

TANGRAM

Projet de C++
Rapport de modélisation

COUMES Quentin - HARDY Elise
Master 2 Informatique
2019 / 2020

Sommaire

| | |
|------------------------------|-----------|
| I. Introduction..... | 3 |
| II. Architecture..... | 4 |
| 1. Game..... | 5 |
| A. Event..... | 5 |
| B. InputState..... | 6 |
| C. Engine..... | 6 |
| D. Updatable..... | 7 |
| 2. State..... | 8 |
| A. State..... | 9 |
| B. ShapeLoaderState..... | 9 |
| C. Menu..... | 9 |
| D. Create..... | 10 |
| E. Load..... | 10 |
| F. Edit..... | 11 |
| G. Play..... | 11 |
| 3. GUI..... | 12 |
| A. Drawable..... | 12 |
| B. FadingText..... | 13 |
| C. ShapePreview..... | 13 |
| D. ButtonAbstract..... | 13 |
| E. ButtonDraw..... | 14 |
| F. ButtonText..... | 14 |
| 4. Geometry..... | 15 |
| A. Point..... | 15 |
| B. Triangle..... | 16 |
| C. Polygon..... | 16 |
| D. Shape..... | 17 |
| E. Parser..... | 17 |
| F. ParseException..... | 18 |
| III. Conclusion..... | 18 |
| IV. Annexe..... | 19 |

I. Introduction

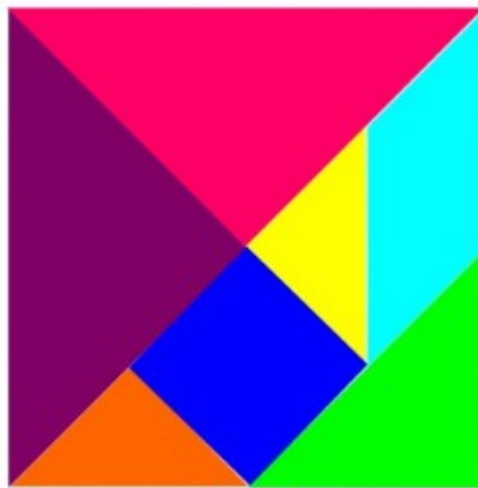
Lors de ce projet nous avons pour objectif de créer un jeu, le Tangram.

Le Tangram est un puzzle / casse-tête originaire de Chine. Composé de 7 pièces, le but est de créer / reproduire des figures.

Pour reproduire une figure toutes les pièces doivent être utilisés et ne pas se chevaucher.

Les pièces sont :

- 5 triangles dont :
 - 2 petits
 - 1 moyen
 - 1 grand
- 1 carré
- 1 parallélogramme



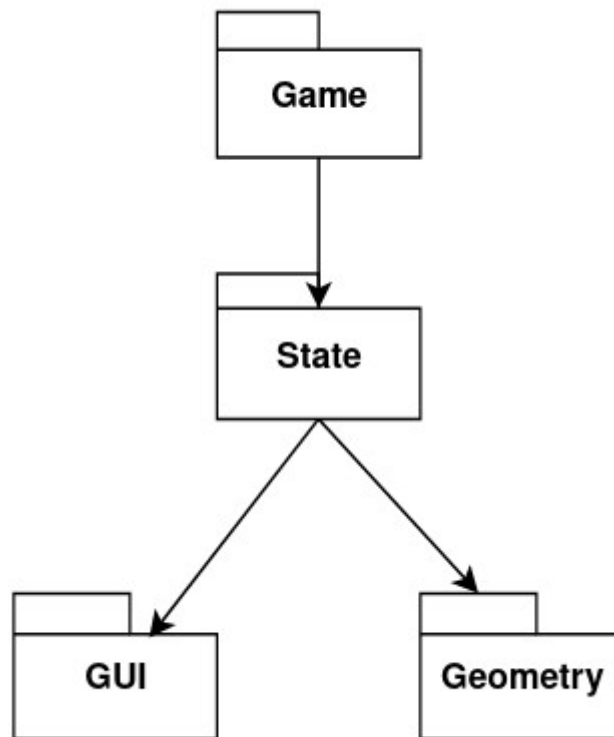
Les 7 pièces du Tangram

Ce rapport a pour objectif de décrire l'architecture utilisée lors de ce projet.

