GOSSET Séverin
BIGUENET Denis
Université Gustave Eiffel

# Rapport
# Projet Tangram

# Table des matières

# Introduction

Le but de ce projet est de créer un jeu de Tangram classique, c'est-à-dire reproduire une figure avec des pièces de différentes formes. On souhaite également la possibilité de créer un niveau grâce à un éditeur basique intégré au jeu, ainsi que de le sauvegarder pour le rendre jouable.
Le jeu doit-être avant tout intuitif et ergonomique à l'utilisation. Pour cela, nous avons mis en œuvre plusieurs solutions.
Tout d'abord, nous avons ajouté du magnétisme entre les pièces, afin que les pièces s'attirent entre elles, et soient attirées par la figure à reproduire. Cette solution évite de devoir placer au pixel près les pièces : le magnétisme le fait tout seul lorsque la pièce est placée à proximité de l'emplacement voulu.
Une tolérance à l'erreur a aussi été mise en place afin que l'utilisateur gagne même si une ou plusieurs pièces ne sont pas parfaitement bien placées. Enfin, l'interface doit-être dans l'idéal *responsive*, et s'adapter à la taille de l'écran.

# Manuel utilisateur

## Menu

Le jeu se lance sur un menu principal qui offre plusieurs actions possibles, telles que jouer un niveau, créer un niveau, ou quitter.

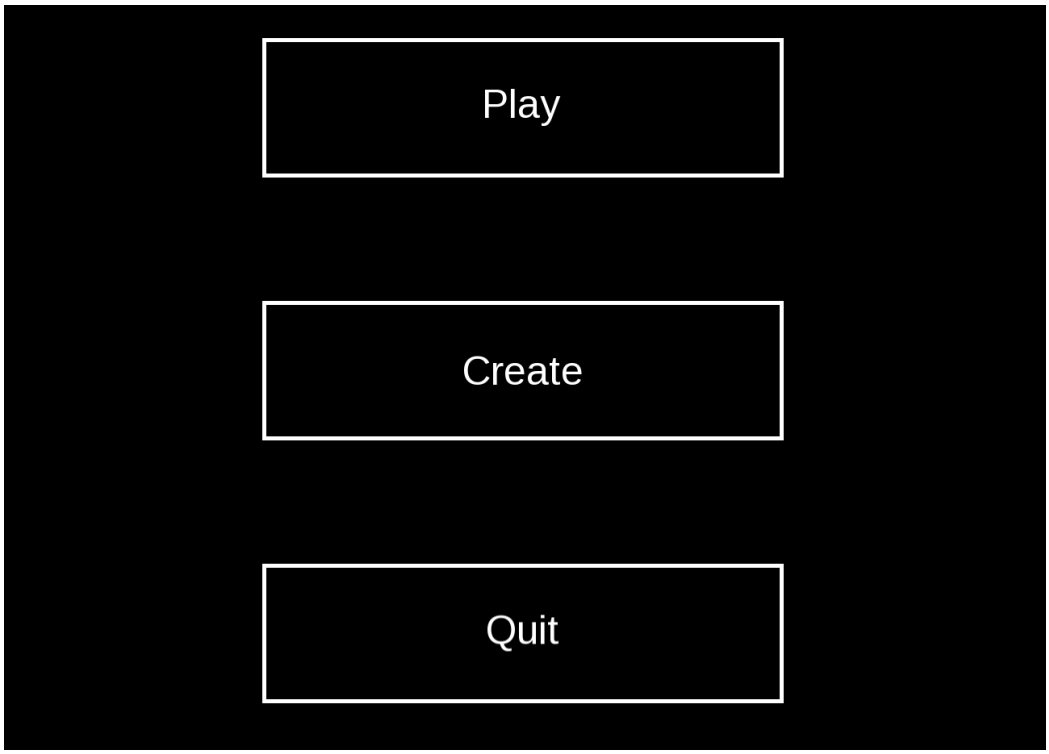



*Figure 1 : menu principal du jeu*

## Sélectionner un niveau

Pour jouer un niveau, cliquer sur « Play », un écran de sélection apparait alors, avec tous les niveaux possibles. Chaque niveau est représenté par un rectangle, ainsi que la figure à reproduire à l'intérieur. Pour naviguer entre les différents niveaux, cliquer sur les boutons « << « et « >> », respectivement à gauche et à droite de l'écran. Pour jouer un niveau, cliquer sur le rectangle correspondant.

