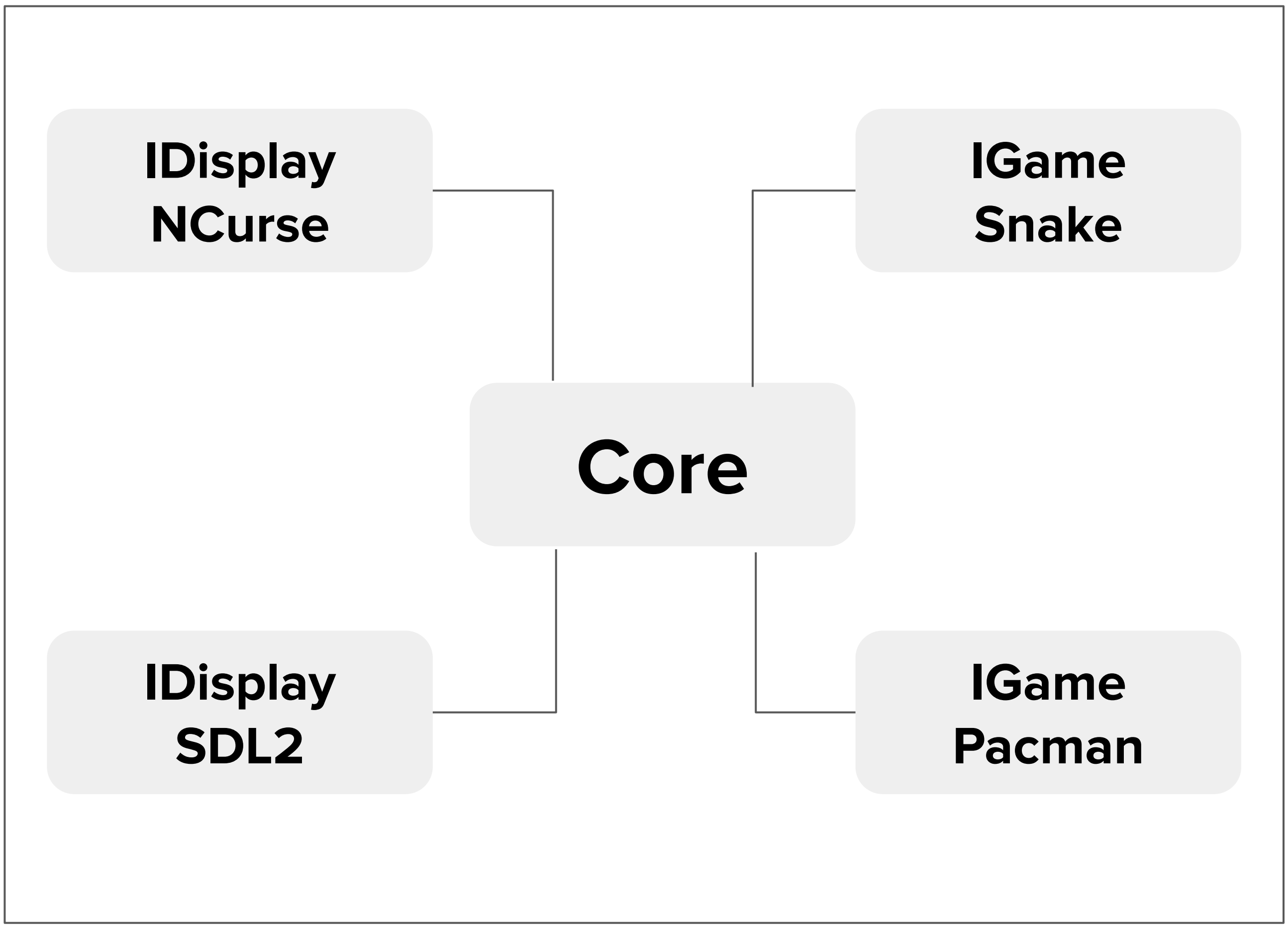
Arcade Technical Documentation

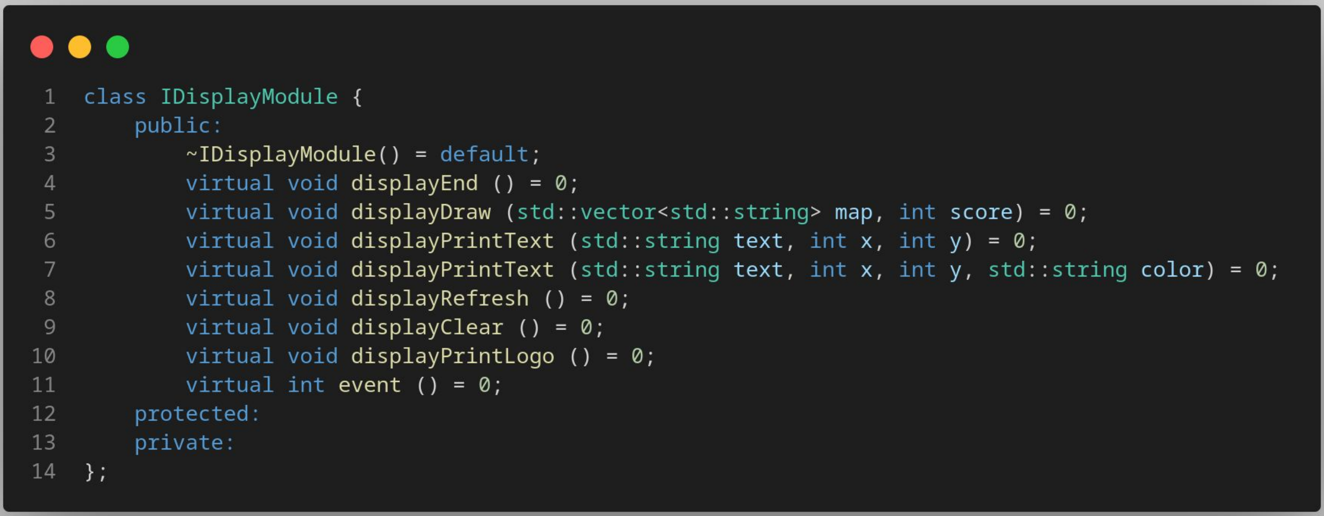
Epitech - Object Oriented Programming Module Project

Project Name: Arcade
Description: Platform allowing to play games with several graphical libraries
Authors: Kentin PAILLE and Paul THIEBAULT

Arcade is a 4 weeks project
The goal is to create a versatile gaming software that allows the user to play multiple games on several different graphical libraries.
Libraries can be dynamically loaded or swapped during execution
Highscores are saved and loaded at startup.



IDisplay Modules



```cpp
class IDisplayModule {
    public:
        ~IDisplayModule() = default;
        virtual void displayEnd () = 0;
        virtual void displayDraw (std::vector<std::string> map, int score) = 0;
        virtual void displayPrintText (std::string text, int x, int y) = 0;
        virtual void displayPrintText (std::string text, int x, int y, std::string color) = 0;
        virtual void displayRefresh () = 0;
        virtual void displayClear () = 0;
        virtual void displayPrintLogo () = 0;
        virtual int event () = 0;
    protected:
    private:
};
```

The IDisplay class corresponds to the graphical libraries used to display the menu and the games.
Every graphical library should posses theses functions as they are used during runtime.

```
void displayRefresh (void)

  takes: void
  returns: void
```

The **displayRefresh** function simply calls the refresh function from the graphical library, it is used to refresh the screen to display new elements or erase deleted ones.

displayRefresh is mandatory to display new elements in the window, it is advised to call it once inside the window's infinite loop.

```
void displayClear (void)

 takes: void
 returns: void
```

The **displayClear** function calls the clear function from the graphical library. It is used to erase all the elements shown at screen.

