C_GTriangle()
+ C_GTriangle()
+ C_GTriangle()
+ aMove()
+ aRotate()
+ aRightFlip()
+ aLeftFlip()
+ aReverse()
+ iDraw()
+ iDraw()
+ aIsInShape()
+ aGetPoints()
+ aSetPoints()
+ aToString()
+ aCurrentAngular()
+ aLeftCorner()
+ aGetShape()
+ aGetColor()
+ aGetArea()
+ aGetStatusReverse()
C_MTriangle
+ ~C_MTriangle()
+ C_MTriangle()
+ C_MTriangle()
+ C_MTriangle()
+ aMove()
+ aRotate()
+ aRightFlip()
+ aLeftFlip()
+ aReverse()
+ iDraw()
+ iDraw()
+ aIsInShape()
+ aGetPoints()
+ aSetPoints()
+ aToString()
+ aCurrentAngular()
+ aLeftCorner()
+ aGetShape()
+ aGetColor()
+ aGetArea()
+ aGetStatusReverse()
C_Parallelogram
+ ~C_Parallelogram()
+ C_Parallelogram()
+ C_Parallelogram()
+ C_Parallelogram()
+ aMove()
+ aRotate()
+ aRightFlip()
+ aLeftFlip()
+ aReverse()
+ iDraw()
+ iDraw()
+ aIsInShape()
+ aGetPoints()
+ aSetPoints()
+ aToString()
+ aCurrentAngular()
+ aLeftCorner()
+ aGetShape()
+ aGetColor()
+ aGetArea()
+ aGetStatusReverse()
C_Square
+ ~C_Square()
+ C_Square()
+ C_Square()
+ C_Square()
+ aMove()
+ aRotate()
+ aRightFlip()
+ aLeftFlip()
+ aReverse()
+ iDraw()
+ iDraw()
+ aIsInShape()
+ aGetPoints()
+ aSetPoints()
+ aToString()
+ aCurrentAngular()
+ aLeftCorner()
+ aGetShape()
+ aGetColor()
+ aGetArea()
+ aGetStatusReverse()
C_STriangle
+ ~C_STriangle()
+ C_STriangle()
+ C_STriangle()
+ C_STriangle()
+ C_STriangle()
+ aMove()
+ aRotate()
+ aRightFlip()
+ aLeftFlip()
+ aReverse()
+ iDraw()
+ iDraw()
+ aIsInShape()
+ IsInSTriangle()
+ aToString()
+ aGetPoints()
+ aSetPoints()
+ GetCenterPoint()
+ aCurrentAngular()
+ aLeftCorner()
+ aGetShape()
+ aGetColor()
+ Rotate()
+ RightFlip()
+ LeftFlip()
+ Reverse()
+ aGetArea()
+ GetFlip()
+ aGetStatusReverse()
+ CenterPoint()

Module Shape :

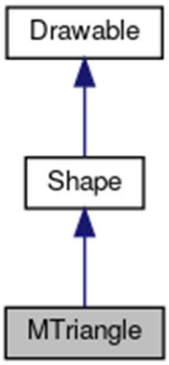
C_GTriangle :