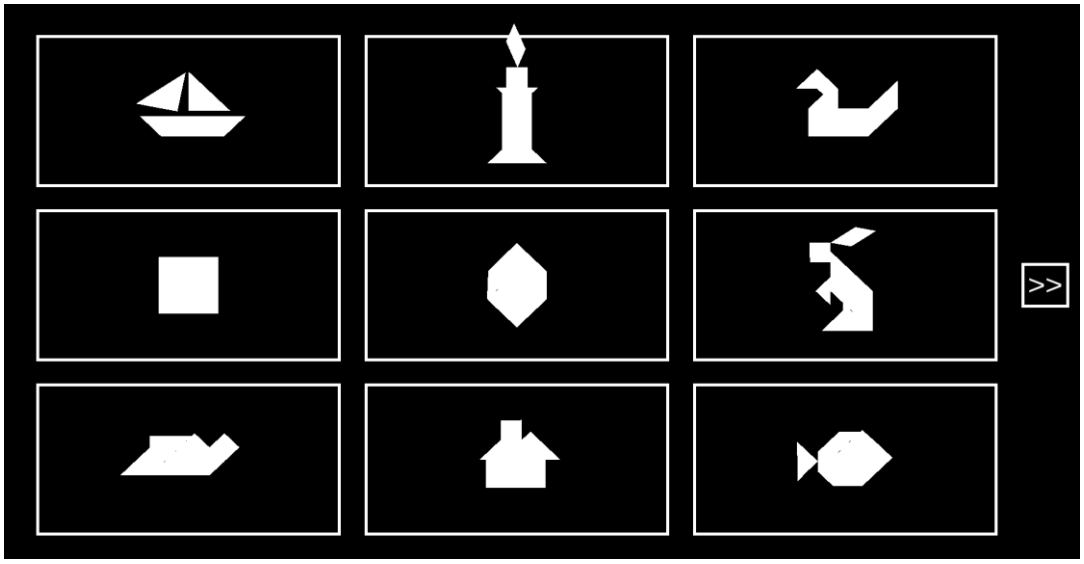



*Figure 2 : menu de sélection de niveau*

## Jouer un niveau

Une fois que le joueur a sélectionné le niveau de son choix. Il a accès à une interface composée de plusieurs boutons :

- « Load », afin de charger un autre niveau
- « Menu », pour revenir au menu principal
- « Quit », afin de quitter le jeu

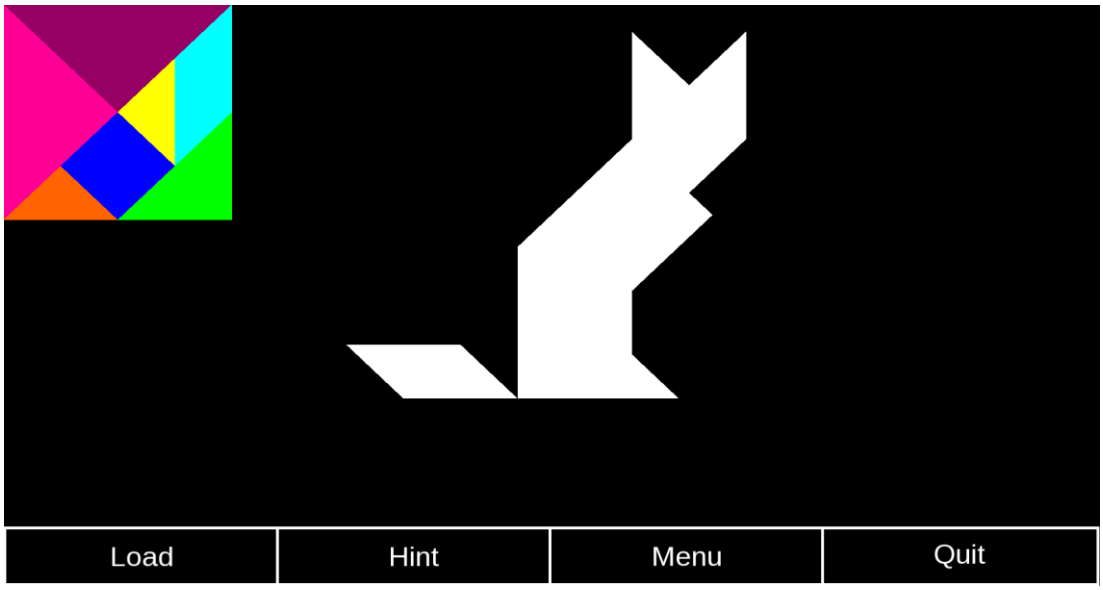



*Figure 3 : écran de jeu au cours d'une partie*

Les pièces utilisables (en couleur) sont placées en haut à gauche en début de partie. Au milieu en blanc, la figure à reproduire. Le but du jeu est de recouvrir toute la surface de la figure à reproduire grâce aux pièces.