It is advised to use *displayClear* to get rid of old elements rendered at screen, the function should be called once inside the window's infinite loop before the *displayRefresh* function.

```
void displayEnd (void)

 takes: void
 returns: void
```

The **displayEnd** function closes the current graphical library and frees its components

DisplayEnd is called when the program stops or during a library switch. The new library is instantiated with its constructor and the old one is closed through this function

```
void displayDraw (vector<string> map, int score)
```

```
takes: a vector of strings that represents the
       map, the current player score
returns: void
```

The **displayDraw** function is used to draw the map of the game on the screen, the functions takes the score as an argument so that it is displayed in the corner and updated at runtime.

DisplayDraw is called once inside the main loop, depending on the current display library, the function calls the corresponding system function that prints the lines of the map in a loop.

```
void displayPrintText (string text, int x, int y)
```

```
takes: a string containing the text to print
       a position x, a position y
returns: void
```

```
void displayPrintText (string text, int x, int y,
string color)
```

```
takes: a string containing the text to print
       a position x, a position y, a color
returns: void
```

The **displayPrintText** function calls the display library's system function to print the string passed as argument on the screen at position (x, y)

DisplayPrintText can optionally take a color as argument, the text is then print in the selected color.

```
void displayPrintLogo (void)

 takes: void
 returns: void
```

The **displayPrintLogo** function is used to print the logo stored in the /assets/logo/ folder, the printed logo depends on the current display library

If the dimensions stays the same, the user could theoretically change the logo by replacing the asset in the corresponding folder.

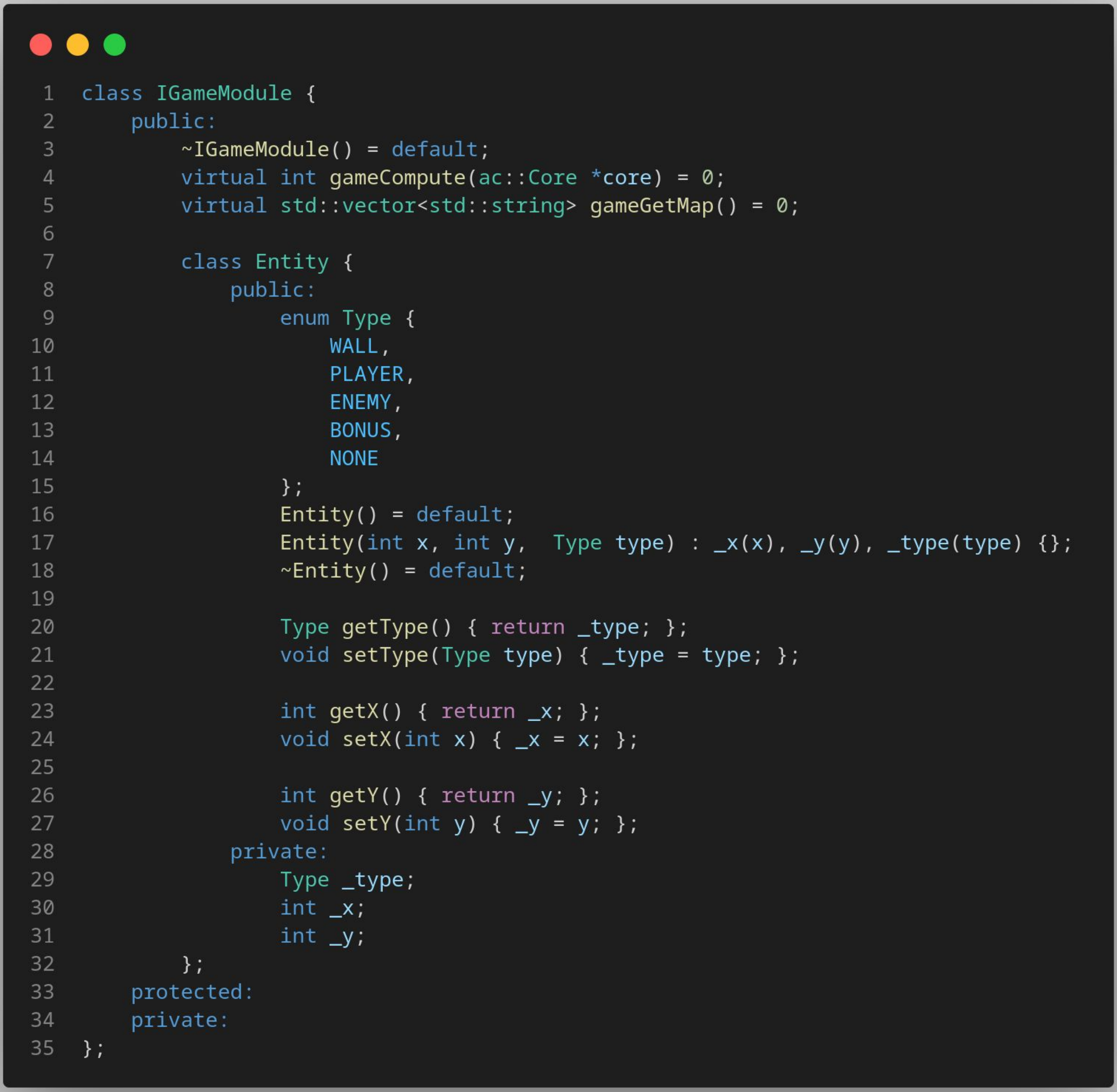
```
int event (void)

 takes: void
 returns: int corresponding to the key pressed
```