II. Architecture

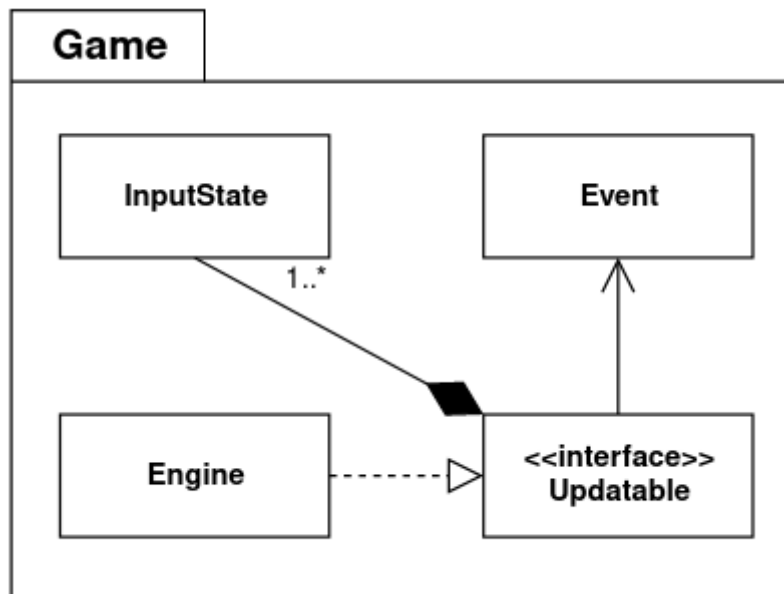
Nous avons décidé de découper notre projet en différents modules. Chaque module s'occupe d'une partie différente du jeu.

Nous avons quatre modules :



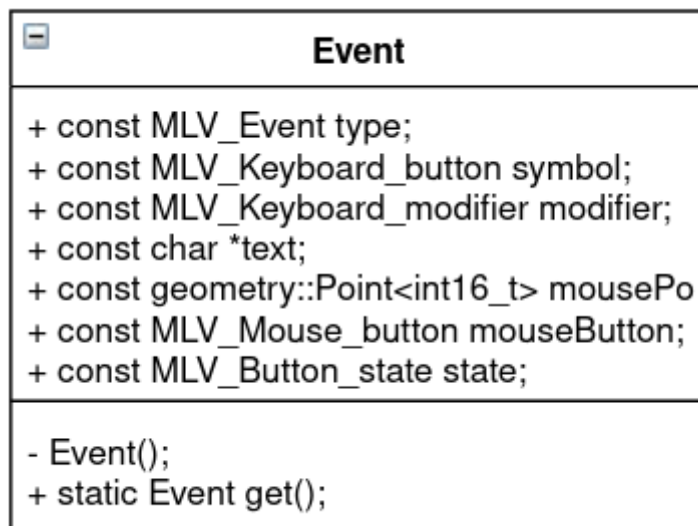
L'ensemble des UML a été simplifié pour plus de lisibilité, voir **IV. Annexe** pour les UMLs complets.

1. Game



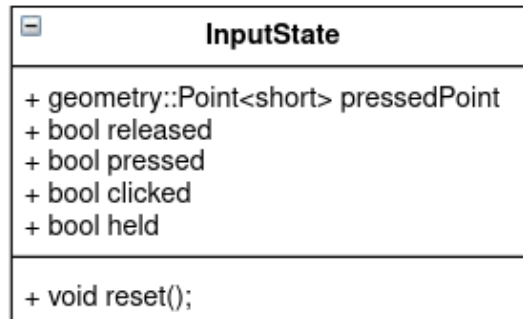
A. Event

Représente un événement telle que le mouvement d'une souris, l'appuis / où le relachement d'une touche de clavier / souris, où l'entrée d'une boîte de saisie. Dépend grandement de la librairie graphique. Une instance de cette classe ne peut être obtenue que par la méthode factorie *get()*, le constructeur étant privé. Enfin, tous les champs sont constants et de visibilité publique, afin de faciliter l'accès aux informations relatives aux événements.