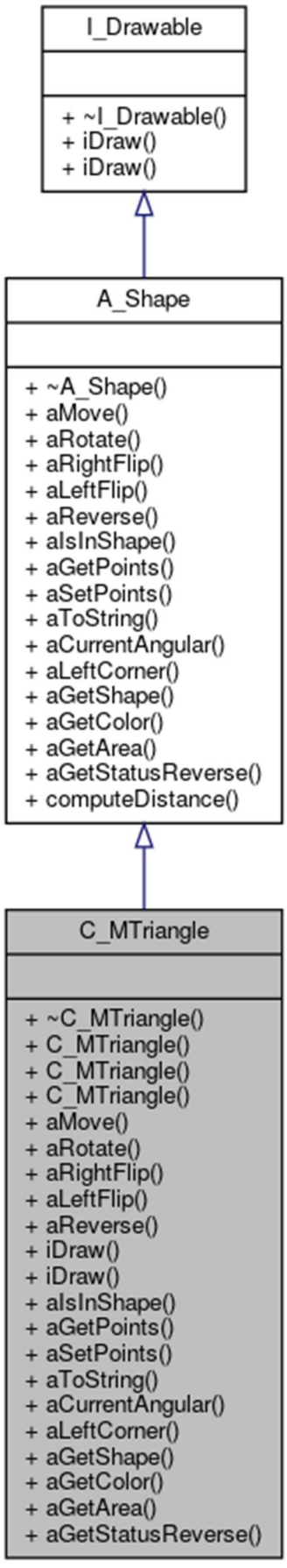


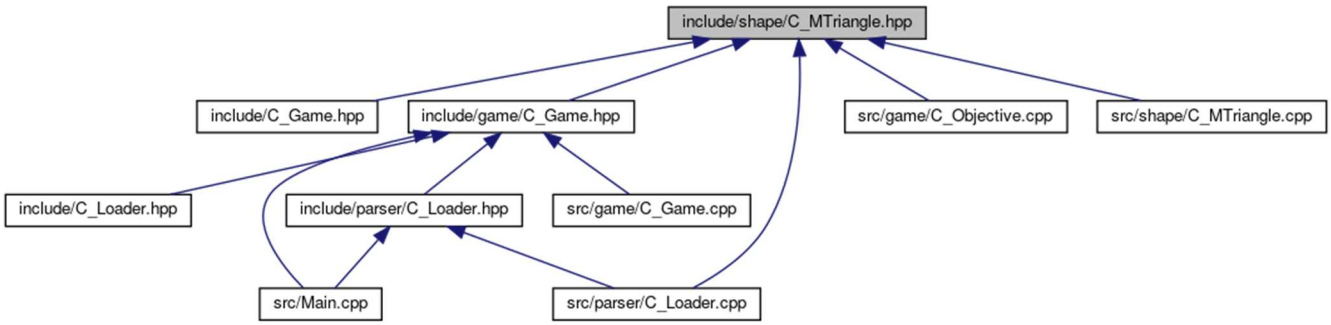



include/shape/C_GTriangle.hpp
include/C_Game.hpp
include/game/C_Game.hpp
src/game/C_Objective.cpp
src/shape/C_GTriangle.cpp
include/C_Loader.hpp
include/parser/C_Loader.hpp
src/game/C_Game.cpp
src/Main.cpp
src/parser/C_Loader.cpp

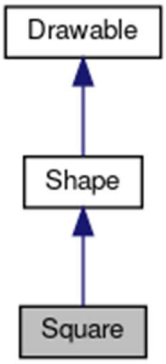
C_MTriangle :

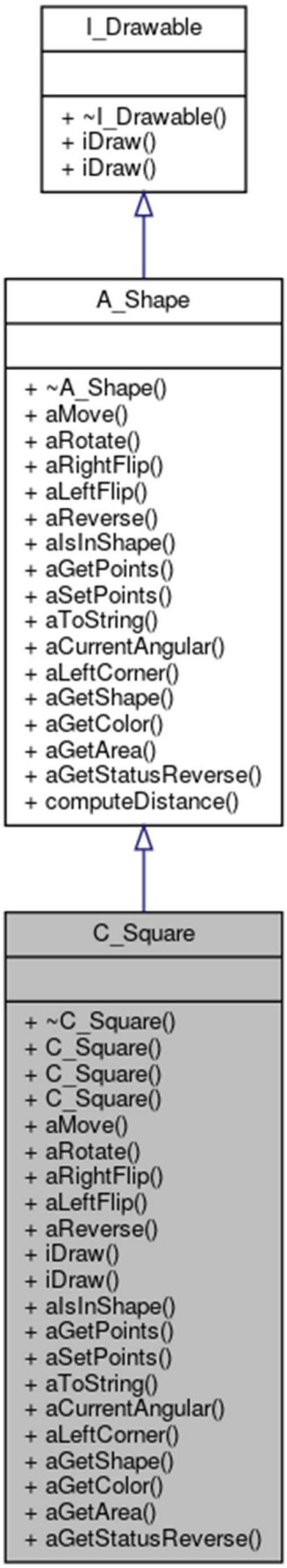

Drawable
Shape
MTriangle

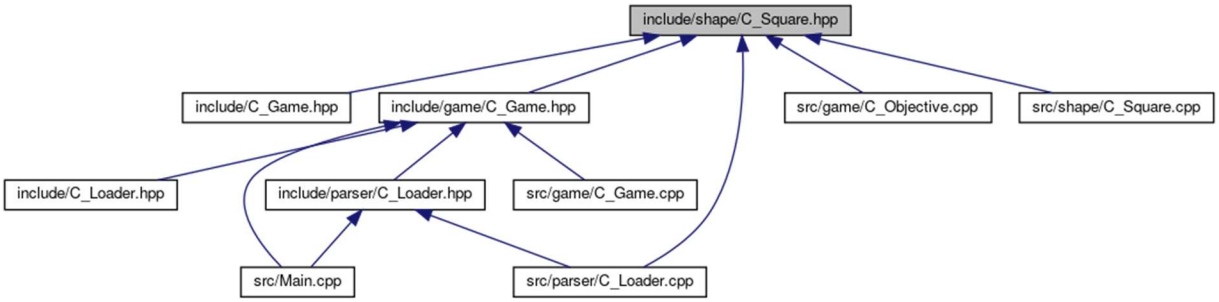

I_Drawable
+ ~I_Drawable()
+ iDraw()
+ iDraw()
A_Shape
+ ~A_Shape()
+ aMove()
+ aRotate()
+ aRightFlip()
+ aLeftFlip()
+ aReverse()
+ aIsInShape()
+ aGetPoints()
+ aSetPoints()
+ aToString()
+ aCurrentAngular()
+ aLeftCorner()
+ aGetShape()
+ aGetColor()
+ aGetArea()
+ aGetStatusReverse()
+ computeDistance()
C_MTriangle
+ ~C_MTriangle()
+ C_MTriangle()
+ C_MTriangle()
+ C_MTriangle()
+ aMove()
+ aRotate()
+ aRightFlip()
+ aLeftFlip()
+ aReverse()
+ iDraw()
+ iDraw()
+ aIsInShape()
+ aGetPoints()
+ aSetPoints()
+ aToString()
+ aCurrentAngular()
+ aLeftCorner()
+ aGetShape()
+ aGetColor()
+ aGetArea()
+ aGetStatusReverse()


include/shape/C_MTriangle.hpp
include/C_Game.hpp
include/game/C_Game.hpp
src/game/C_Objective.cpp
src/shape/C_MTriangle.cpp
include/C_Loader.hpp
include/parser/C_Loader.hpp
src/game/C_Game.cpp
src/Main.cpp
src/parser/C_Loader.cpp