The **event** function is used to handle every keyboard input through the program. The returned value corresponds to an enum listing all the notable keys:

```
enum Key {
    UP = 1,
    DOWN,
    LEFT,
    RIGHT,
    ESCAPE,
    ENTER,
    SPACE,
    BACKSPACE,
    NONE,
    Z,
    Q,
    S,
    D
};
```

IGame Modules



```cpp
class IGameModule {
    public:
        ~IGameModule() = default;
        virtual int gameCompute(ac::Core *core) = 0;
        virtual std::vector<std::string> gameGetMap() = 0;

        class Entity {
            public:
                enum Type {
                    WALL,
                    PLAYER,
                    ENEMY,
                    BONUS,
                    NONE
                };
                Entity() = default;
                Entity(int x, int y,  Type type) : _x(x), _y(y), _type(type) {};
                ~Entity() = default;

                Type getType() { return _type; };
                void setType(Type type) { _type = type; };

                int getX() { return _x; };
                void setX(int x) { _x = x; };

                int getY() { return _y; };
                void setY(int y) { _y = y; };
            private:
                Type _type;
                int _x;
                int _y;
        };
    protected:
    private:
};
```

The IGameModule class contains an Entity class that allows to define every element of the game, such as the player, the walls of the map and even the power-ups and bonuses.

```
int gameCompute (Core *core)
```

```
takes: the instance of the Core class
returns: an int corresponding to the state of
         the current game
```

The **gameCompute** function is used to launch an instance of the game, starting the computation of the game's mechanics.

The *gameCompute* function's return value is used to check the current game's status, if the return value is 1, the game is over and the menu is brought back to the screen.

```
vector<string> gameGetMap (void)
```

```
takes: void
returns: a vector of the current map
```

The **gameCompute** function is used to launch an instance of the game, starting the computation of the game's mechanics.

```
class Entity {}

contains: Type _type
          int _x
          int _y
```

The **Entity** class contains all the elements necessary to define the different entities of the game, defined by the **Type** enum:

```
enum Type {
                     WALL,
                     PLAYER,
                     ENEMY,
                     BONUS,
                     NONE
                 };
```

Each Entity is defined by its type and position (x, y).
The said position is then modified during *gameCompute* to update the entity position and move the element across the screen.

How to add a library

In order to add a game or display library, these are the steps to follow:

- Your library must be compiled as a shared library
- Your library lust be encapsulated in C through an entry point function (named *gameEntryPoint* or *displayEntryPoint* according to the library's type)
- Your library must be compiled at the root of the *./lib/* folder
- Your libraries must not depend on any other library or arcade core