B. InputState

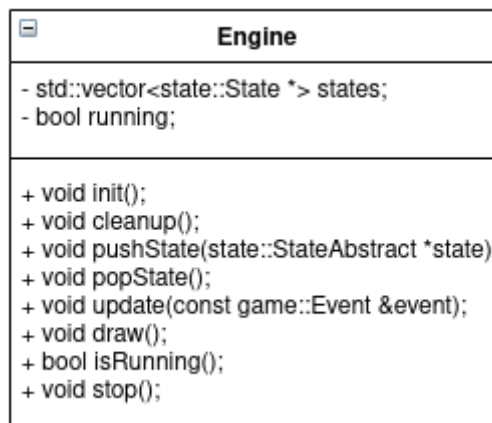
Permet de représenter l'état d'un bouton ou d'une touche de clavier.



C. Engine

Modélise l'état actuelle du jeu. Cette classe contient les méthodes appelé par la boucle *main* du jeu.

Elle utilise une pile d'états représentant les différentes parties du jeu (menu, création de pièce, chargement de pièce, etc.), l'état en haut de la pile étant l'état courant.



D. Updatable

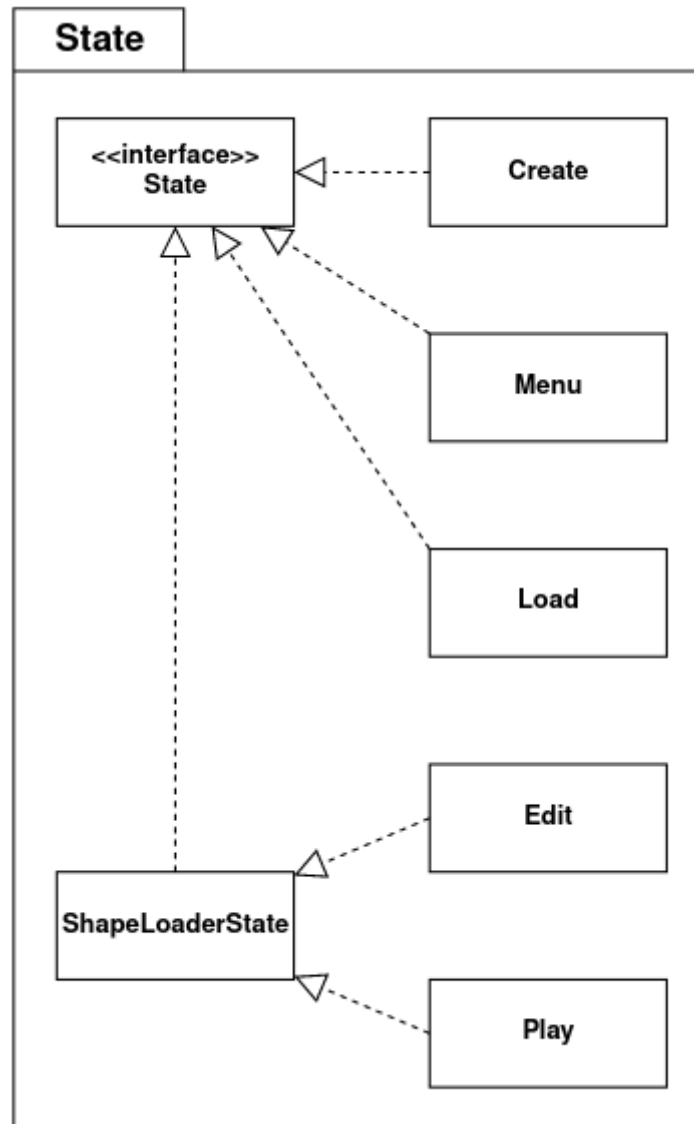
Updatable est une interface permettant à l'utilisateur d'interagir avec les classes l'implémentant. Elle implémente l'ensemble des méthode permettant de vérifier une interaction spécifique, par exemple vérifier si un bouton spécifique à été appuyé / relaché.