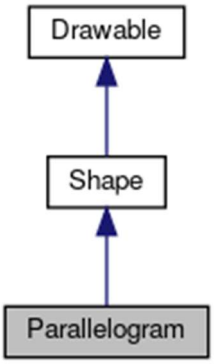
C_Square :

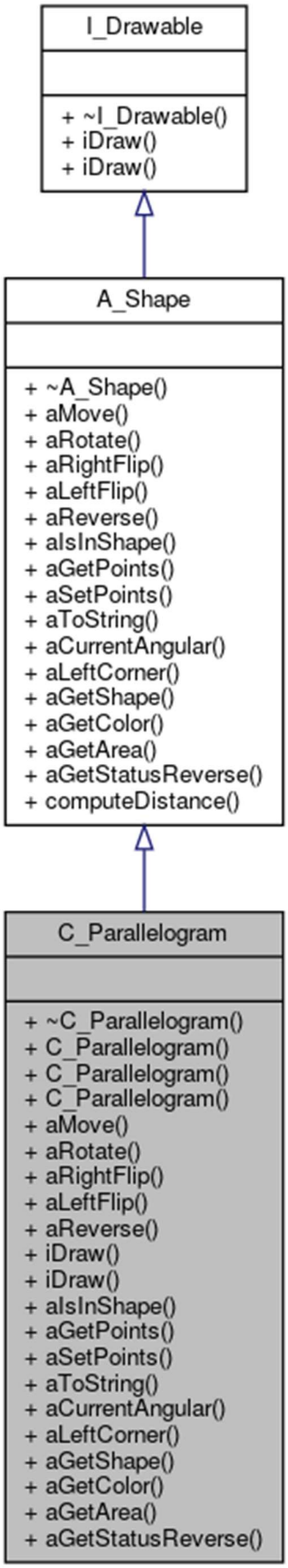

Drawable
Shape
Square

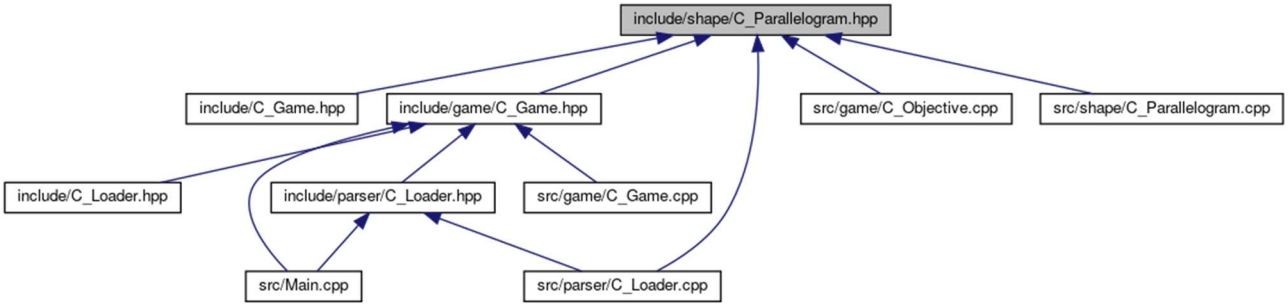

I_Drawable
+ ~I_Drawable()
+ iDraw()
+ iDraw()
A_Shape
+ ~A_Shape()
+ aMove()
+ aRotate()
+ aRightFlip()
+ aLeftFlip()
+ aReverse()
+ aIsInShape()
+ aGetPoints()
+ aSetPoints()
+ aToString()
+ aCurrentAngular()
+ aLeftCorner()
+ aGetShape()
+ aGetColor()
+ aGetArea()
+ aGetStatusReverse()
+ computeDistance()
C_Square
+ ~C_Square()
+ C_Square()
+ C_Square()
+ C_Square()
+ aMove()
+ aRotate()
+ aRightFlip()
+ aLeftFlip()
+ aReverse()
+ iDraw()
+ iDraw()
+ aIsInShape()
+ aGetPoints()
+ aSetPoints()
+ aToString()
+ aCurrentAngular()
+ aLeftCorner()
+ aGetShape()
+ aGetColor()
+ aGetArea()
+ aGetStatusReverse()


include/shape/C_Square.hpp
include/C_Game.hpp
include/game/C_Game.hpp
src/game/C_Objective.cpp
src/shape/C_Square.cpp
include/C_Loader.hpp
include/parser/C_Loader.hpp
src/game/C_Game.cpp
src/Main.cpp
src/parser/C_Loader.cpp