## Liste des commandes lors d'une partie

- Sélectionner une pièce : maintenir clic gauche en visant la pièce, puis bouger la pièce en déplaçant la souris
- Faire tourner une pièce : molette de la souris

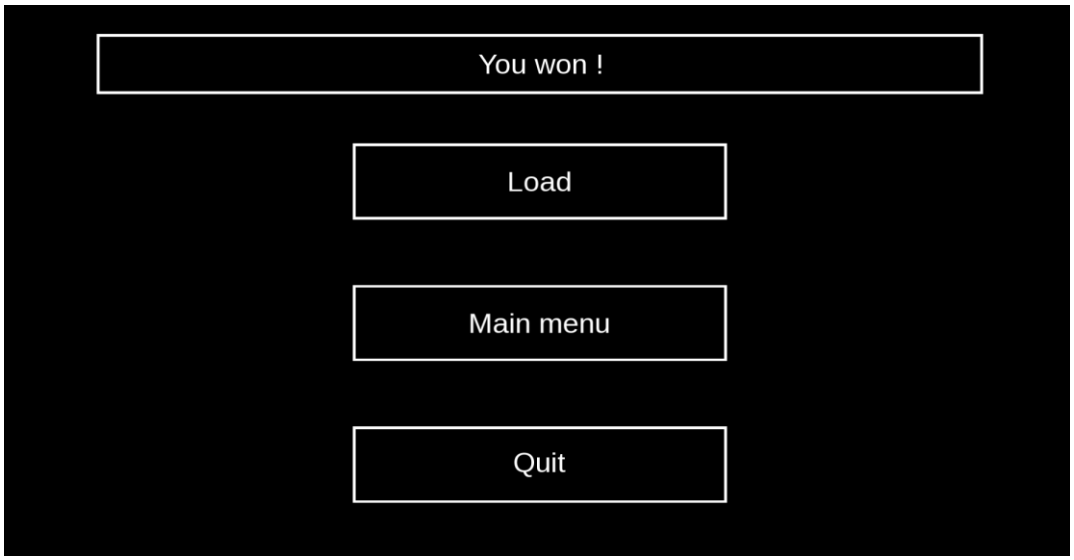



*Figure 4 : écran de victoire*

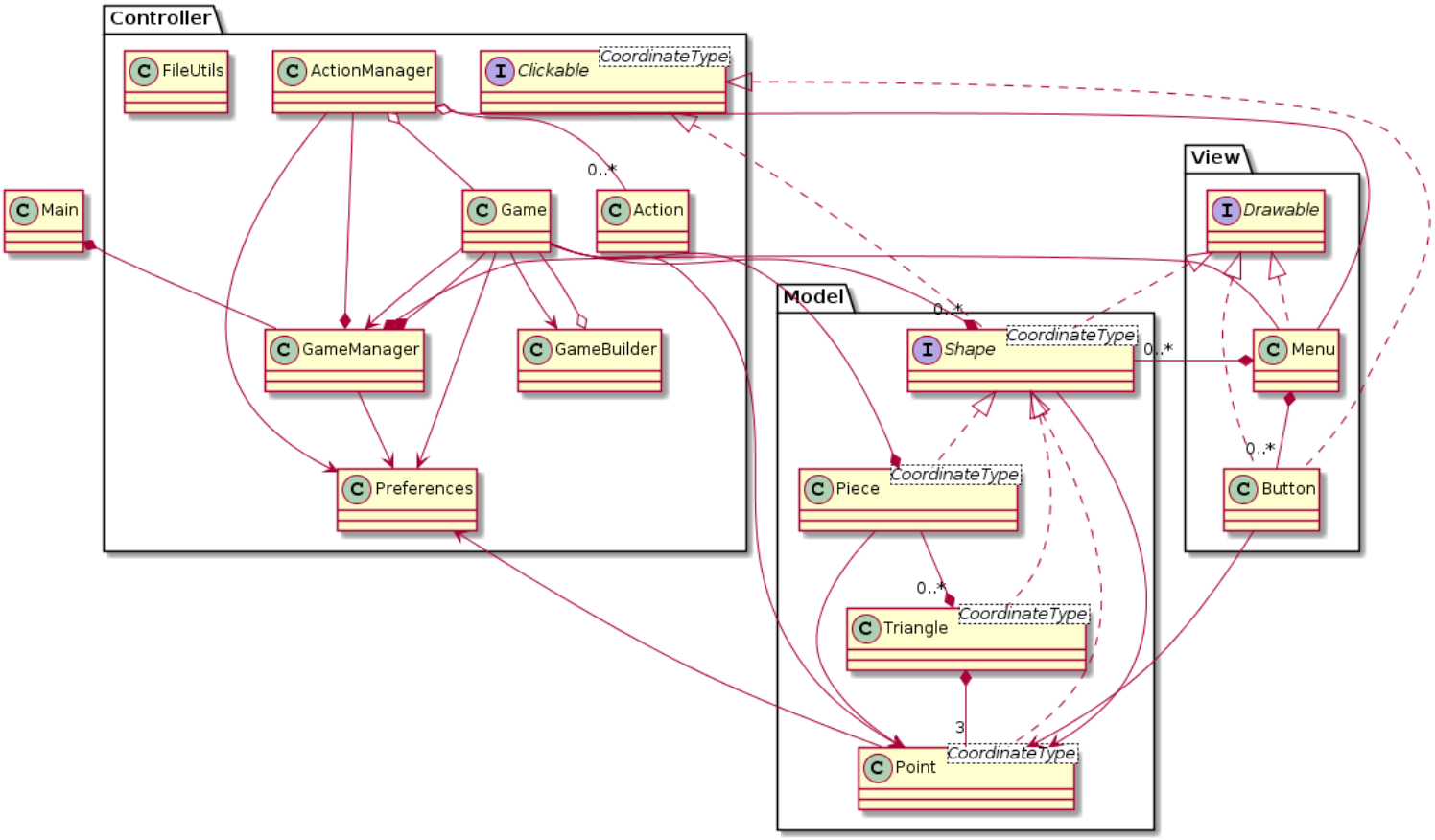
## Créer un niveau

Pour créer un niveau, cliquer sur le bouton « Create level » à partir du menu principal du jeu. Le joueur dispose des sept pièces du Tangram, qu'il peut bouger, faire tourner, afin d'obtenir la forme voulue.
Pour sauvegarder le niveau, cliquer sur le bouton « Save level».


*Figure 5 : écran de création d'un niveau*

Il est possible de choisir le nom du niveau, qui sera alors le nom du fichier sauvegardé.

# Schéma UML




L'UML complet avec les méthodes de classes est disponible dans les sources.

# Rôle des classes

## Point

La classe la plus basique du projet est la classe Point, qui représente un point dans un espace 2D, un point possède donc deux attributs x et y pour ses coordonnées. Il peut être judicieux de faire de cette classe un template, nous permettant de choisir le type de données que l'on souhaite (int, float, double…).

## Triangle & Pièce

Un triangle est composé de trois points distincts (relation de composition), ainsi que d'un angle et de trois points bruts, qui correspondent aux trois points précédents sans rotations. En effet, les rotations successives peuvent déformer le triangle ou donner une nouvelle position approximative, il faut alors garder en mémoire la position brute du triangle afin de n'effectuer qu'une seule rotation de thêta degrés dessus, plutôt qu'une succession de rotations.
Chaque pièce est composée d'un certain nombre de triangles, ce qui est très intéressant pour la condition de victoire : chaque pièce ainsi que la forme à reproduire, peut être vue comme un ensemble de triangles basiques. La condition de victoire sera détaillée plus tard dans ce document.