Les classes implémentant cette interface doivent implémenté la méthode *update()*.

| <div data-bbox="233 584 263 616">☐</div> <div data-bbox="715 593 863 647"><<interface> Updatable</div> |
|--|
| <pre># std::unordered_map<MLV_Mouse_button, InputState> buttons # std::unordered_map<MLV_Keyboard_button, InputState> keys # bool hovered; # bool enabled;</pre> |
| <pre>- void reset(); + virtual bool update(const game::Event &event, game::Engine &engine) = 0;+ bool isHovered() const; + bool isEnabled() const; + void enable(); + void disable(); + InputState getKeyInputState(MLV_Keyboard_button key) const; + bool isKeyPressed(MLV_Keyboard_button key) const; + bool isKeyHeld(MLV_Keyboard_button key) const; + bool isKeyReleased(MLV_Keyboard_button key) const; + bool isKeyClicked(MLV_Keyboard_button key) const; + geometry::Point16 getKeyPressedPoint(MLV_Keyboard_button key) const; + InputState getButtonInputState(MLV_Mouse_button button) const; + bool isButtonPressed(MLV_Mouse_button button) const; + bool isButtonHeld(MLV_Mouse_button button) const; + bool isButtonReleased(MLV_Mouse_button button) const; + bool isButtonClicked(MLV_Mouse_button button) const; + geometry::Point16 getButtonPressedPoint(MLV_Mouse_button button) const; + bool isRightPressed() const; + bool isRightHeld() const; + bool isRightReleased() const; + bool isRightClicked() const; + geometry::Point16 getRightPressedPoint() const; + bool isLeftPressed() const; + bool isLeftHeld() const; + bool isLeftReleased() const; + bool isLeftClicked() const; + geometry::Point16 getLeftPressedPoint() const;</pre> |

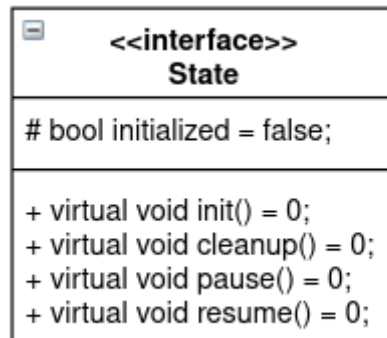
2. State

Modélise les différents états du jeu (le jeu, l'édition, les menus).



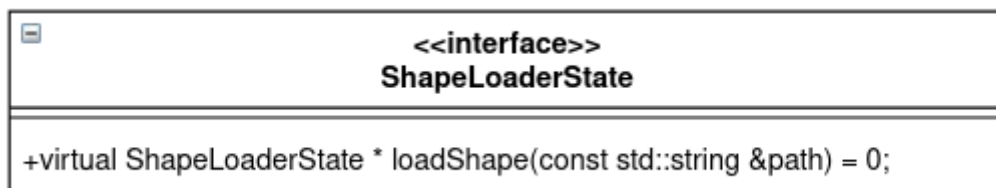
A. State

Modélise les différents états que contiendra le jeu. L'interface hérite de *Drawable* et *Updatable*. En effet on souhaite dessiner chaque états et que l'utilisateur puisse interagir avec ceux-ci. Chaque état est un *Singleton*, car le même état ne peut être présent plusieurs fois dans la liste, et cela simplifie l'interaction entre différents états.