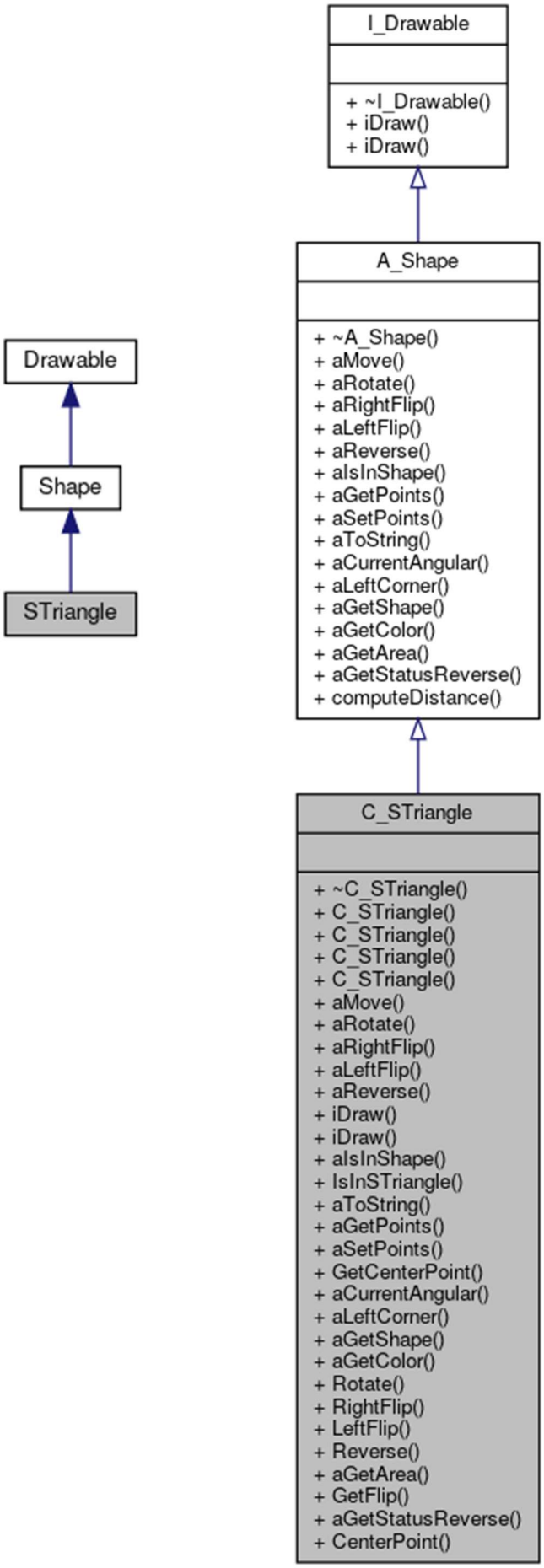
C_Parallelogram :

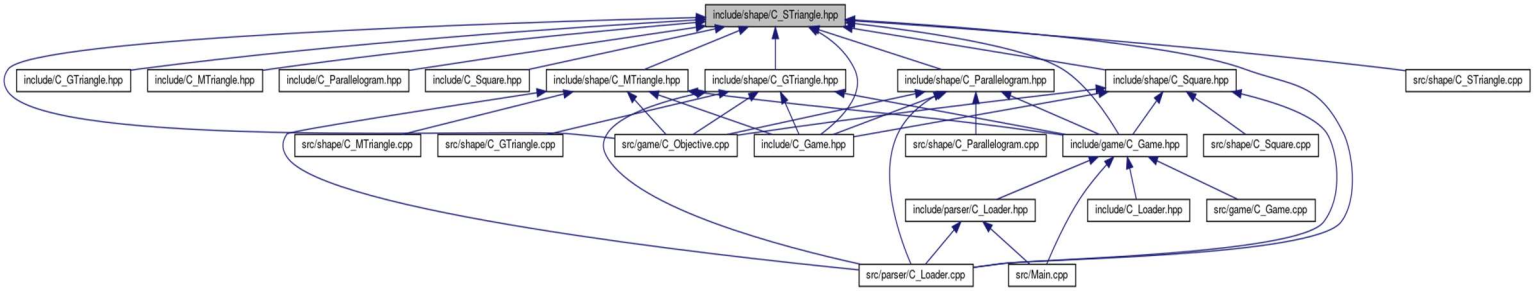





include/shape/C_Parallelogram.hpp
include/C_Game.hpp
include/game/C_Game.hpp
src/game/C_Objective.cpp
src/shape/C_Parallelogram.cpp
include/C_Loader.hpp
include/parser/C_Loader.hpp
src/game/C_Game.cpp
src/Main.cpp
src/parser/C_Loader.cpp