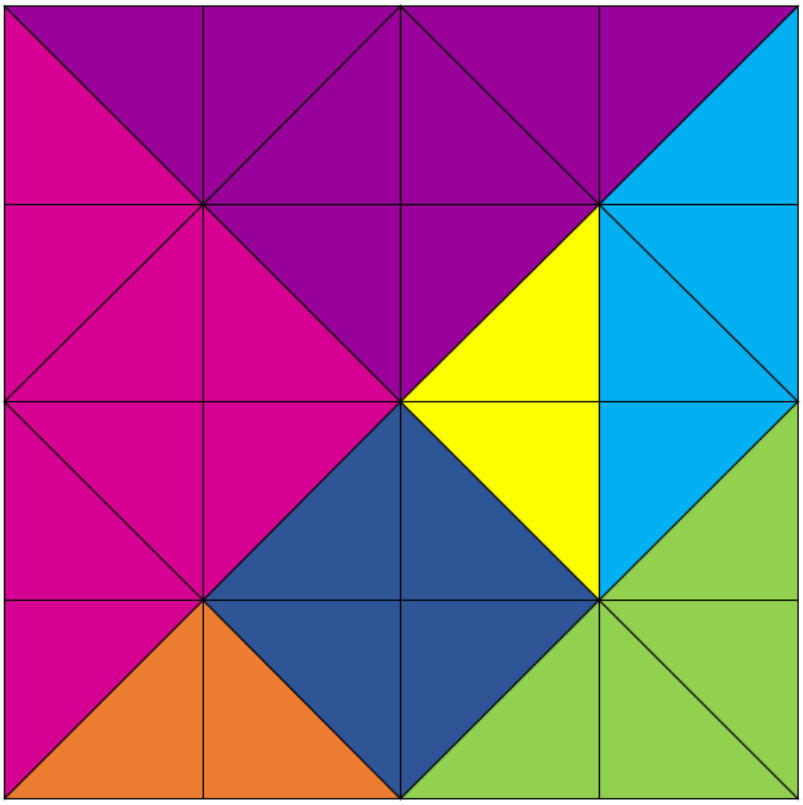


*Figure 6 : chaque pièce est composée de plusieurs triangles de base*

Il y a plusieurs avantages au design pattern de composition :

- Inutile de créer une classe par type de pièce, il n'y aura que deux classes : Triangle et Piece, peu importe le nombre de formes.

- Une seule implémentation pour chaque méthode peu importe le nombre de forme de pièce possible, toute pièce est traitée comme un ensemble de triangles.

### Shape

Shape permet de définir une interface commune entre les pièces et les triangles, qui se traitent de la même manière. Cette interface implémente elle-même l'interface Clickable, car une pièce doit être sélectionnable par l'utilisateur, ainsi que l'interface Drawable, car on peut dessiner une pièce.

### Button

Un bouton est un carré ou un rectangle définit par un point haut-gauche et un point bas-droit, un texte et une fonction à appliquer lorsque l'on clique dessus (il s'agit donc d'une lambda). Etant donné que l'on peut interagir avec un bouton, cette classe implémente donc l'interface Clickable mais aussi Drawable.

### Clickable

Interface commune pour tous les objets alors lesquels le joueur peut interagir, en l'occurrence, les pièces et les boutons.

### Action

Une action est représentée par une lambda, qui est appelée lorsqu'un certains événement a lieu. On a donc fait de cette classe un foncteur : on appelle la fonction dans la surcharge de l'opérateur ().

### ActionManager

ActionManager sert à gérer les événements et les actions associés à l'utilisateur, tels que les événements de clavier et de souris. Cette classe est composée d'une map qui associe un événement SFML à une Action. Au lancement du programme, la map est initialisée avec toutes actions réalisables par l'utilisateur. La classe GameManager contient aussi des pointeurs vers Game et Menu, car toutes les actions ont pour but de modifier l'état de l'interface ou de la partie.

### Menu

Un menu est composé de boutons avec lesquels peut interagir le joueur, mais aussi de pièces utilisées pour décorer l'interface, ces pièces sont non utilisables par le joueur. Un menu peut être dessiné et implémente donc l'interface Drawable.

### Game

La classe Game sert à définir une partie, c'est-à-dire :

- Un ensemble de pièces
- Un objectif : une forme à reproduire avec les pièces à disposition
- Une pièce sélectionnée parmi les pièces à disposition
- Un état (en cours ou terminée)

Cette classe sert à donner un comportement à gérer le déroulement d'une partie : vérifier la condition de victoire, effectuer les transformations sur les pièces (comme les rotations ou les translations). Elle est directement reliée à la classe ActionManager, qui appelle des méthodes de Game selon les événements.
Cette classe implémente aussi l'interface Drawable car elle contient tout ce qui peut être et doit être dessiné à un moment donné de la partie.

## GameManager

Le GameManager est composé d'une Game, d'un Menu, d'un ActionManager, ainsi que d'une fenêtre SFML. Cette classe permet de gérer le jeu dans sa globalité : modifier le Menu et la Game selon les boutons utilisés, dessiner ce qui doit l'être sur la fenêtre, ou même libérer l'espace mémoire quand une partie vient d'être terminée ou abandonnée.

## Drawable

L'interface Drawable permet de définir une interface commune pour tout ce qui peut être dessiné, notamment les pièces, les boutons, et le menu.

## Preferences

Cette classe est un singleton qui sert à stocker les différentes globales, comme le magnétisme, ou la tolérance pour la condition de victoire. Il serait possible d'ajouter un menu Options pour modifier ces globales, par exemple, enlever le magnétisme.

## Main

Cette classe crée une partie et n'appelle que des méthodes de celle-ci (condition de victoire, affichage…), afin de déléguer un maximum de tâches et d'écrire un minimum de code dans ce Main.