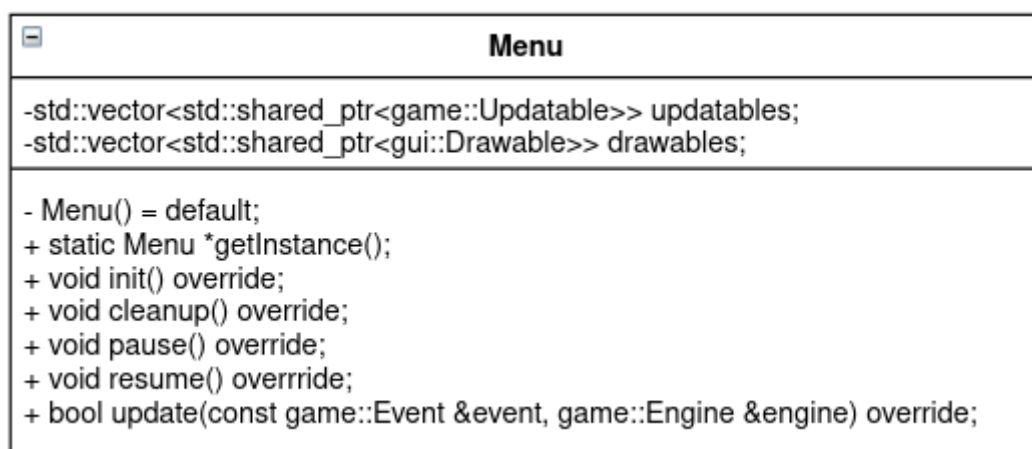
B. ShapeLoaderState

Cette interface est utilisée pour les *States* ayant besoin de charger une forme (*edit*, *play*...).



C. Menu

Modélise le menu principal. Il permet au joueur de choisir ce qu'il veut faire. Le joueur peut alors choisir entre éditer, créer une figure, jouer ou quitter. La classe contient une liste de *Drawable* ainsi qu'une liste d'*Updatable*, qui correspondent aux boutons du menu.



D. Create

Create permet de créer de nouvelle figure.

| Create |
|---|
| <ul style="list-style-type: none"> - std::vector<std::shared_ptr<game::Updatable>> updatables; - std::vector<std::shared_ptr<gui::Drawable>> drawables; - geometry::Shape player = geometry::Shape(); |
| <ul style="list-style-type: none"> - Load() = default; + static Load *getInstance(); + void init() override; + void cleanup() override; + void pause() override; + void resume() override; + bool update(const game::Event &event, game::Engine &engine) override; |

E. Load

Cette état correspond à la fenêtre de chargement des fichiers. Il permet d'afficher la liste des figures créées, ainsi qu'une *preview* correspondante.

| Load |
|--|
| <ul style="list-style-type: none"> - std::unordered_map<std::string, gui::ShapePreview> previews; - std::unordered_map<std::string, gui::ButtonText> prevButtons; - std::unordered_map<std::string, gui::ButtonDraw> delButtons; - std::unique_ptr<gui::ButtonAbstract> next; - std::unique_ptr<gui::ButtonAbstract> prev; - std::unique_ptr<gui::ButtonAbstract> menu; - ShapeLoaderState *nextState = nullptr; - std::vector<std::string> prevOrder; - uint16_t page = 0; |
| <ul style="list-style-type: none"> - Load() = default; + static Load *getInstance(); + Load *setNextState(ShapeLoaderState *nextState); + void init() override; + void cleanup() override; + void pause() override; + void resume() override; + bool update(const game::Event &event, game::Engine &engine) override; |

F. Edit

Modélise l'état permettant l'édition d'une figure.

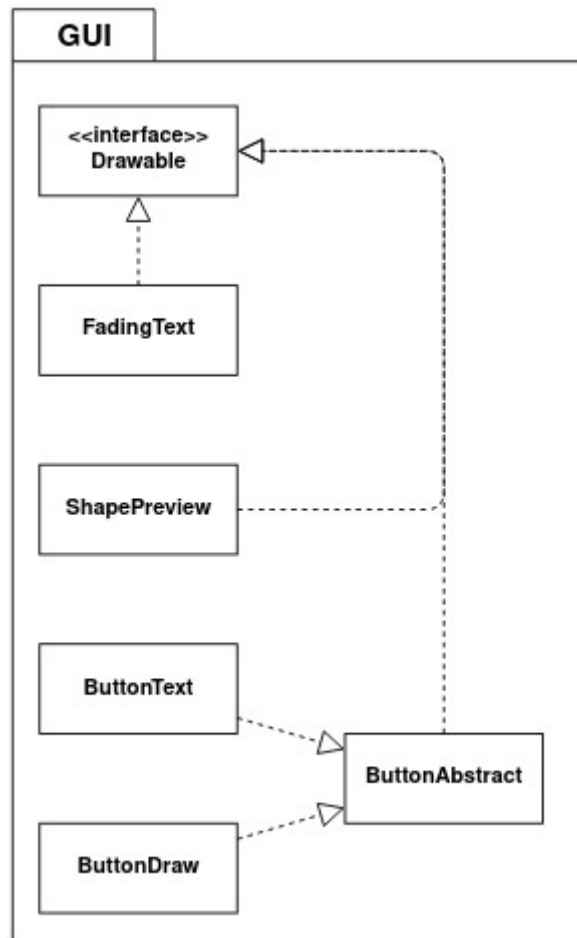
| Edit |
|---|
| <ul style="list-style-type: none"> - std::vector<std::shared_ptr<game::Updatable>> updatables; - std::vector<std::shared_ptr<gui::Drawable>> drawables; - std::shared_ptr<gui::FadingText> savedText; - geometry::Shape player; - std::string title; |
| <ul style="list-style-type: none"> - Edit() = default; + static Edit *getInstance(); + Edit * loadShape(const std::string &path) = 0; + void init() override; + void cleanup() override; + void pause() override; + void resume() override; + bool update(const game::Event &event, game::Engine &engine) override; |