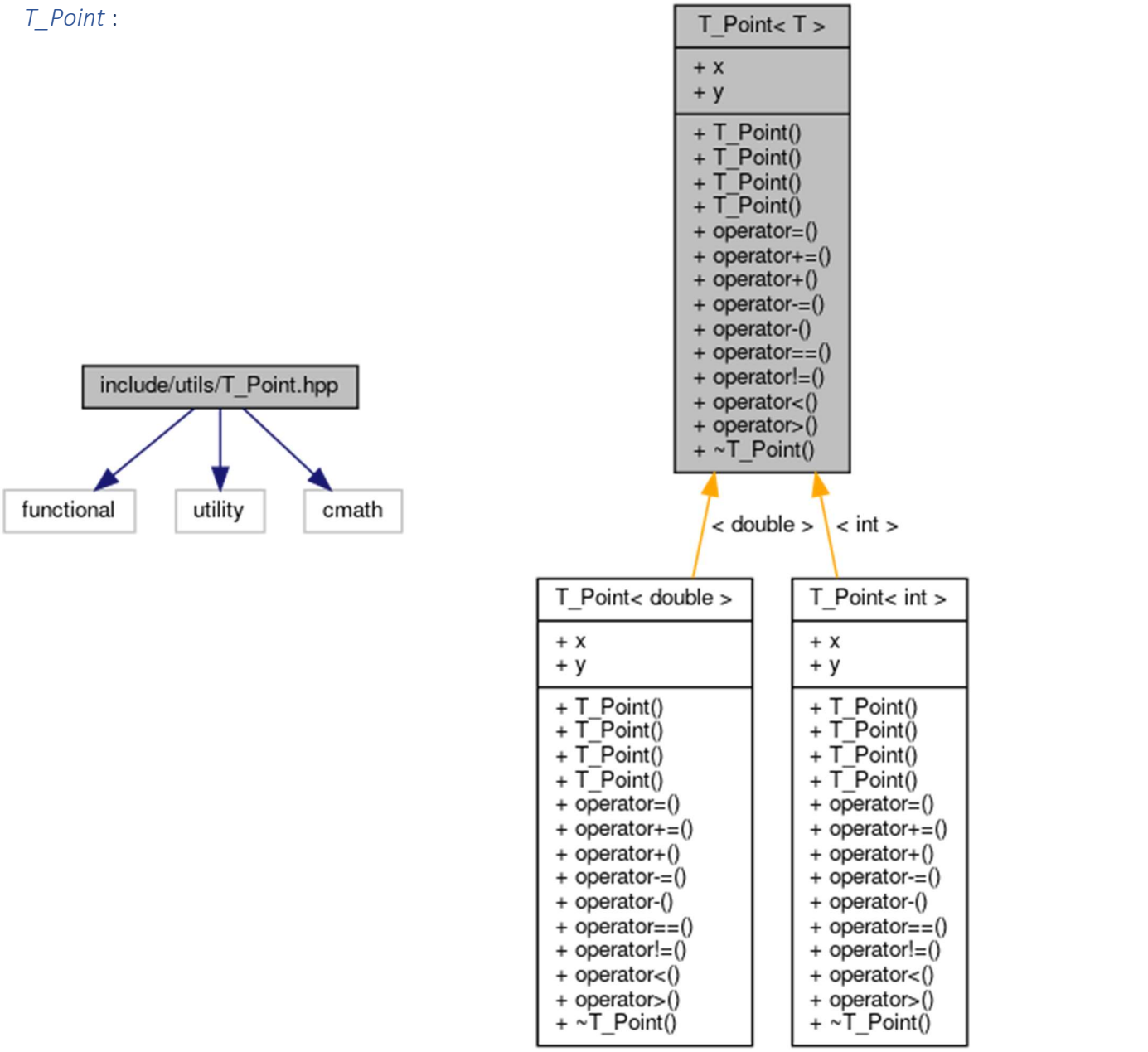
C_STriangle :

