# Arcade Documentation

## How to compile ?

Our Makefile has the following rules (including *all, clean, fclean, re*):

- *core*: it build the core of the program (not the games nor the graphical librairies)
- *games*: it build games librairies
- *graphicals*: it build graphical librairies

*All* rules build core, games and graphicals at the same time.

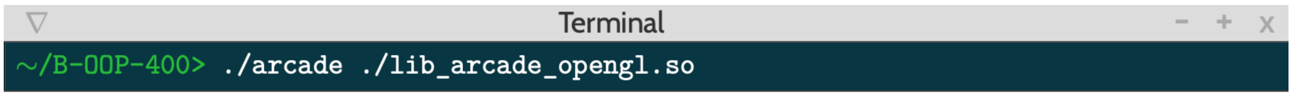
- The core build an executable that is found in: *./arcade*
- The games build libraries that are found in: *./games/*
- The graphicals build libraries that are found in: *./lib/*

## Usage :

Compile then execute the following command :

*./arcade ./lib/lib_arcade_{name of the graphical lib you want to start}.so*

*For example :*



```
~/B-OOP-400> ./arcade ./lib_arcade_opengl.so
```

## Key Configuration :

- Up Arrow : Next Game
- Down Arrow : Previous Game
- Right Arrow : Next Graphical Library
- Left Arrow : Previous Graphical Library
- Enter : Choose the game or enter your username
- P : Pause
- R : Restart
- O : Back to Menu
- Esc : Quit

## Menu :

In the menu, you have 3 options :

You can choose between 2 games.

You can also enter your name.

## Score Gestion :

Each game has a file named ./{name of the game}.score

The file is composed like that: {player}=={score}

## DLLoader :

We created a DLLoader class in a template to encapsulate the functions in dlopen(), dlsym(), dlclose(), dlerror() because they are C functions.

```
template <typename T>
class DLLoader {
    public:
        DLLoader(const std::string &libPath) {
            _handle = dlopen(libPath.c_str(), RTLD_LAZY);
            if (!_handle) {
                std::cerr << "dlopen() error : " << dlerror() << std::endl;
                exit(84);
            }
        };
        ~DLLoader();
        T *getInstance(const std::string &entryPoint) {
            T *(*ret)(void) = nullptr;
            *(void **) (&ret) = dlsym(_handle, entryPoint.c_str());
            return (ret());
        };
        void closeLib() {
            if (dlclose(_handle) != 0) {
                std::cerr << "dlclose() error : " << dlerror() << std::endl;
                exit(84);
            }
        }
    protected:
    private:
        void *_handle;
};
```

## Interfaces

### Games :

This enumeration makes it possible to know the direction of the player.

```
enum DIRECTION
{
    UP_DIR,
    DOWN_DIR,
    RIGHT_DIR,
    LEFT_DIR
};
```

Each games inherits from the following interface class : —> IGameModule

```
class IGameModule {
    public:
        virtual ~IGameModule() {};
        virtual const std::vector<std::string> &getMap() const = 0;
        virtual int refreshGame() = 0;
        virtual void makeAction(DIRECTION direction) = 0;
        virtual int getScore() const = 0;
        virtual const std::string &getName() const = 0;
        virtual const std::map<char, Entity> &getMapEntity() const = 0;
        virtual void initGame() = 0;
        virtual void resetGame() = 0;
    protected:
    private:
};
```

Methods :

- initGame —> Initialize the map and values for the game
- resetGame —> Reset and clear all the value from the concerned game
- refreshGame —> Receive the event in real time and update the game
- makeAction —> Set the new direction to follow for the game

## Graphical :

This enumeration is used to manage the interaction with the player.

```
enum KEY {
    NONE,
    END,
    PREV_LIB,
    NEXT_LIB,
    UP,
    DOWN,
    RIGHT,
    LEFT,
    PAUSE,
    RESTART,
    NEXT_GAME,
    PREV_GAME,
    MENU
}:
```

Each libraries inherits from the following interface class : —> IDisplayModule

```
class IDisplayModule {
    public:
        virtual ~IDisplayModule() {};
        //DISPLAY
        virtual void openWindow() = 0;
        virtual void closeWindow() = 0;
        virtual bool isOpen() const = 0;
        virtual void clearWindow() = 0;
        //EVENT
        virtual KEY manageEventsMenu() = 0;
        virtual KEY manageEventsGame() = 0;
        virtual KEY manageGameOver() = 0;
        //MENU
        virtual void drawMenu() = 0;
        //GAME
         virtual void drawMap(std::vector<std::string> map,
                              std::map<char, Entity> _mapEntity) = 0;
        //HIGHSCORE
        virtual void drawScores(std::string player1, int score,
                                std::string player2, int highScore) = 0;
        virtual void drawGameOver(int score) = 0;
        virtual const std::string &getPlayerName() const = 0;
    protected:
    private:
};
```

Methods :

- drawMap —> Display the map of the current game
- drawScores —> Display the score of the current game
- manageEventsMenu —> Manage all the events linked to the menu
- manageEventsGame —> Manage all the events linked to the game
- manageEventsGameOver —> Manage all the events in the Game Over screen

## Entity :

In order to make our drawMap() process generic and usable for all games, we had to create an intermediate "Entity" class that we can see below.

```
enum Shape {
    CIRCLE,
    RECT
};

class Entity {
    public:
        Entity(const Shape &shape, const std::vector<float> &size,
      const std::vector<int> &color, int ncursesColor);
        ~Entity();
        const Shape &getShape() const;
        const std::vector<float> &getSize() const;
        const std::vector<int> &getColor() const;
        int getncursesColor() const;
    protected:
    private:
        Shape _shape;
        std::vector<float> _size;
        std::vector<int> _color;
        int _ncursesColor;
};
```

When we create the map, we assimilate each character to an Entity in a vector of the game library. This one is then retrieved by the graphics library and can be display thanks to his specification and allows the drawMap() function to be generic.

Specifications :

- The shape of the entity (circle or rectangle)
- The size of the entity
- The color of the entity the color