G. Play

Enfin, cette état représente fenêtre de jeu principale. Cette avec cet état que le joueur essaie de reconstruire une des figures.

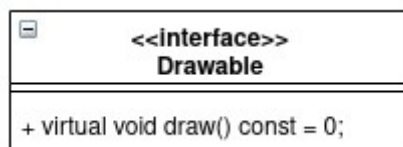
| Play |
|---|
| <ul style="list-style-type: none"> - std::vector<std::shared_ptr<game::Updatable>> updatables; - std::vector<std::shared_ptr<gui::Drawable>> drawables; - geometry::Polygon shadow; - geometry::Shape player; - geometry::Shape goal; - bool success; - std::string title; |
| <ul style="list-style-type: none"> - Play() = default; + static Play *getInstance(); + void cleanup() override; + void pause() override; + void resume() override; + bool update(const game::Event &event, game::Engine &engine) override; |

3. GUI




A. Drawable

Interface permettant de dessiner les instances des classes l'implémentant. Celles-ci doivent implémenter la méthode *draw()*.




B. FadingText

Permet d'afficher un texte qui disparaît après un court laps de temps, utile pour des notifications.

|  FadingText |
|--|
| - const std::string text; - const int16_t x; - const int16_t y; - const MLV_Color color; - int32_t width = 0; - MLV_Font *font; |
| +void rewind(); +bool update(const game::Event &event, game::Engine &engine) override; +void draw() const override; |


C. ShapePreview

Permet de d'afficher une vue simplifier de forme de Tangram.

|  ShapePreview |
|---|
| - const geometry::Shape shape; - const int16_t side; - const int16_t x; - const int16_t y; |
| +void draw() const override; |


D. ButtonAbstract

Classe abstraite représentant un bouton qui, lorsque l'utilisateur clique dessus, lance la fonction *execute()* lié à l'instance.

|  ButtonAbstract |
|--|
| # const uint16_t x; # const uint16_t y; # const uint16_t w; # const uint16_t h; # const std::function<game::Engine &> execute; |
| + Button(); |


E. ButtonDraw

Héritant de `ButtonAbstract`, cette classe permet de dessiner le bouton grâce à une fonction pour chacun de ses états : normal, survolé par une souris, lorsque l'utilisateur clique dessus.

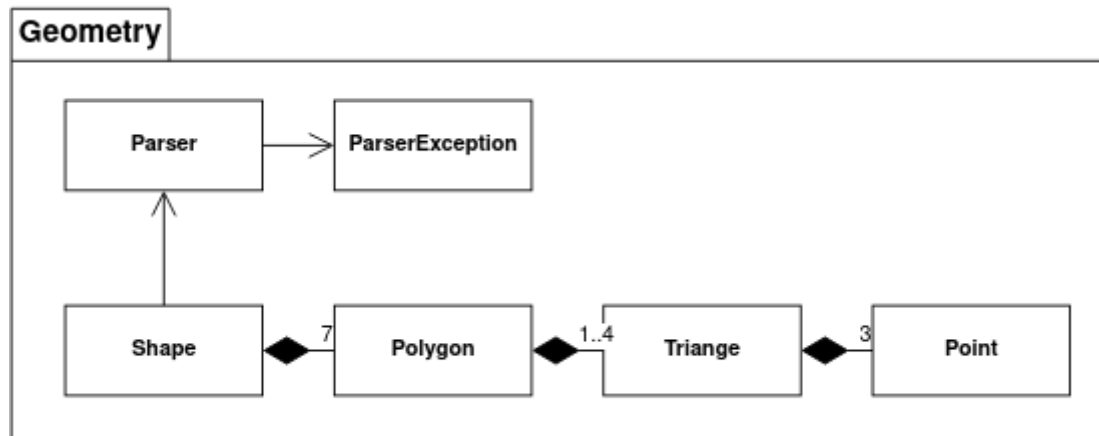
|  ButtonDraw |
|--|
| <ul style="list-style-type: none"> - <code>const std::function<void(int16_t, int16_t, int16_t, int16_t)> customDraw;</code> - <code>const std::function<void(int16_t, int16_t, int16_t, int16_t)> hCustomDraw;</code> - <code>const std::function<void(int16_t, int16_t, int16_t, int16_t)> cCustomDraw;</code> |
| <ul style="list-style-type: none"> + <code>void draw();</code> |