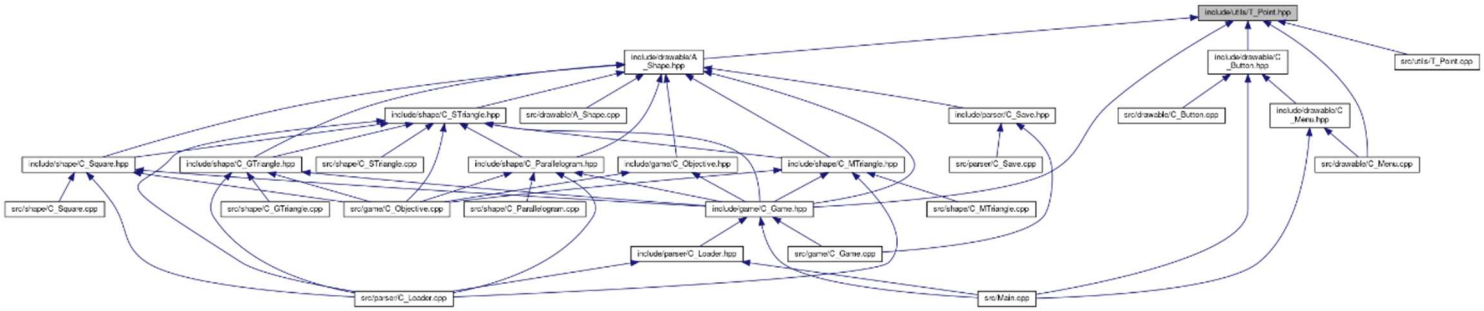

I_Drawable
+ ~I_Drawable()
+ iDraw()
+ iDraw()
A_Shape
+ ~A_Shape()
+ aMove()
+ aRotate()
+ aRightFlip()
+ aLeftFlip()
+ aReverse()
+ aIsInShape()
+ aGetPoints()
+ aSetPoints()
+ aToString()
+ aCurrentAngular()
+ aLeftCorner()
+ aGetShape()
+ aGetColor()
+ aGetArea()
+ aGetStatusReverse()
+ computeDistance()
C_STriangle
+ ~C_STriangle()
+ C_STriangle()
+ C_STriangle()
+ C_STriangle()
+ C_STriangle()
+ aMove()
+ aRotate()
+ aRightFlip()
+ aLeftFlip()
+ aReverse()
+ iDraw()
+ iDraw()
+ aIsInShape()
+ IsInSTriangle()
+ aToString()
+ aGetPoints()
+ aSetPoints()
+ GetCenterPoint()
+ aCurrentAngular()
+ aLeftCorner()
+ aGetShape()
+ aGetColor()
+ Rotate()
+ RightFlip()
+ LeftFlip()
+ Reverse()
+ aGetArea()
+ GetFlip()
+ aGetStatusReverse()
+ CenterPoint()
Drawable
Shape
STriangle


include/shape/C_STriangle.hpp
include/C_GTriangle.hpp
include/C_MTriangle.hpp
include/C_Parallelogram.hpp
include/C_Square.hpp
include/shape/C_MTriangle.hpp
include/shape/C_GTriangle.hpp
include/shape/C_Parallelogram.hpp
include/shape/C_Square.hpp
src/shape/C_STriangle.cpp
src/shape/C_MTriangle.cpp
src/shape/C_GTriangle.cpp
src/game/C_Objective.cpp
include/C_Game.hpp
src/shape/C_Parallelogram.cpp
include/game/C_Game.hpp
src/shape/C_Square.cpp
include/parser/C_Loader.hpp
include/C_Loader.hpp
src/game/C_Game.cpp
src/parser/C_Loader.cpp
src/Main.cpp

Module Utility :

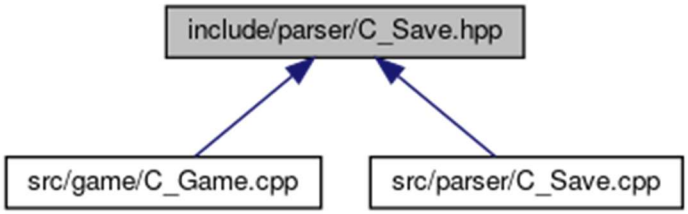
T_Point :



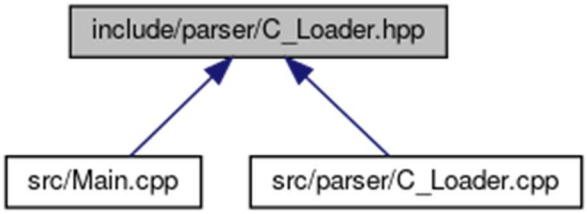


Module Parser :

C_Save :

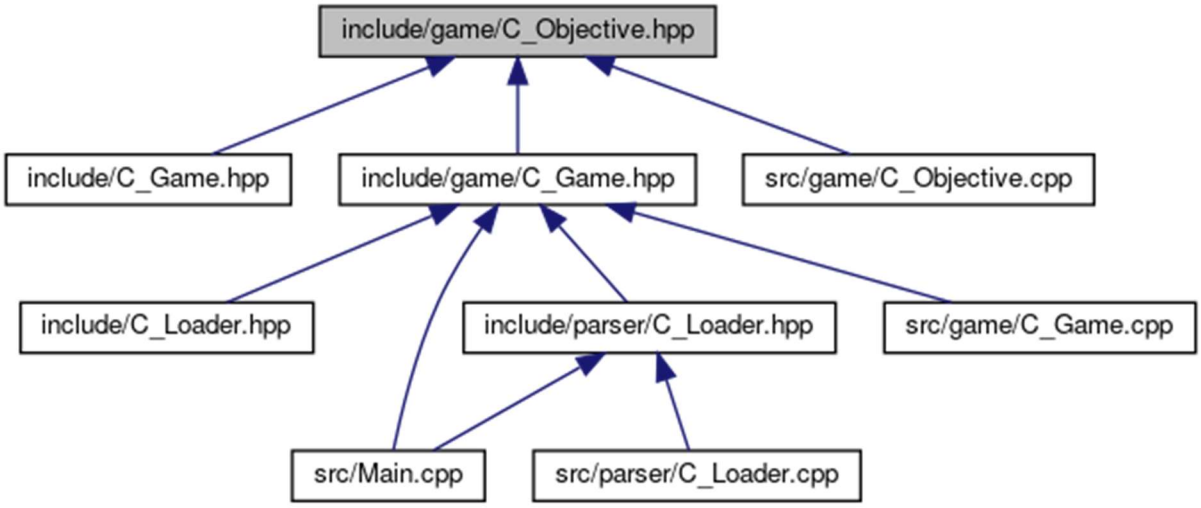


C_Load :