## FileUtils

Cette classe statique sert à gérer les écritures et lectures de fichiers (charger ou sauvegarder une partie).
Dans notre Tangram, on souhaite pouvoir créer un niveau à partir des sept pièces de base, ainsi que charger un niveau déjà existant à partir d'un fichier.
Etant donné que cette classe ne contient pas d'attribut et que l'on ne souhaite pas l'instancier, il est alors logique d'en faire une classe statique, elle ne contiendra alors que des méthodes statiques que l'on peut appeler depuis les autres classes.

# Principales fonctions

## Magnétisme

Un problème est apparu lorsque l'on essayait de reproduire la figure. En effet, il est en effet très difficile de placer une pièce au pixel près, or les coordonnées des pièces placées sont très importantes dans la vérification des conditions de victoires.
Pour résoudre ce problème il y a plusieurs possibilités :

1. Mettre en place une tolérance dans la vérification des conditions de victoire, afin de pouvoir gagner, et ce même si les pièces ne sont pas exactement bien placées.
2. Ajouter du magnétisme entre les pièces, c'est-à-dire faire en sorte qu'une pièce soit attirée par une autre pièce lorsqu'on la relâche, mais aussi attirée par la figure à reproduire. Une pièce est attirée par les sommets d'une autre forme si elle est suffisamment à proximité.

```
void Game::magnetize() {
	// Distance entre deux points de la pièce relâchée et d'une autre pièce
	double dist;
	// Plus petite distance entre deux points de la pièce relâchée et d'une autre
	pièce
	double minDist = DBL_MAX;

	std::vector<Point<double>> points(2);

	std::vector<Point<double>> minPoints(2);
	// Translation à effectuer sur la pièce relâchée
	Point<double> translation;
	// On récupère l'instance de préférences pour la distance de magnétisme
	Preferences& pref = Preferences::getInstance();
	// On récupère la plus petite distance entre la pièce relâchée et une autre
	pièce, on parcourt le vecteur de pièces
	for (auto& piece : pieces) {
		if (piece != selected) {
			// On calcule la distance entre les deux pièces, les deux points
			sont stockés dans le vecteur points
			dist = selected->distance(piece.get(), points);
			// Si cette distance est plus petite que minDist, alors minDist
			devient dist et minPoints devient points
			if (dist < minDist) {
				minDist = dist;
				minPoints = points;
			}
		}
	}
	// On récupère la plus petite distance entre la pièce relâchée et la figure à
	reproduire
	if (goal != nullptr) {
		dist = selected->distance(goal.get(), points);
		if (dist < minDist) {
			minDist = dist;
			minPoints = points;
		}
	}
	// On n'effectue le magnétisme que si la distance est plus petite que la
	distance de magnétisme récupérée dans les préférences
	if (minDist < pref.getMagnetism()) {
		// On effectue une translation sur la pièce relachée
		translation = minPoints[1] - minPoints[0];
		selected->translate(translation);
	}
}
```

*Figure 7 : code de la méthode de magnétisme*

Le principe du magnétisme est assez simple : on vérifie s'il existe un sommet d'une pièce ou de la forme à reproduire à proximité de la pièce que l'on vient de relâcher. On calcule alors la distance entre les deux pièces, dans notre cas, on peut la définir comme la plus petite distance entre deux sommets de chaque pièce. Si cette distance est inférieure à la distance de magnétisme donnée, alors la pièce relâchée est attirée par l'autre pièce. Dans ce cas on translate la pièce relâchée par le vecteur obtenu à partir des deux points.