F. ButtonText

Ce bouton-ci dessine un bouton contenant du texte. Pour chacun des trois états mentionné sur *ButtonDraw*, il se sert de trois couleurs différentes pour le texte, la bordure, et l'arrière du bouton.

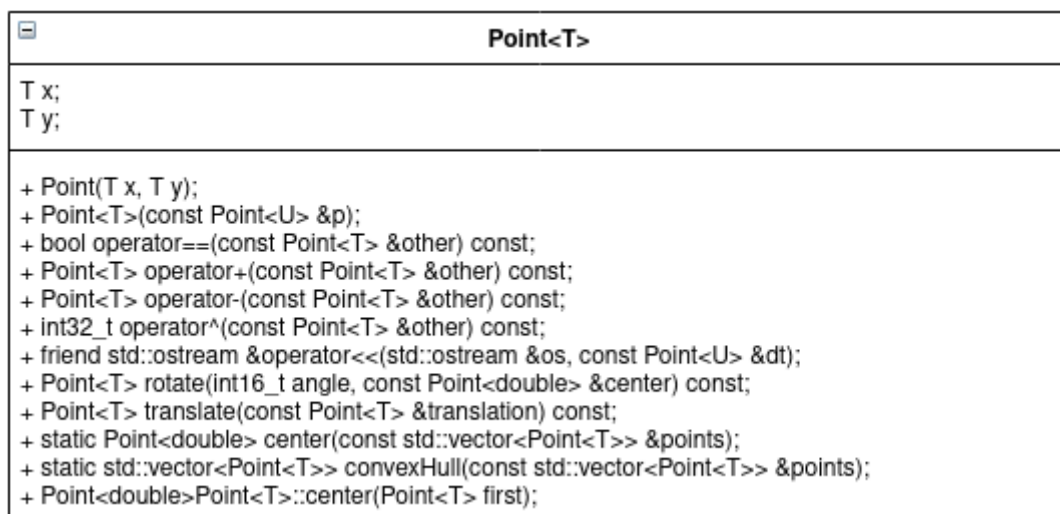
|  ButtonDraw |
|--|
| <ul style="list-style-type: none"> - <code>const std::function<void(int16_t, int16_t, int16_t, int16_t)> customDraw;</code> - <code>const std::function<void(int16_t, int16_t, int16_t, int16_t)> hCustomDraw;</code> - <code>const std::function<void(int16_t, int16_t, int16_t, int16_t)> cCustomDraw;</code> |
| <ul style="list-style-type: none"> + <code>void draw();</code> |

4. Geometry



A. Point

Classe *template* modélisant un point / vecteur 2D pouvant utilisé n'importe quel type arithmétique pour ses coordonnées.