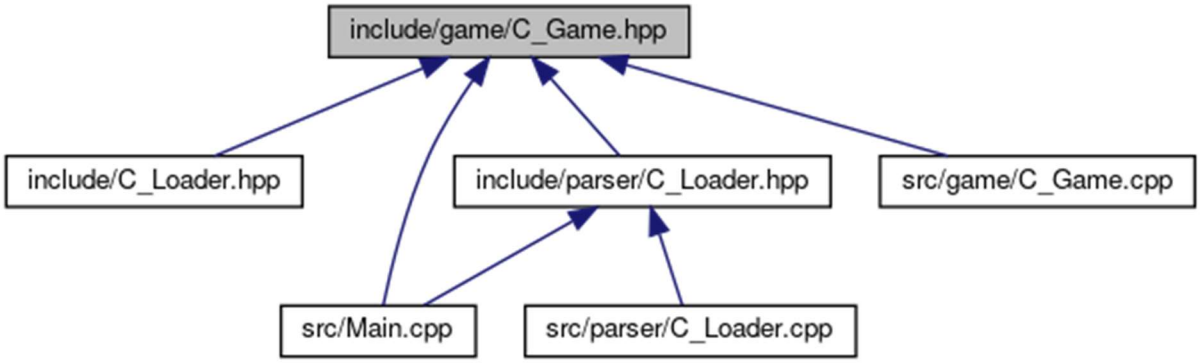


Module Game :

C_Objective :



C_Game :



Explication détaillée des modules :

Conventions de nommages :

Afin de maintenir une cohérence, une maintenabilité et une plus grande lisibilité du code je me suis imposé quelques conventions de nommage :

- Les classes d'objets commencent toutes par C_XXX, ex : C_STriangle
- Les interfaces commencent toutes par I_XXX, ex : I_Drawable
- Les classes abstraites commencent toutes par A_XXX, ex : A_Shape
- Les classes templates commences toutes par T_XXX, ex : T_Point
- Les méthodes de classes privées commencent toutes par __XXX , ex : __CenterPoint() dans C_STriangle
- Les méthodes d'interfaces commencent toutes par iXXX, ex : iDraw()
- Les méthodes de classes abstraites commencent toutes par aXXX, ex : aMove()
- Les champs de chaque classe commencent tous par mXXX, ex : mPoint dans C_STriangle

Grâce à ces conventions, il est simple d'identifier la provenance d'une méthode override, par exemple, ou de savoir directement à quoi sert exactement une classe.

Module Utils : Ce module est un module utilitaire qui ne contient qu'une seule classe Template T_Point. Cette classe décrit un point avec deux coordonnées x et y. Ces points serviront à construire et repérer les pièces à jouer ainsi que celles de l'objectif. Elle est utilisée dans le reste du projet et est construite comme une classe template puisque j'avais besoin de Point int et float (avec un comportement d'approximation sur les floats notamment pour l'opérateur ==). De plus j'avais également besoin de pouvoir créer une structure permettant de faire un hash sur un point.

Module Drawable :

Il est composé de 4 fichiers :

- Une interface fonctionnelle I_Drawable ne comportant qu'une seule méthode iDraw() permettant d'afficher un objet « drawable ».
- Une classe abstraite A_Shape qui correspond aux figures du Tangram
- Une classe C_Button permettant de gérer la navigation entre les différents affichages du menu grâce à un pointeur de fonction « callback ». Dans ce principe, quand on clique sur un bouton, on empile un appel de fonction à chaque clique.
- Une classe C_Menu qui stocke dans un std :: vector les boutons et leurs « callback ».

Module Game

Il est composé de deux classes :

I. Une classe Game qui régit tout le déroulement du jeu, ses commandes et leurs effets (collision, sticky), ainsi que l'actualisation de l'affichage du programme. Cette classe étant au cœur du programme et essentielle à la compréhension du reste du projet, notamment les méthodes collision et le stick. Dans un souci de clarté je vais expliquer ce que font ces dernières :

- Méthode collision :
- Elle vérifie à chaque déplacement d'une pièce si cette dernière va rencontrer une autre figure. Si c'est le cas elle annule le déplacement. Pour vérifier cela, je regarde si un des points de ma pièce qui est en mouvement est à l'intérieur d'une autre figure (méthode IsInShape() prenant un Point en paramètre).

Cette fonction implique une conséquence, c'est que les pièces ne peuvent pas se superposer, mais des erreurs d'approximations dues aux float peuvent autoriser un léger chevauchement d'1 ou 2 pixels. Cela aura des conséquences pour les explications suivantes.

- Méthode stick :

1. Quand une pièce vient d'être déplacée, tournée ou flipée et qu'on relâche le bouton de la souris :
2. On crée un set de points avec ceux de la pièce en question, et une map vide
3. Puis pour chaque point dans ce set :
4. On parcourt chaque point du set objectif (en champ de classe prérempli)
5. On calcule la distance euclidienne entre les deux points
6. Si la distance est inférieure à 12 (c'est-à-dire que les points sont assez proches)
7. Et Si notre map contient déjà ce point, on vérifie que la distance stockée dedans est supérieur pour mettre à jour le point déjà stockée dedans.
8. Si la distance est inférieure pour ce même point on ne fait rien
9. Si notre map ne contient pas le point alors on l'ajoute <key : point_objectif, value : <point_shape, distance> >
10. La double boucle terminée on vérifie que qu'on a autant de points dans notre map ou plus que dans le set crée au début, sinon on sort de la fonction.
11. Si oui alors on crée un vector « save » dans lequel on copie les points du set
12. Puis pour chaque point de la map on prend les N points les plus proches (càd ceux dont la distance par rapport au point_shap est inférieur à 12, sachant que l'on veut prendre parmi tous ceux-là, les plus proches du point_shape). N correspond au nombre de points du shape.
13. Ensuite on set une version temporaire de notre shape aux points de la map, on vérifie que son aire n'a pas bougé plus de 10%, si oui on annule le stick et on sort de la fonction.
14. Sinon on valide le stick, on met à jour les points de la pièce et on sort de la fonction.