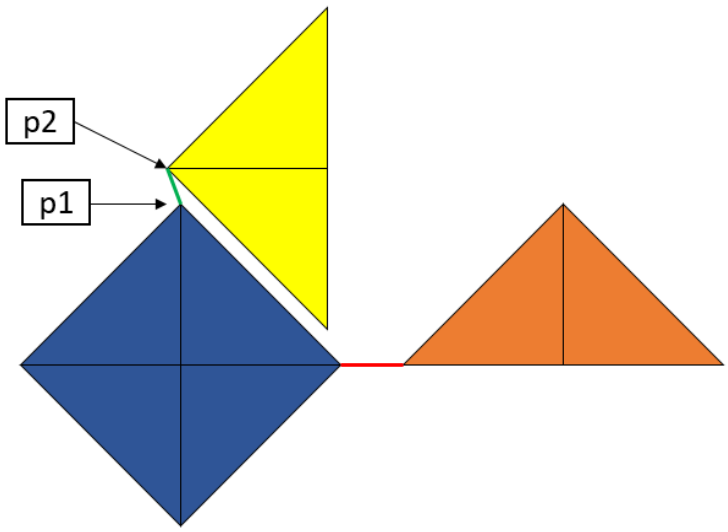



*Figure 8*

Dans l'exemple si dessus, la pièce bleue vient d'être relâchée. Le segment vert représente la distance entre la pièce bleue et la pièce jaune. Si cette distance est inférieure à la distance de magnétisme donnée dans Preferences.hpp, alors la pièce bleue est attirée par la pièce jaune. On translate alors la pièce bleue par le vecteur p2 – p1. Cependant, la pièce bleue peut être à proximité d'autres pièces comme le triangle orange. Dans ce cas, la pièce bleue est attirée par la pièce la plus proche, le triangle jaune en l'occurrence, puisque le segment rouge est plus long que le segment vert.

## La condition de victoire

Tout d'abord, il paraît évident de ne vérifier cette condition de victoire uniquement lorsque l'utilisateur pose une pièce, puisque l'état du jeu ne devrait pas changer autrement.

La condition de victoire dans notre jeu est assez simple, comme déjà expliqué précédemment dans le document, notre figure à reproduire est en fait une grille composée de triangles élémentaires.

```
void Game::validateShape() {
	// Lorsque l'on crée un niveau, la condition de victoire n'est pas vérifiée
	if (goal == nullptr) {
		return;
	}
	// points issus de la figure à reproduire
	std::vector<Point<double>> goalPoints = goal->getPoints();
	// points de toutes les pièces utilisables par le joueur
	std::vector<Point<double>> piecesPoints;
	// points d'une pièce
	std::vector<Point<double>> piecePoints;

	// Pour toutes les pièces utilisables :
	for (auto& piece : pieces) {
		// On récupère les points de la pièce courante
		piecePoints = piece->getPoints();
		// On l'ajoute dans l'ensemble des points de toutes les pièces
		piecesPoints.insert(piecesPoints.end(), piecePoints.begin(),
		piecePoints.end());
	}
	// On obtient alors un vecteur contenant tous les points de toutes les pièces
	utilisables par le joueur (picesPoints)
	// Ainsi qu'un vecteur contenant les points de la forme à reproduire
	(goalPoints)

	// On vérifie si goalPoints est une permutation de piecePoints
	// Si c'est le cas, on change l'état de la partie pour indiquer que le joueur a
	gagné
	if (std::is_permutation(goalPoints.begin(), goalPoints.end(),
	piecesPoints.begin(), piecesPoints.end())) {
		gameState = true;
	}
}
```

*Figure 9 : code de la condition de victoire*

La première chose à vérifier est de tester si la figure à reproduire est non nulle : dans le cas où l'on crée un niveau, il n'y a pas de condition de victoire.
Ensuite, on récupère les points de la figure à reproduire dans un vecteur, ainsi que les points des pièces utilisables dans un autre vecteur.
Dans le cas où le joueur a gagné, les deux ensembles ont alors les mêmes points, potentiellement dans un ordre différent. Pour vérifier cela, on teste si l'un des deux vecteurs est une permutation de l'autre. La fonction std::is_permutation est de complexité quadratique, cependant, étant donné que cette méthode est appelée uniquement lorsque l'on relâche une pièce, et qu'il y a relativement peu de points, le test se fait instantanément.
Malgré tout, il faut faire attention au cas des coordonnées de type *double* et *float*. En effet, il faut surcharger l'opérateur == pour ces types, afin que deux nombres très proches soient considérés comme égaux. On a donc mi une globale de tolérance à l'erreur dans le singleton Preferences.

## Création de niveau

```
void Game::save() {
	std::vector<Point<double>> points;
	// On récupère les points de chaque pièce dans un vecteur
	for_each(pieces.cbegin(), pieces.cend(),
[&points](std::shared_ptr<Shape<double>> s) {
		std::vector<Point<double>> shapePoints = s->getPoints();
		points.insert(points.cend(), shapePoints.cbegin(), shapePoints.cend());
	});
	// On normalise les coordonnées pour que celles ci ne dépendent pas de la
position de la forme sur l'écran
	// On récupère la coordonnée x (xMin) minimale et la coordonnée y (yMin)
minimale.
	// La forme est translatée par (-xMin, -yMin)
	Point<double>::normalize(points);
	// On sauvagrde les points obtenus dans un fichier, pour ensuite pouvoir charger
ce niveau
	FileUtils::writeFile(points, "levels/save.txt");
}
```