B. Triangle

Modélise un triangle. Le triangle est la forme géométrique de base pour toutes les pièces. Il est en effet possible de toutes les décomposer en triangles unitaires. Cette classe contient les méthodes de rotation et translation d'un triangle. Ainsi que des constantes comme la taille des côtés, le pas pour la rotation.

| Triangle |
|---|
| <ul style="list-style-type: none"> - Point<uint16_t> p1; - Point<uint16_t> p2; - Point<uint16_t> p3; |
| <ul style="list-style-type: none"> + explicit Triangle(const Point<uint16_t> &offset); + Triangle(Point<uint16_t> p1, Point<uint16_t> p2, Point<uint16_t> p3) + Triangle translate(const Vector<int16_t> &v) const; + Triangle translate(int16_t x, int16_t y) const; + Triangle rotate(int16_t n, const Point<double> &center) const; + bool contains(const Point<uint16_t> &p) const; + std::vector<Point<uint16_t>> getPoints() const; + void draw(MLV_Color color) const; |

C. Polygon

Représente une des sept pièces du Tangram. Une pièce est un ensemble de 1 à 4 Triangle.

| Polygon |
|---|
| <ul style="list-style-type: none"> - std::vector<Triangle> triangles; - Point<uint16_t> center; - int16_t currentRotation; - MLV_Color color; - int16_t angle; |
| <ul style="list-style-type: none"> + Polygon() = default; + Polygon(MLV_Color color); + Polygon(const std::vector<Triangle> &triangles, MLV_Color color); + Polygon(const Polygon &) = default; + Polygon(Polygon &&) = default; + Polygon &operator=(const Polygon &) = default; + Polygon &operator=(Polygon &&) = default; + std::vector<Point16> getPoints() const; + int16_t getCurrentRotation() const; + void setCurrentRotation(int16_t currentRotation); + void setColor(MLV_Color color); + Polygon &translate(const Point<short> &v); + Polygon &translate(int16_t x, int16_t y); + Polygon &scale(const Point<double> &v); + Polygon &scale(double x, double y); + Polygon &scale(double factor); + Polygon &rotate(int16_t n); + friend std::ostream &operator<<(std::ostream &os, const Polygon &p); + bool contains(const Point16 &p) const; + void add(const std::vector<Triangle> &triangles); + void add(const Triangle &t); + bool update(const game::Event &event, game::Engine &engine) override; + void draw() const override; |

D. Shape

Modélise une figure complète composée de sept Polygons. Cette figure est celle que l'utilisateur doit créer / éditer / reproduire.

| Shape |
|--|
| - std::vector<Polygon> polygons; |
| + Shape() + Shape(std::string path); + bool operator==(const Shape &other) const; + static Shape load(const std::string &path); + Shape &translate(const Vector16 &v); + Shape &translate(int16_t x, int16_t y); + Shape &scale(const VectorD &v); + Shape &scale(double x, double y); + Shape &setColor(MLV_Color color); + std::vector<Point16> getPoints() const; + void ensureInbounds(Point16 lowerBound, Point16 upperBound); + Polygon getRandomPolygon() const; + void addPolygon(const Polygon &polygon); |


E. Parser

Foncteur permettant la lecture d'un fichier contenant les informations relatif à une figure (Shape). Le foncteur stock la ligne et la colonne courante afin de faciliter l'écriture de message d'erreur.

| Parser |
|---|
| - std::string path; - uint16_t lin; - uint16_t col ; |
| - void error(const std::string &what) const; - long parseLong(const std::string &s, uint8_t radix = 10); - bool parseBool(const std::string &s); - geometry::Point<int16_t> parsePoint(const std::string &); - geometry::Polygon parsePolygon(const std::string &s); + Parser() = default; + geometry::Shape operator()(std::string); |

F. ParseException

Exception lancée par le Parser en cas d'erreur dans le fichier lus.

|  ParseException |
|--|
| + const std::string path; + const uint16_t lin; + const uint16_t col; + const std::string msg; |
| + ParseException(); + std::string getPath() const; + uint16_t getLine() const; + uint16_t getColumn() const; + std::string getMessage() const; + const char *what() const noexcept; |

III. Conclusion

Avec ce projet, nous avons été en mesure d'en apprendre plus sur le C++, en particulier les apports de la norme **C++11**.

Nous avons appris à manipuler la Standard Template Library (**STL**), que ce soit par ces **structures** (*vector*, *list*, *map*, *function*...) ou ces **algorithmes** (*for_each()*, *find_if()*, *is_permutation()*...). Nous avons aussi appris l'avantage des **pointeurs intelligents** par rapport au pointeur hérité du C.

Nous savons maintenant comment créer des **classes templates**, permettant de généraliser certains types de classes.

IV. Annexe