Dans l'ensemble cette fonction semble lourde mais elle n'est appelée que quand on relâche la pièce et de plus il y a pas mal de conditions qui permettent d'éviter des calculs inutiles et les getters contiennent déjà les points précalculés et sont conçus pour ne s'actualiser que lorsqu'une modification est faite. De plus j'utilise std :: move pour la copie des points ce qui est optimal.

Il est important de comprendre ce que fait cette fonction pour la suite du rapport, je rajouterai d'ailleurs à cela quelques précisions sur les conséquences de cette fonction :

- Une pièce est stickée aux points les plus près d'elle, cela signifie qu'elle peut se sticker sur des points, qui initialement « ne sont pas prévus » pour cette pièce. Dans le sens où, comme expliqué avec la collision, des pièces peuvent légèrement se chevaucher entre elles à 1 ou 2 pixels près.
- Si une pièce est stickée, cela signifie qu'elle est potentiellement bien placée, du moins, elle correspond à des points de l'objectif, et ainsi elle assure une correspondance certaine entre l'objectif et les pièces en jeu.

II. Une classe Objective qui régit tout ce qui concerne la condition de victoire du jeu et calcule l'avancement du joueur dans le puzzle

La classe Objective m'a posé quelques problèmes. Au départ, je souhaitais vérifier que les points de mes pièces objectifs étaient une permutation des points de mes pièces en jeu. J'ai eu ensuite recours à une approche plus directe : comparer la liste des points de mes pièces à jouer et la liste des points des pièces objectifs, supprimer les doublons dans chaque liste et vérifier que chaque point d'une de ces deux listes se trouve dans l'autre. La validité de la victoire dépend en outre de ces deux axiomes :

- le stick, qui sert à ajuster les points d'une pièce en jeu aux points objectifs les plus proches, si, pour chaque point de la pièce à jouer, il y a un point objectif proche. Je garantis ainsi que ma pièce stickée est bien sur une partie de l'objectif.

- la collision interdit que les pièces se superposent. De ce fait, je suis sûr d'avoir mes pièces en jeu sur tout l'objectif pour la condition de victoire.

Il est important de comprendre ici que ces trois conditions permettent d'éviter de tricher en résolvant le puzzle sans avoir placé toutes les pièces. Une autre condition reposant sur un autre principe décrit plus loin permet d'autoriser les solutions multiples.

Si ces trois conditions sont respectées, il ne me reste qu'à compter pour chaque point objectif le nombre de points en jeu. Si ces nombres sont égaux, alors le puzzle est résolu. Il est important de comprendre ici que ces trois conditions permettent d'éviter de tricher en résolvant le puzzle sans avoir placé toutes les pièces. Un autre axiome reposant sur un autre principe permet d'autoriser les solutions multiples par puzzle. (Cf. ci-dessous, design pattern composite)

Module Shape : Il existe 5 classes STriangle, MTriangle, GTriangle, Square et Parallelogram qui implémentent toutes la classe abstraite A_Shape (design pattern factory).

La classe STriangle (design pattern composite) permet à elle seule de créer chaque pièce. Si le puzzle est résolu d'une façon légitime, quel que soit la configuration des pièces, alors les points des pièces sont au même endroit et autorisent donc les solutions multiples. On peut considérer le puzzle comme un simple assemblage de STriangle, ce qui facilite grandement l'usage de toutes les fonctions de manipulation de pièces (« rotate », « move », « flip », « reverse »). Les pièces ont des comportements assez similaires. Le Parallelogram fait exception : la fonction reverse ne retourne pas cette pièce mais crée son miroir par symétrie.

Module Parser : Ce module contient une classe Load et une classe Save, la classe load parse les fichiers contenu dans ./extern/board/pageX pour charger un puzzle et le jouer. La classe Save quant à elle sauvegarde un puzzle créée par le joueur dans ./extern/board/pageX, le numéro de la page dépend du nombre de fichiers par page (max 12) si une page est pleine elle créera automatiquement une nouvelle page pour sauvegarder les nouveaux puzzle. Le nom donné au fichier de sauvegarde est aléatoire, je n'ai pas eu le temps de créer une fonction pour supprimer les puzzles via l'interface ni de renommage non plus (il faut donc le faire manuellement). La classe Save utilise des appels systèmes pour créer les pages (mkdir, etc..).

Conclusion :

Le projet a été bien modularisé, chaque classe maintient les principes de l'encapsulation et certaines fonctionnalités annexes ont été ajoutées dans un souci de jouabilité ou de peaufinage. Dans l'idéal il faudrait que je rajoute les fonctionnalités pour avoir un aperçu du puzzle qui va être chargé, un moyen d'en supprimer via l'interface graphique, ainsi que de les renommer ou de les nommer à la sauvegarde.