*Figure 10 : code de la sauvegarde d'une figure*

## Déroulement d'une partie

```
void GameManager::play() {
	// Evénements SFML
	sf::Event event{};
	// On récupère les événements clavier et souris
	while (window->pollEvent(event)) {
		if (event.type == sf::Event::Closed)
			window->close();
		// On récupère l'action à effectuer selon l'événement
		Action act = actionManager->getAction(event.type);
		// On effectue l'action
		act(event);
	}
	// On vérifie si la partie est terminée ou non (condition de victoire)
	if (game->getGameState()) {
		// Si c'est le cas, on réinitialise la partie
		game = std::shared_ptr<Game>(new Game());
		// On donne la nouvelle partie au
		actionManager->setGame(game);
		// On réinitialise l'interface
		menu->clear();
		// On ajoute l'écran de victoire
		initWinScreenButtons();
	}
	// On dessine la partie et l'interface
	draw();
}
```

*Figure 11 : code du déroulement de la partie*

# MVC

Ce projet suit une architecture MVC, on peut alors le diviser en trois packages Model View et Controller dans lesquels sont répartis nos classes :

- Model : Shape, Piece, Triangle, Point
- View : Drawable, Clikable, Menu, Button
- Controller : Game Action FileUtils Clickable

# Limitations et difficultés rencontrées

## Difficultés rencontrées

Nous avons rencontré plusieurs problèmes au cours du développement du projet. Lors de la définition de la condition de victoire, il était évident que nous ne pouvions pas simplement comparer deux listes de pièces. Nous ne pouvions pas non plus comparer deux listes de points, pour la même raison : il peut il y avoir plusieurs solutions à une même figure. Cela était le plus évident sur le cas la figure la plus simple : le carré initial. S'il est tourné de 90°, la figure est équivalente, mais la liste de points est différente. c'est afin de palier à ce problème que nous avons découpé les pièces en triangles élémentaires (voir #Triangle & Pièce).

Une autre difficulté a été d'ajouter des boutons (contenant des lambdas), dans le menu. Lorsque ceux-ci étaient supprimés lors d'un changement d'état du jeu, le jeu plantait à cause d'une erreur de segmentation. Ne trouvant pas la source du problème et afin de pouvoir continuer le projet, on a d'abord fait abstraction de ce problème, causant des fuites de mémoire. Par la suite on a ajouté des pointeurs intelligents, ce qui n'a pas non plus résolu le problème. Il s'avère que la raison de problème était simple : on clique sur un bouton, par exemple, retour au menu principal, la lambda supprime les boutons du menu, celui sur lequel on vient de cliquer étant inclus. On supprimait donc le bouton contenant la lambda exécutée. Pour résoudre ce problème, on a tout simplement appelée une copie de la lambda, afin que la suppression du bouton ne soit pas un problème.

## Limitations du programme

Notre programme a quelques défauts mineurs, qui ne gênent en rien l'utilisation. Tout d'abord, il souffre de quelques fuites de mémoire, qui sont dues à l'utilisation de SDL. Ensuite, il est difficile d'avoir une interface parfaitement *responsive* avec SFML. En effet, le changement de taille de la fenêtre provoque un changement de coordonnées qui n'est pas pris en compte dans notre programme, résultant en un décalage entre l'affichage et la position réelle des pièces et boutons.

## Améliorations possibles

Une des améliorations possibles aurait été d'ajouter un système d'indice : si le joueur est bloqué, il clique sur un bouton qui place une pièce à sa place. Nous aurions pu également ajouter des options modifiable par l'utilisateur.

# Compilation

À la racine du projet :
mkdir build
cd build
cmake ..
make

Pour lancer le projet :
./Tangram

# Conclusion

Ce projet de Tangram aurait pu être programmé simplement en quelques classes, mais cela n'aurait pas été aussi robuste ni ouverts aux améliorations que comme nous l'avons fait. En effet, le fait de faire en avance a rendu plus simple la programmation des classes, et nous aurions plus de mal à ajouter des fonctionnalités comme la sauvegarde ou les previews lors du chargement. Nous avons rencontrés plusieurs difficultés lors du développement, mais nous avons pu les corriger et continuer à avancer.

Il reste plusieurs améliorations qui auraient pu être effectuées, comme des options personnalisables, ou encore l'ajout d'indice, soit sous la forme d'une pièce placée automatiquement, soit en indiquant le contour des pièces. Nous ne les avons pas implémentées par manque de temps.

# Bibliographie

GitHub du projet : https://github.com/Eradan94/Tangram

Références :

http://www.cplusplus.com/reference/
https://stackoverflow.com/questions/