Project Arcade

Project Arcade is a software that allows you to play arcade games using different graphic libraries. The software allows you to change **dynamically** the game and the graphic library while playing.

Authors

- Léo Le-Diouron (leo.le-diouron@epitech.eu)
- Thomas Le-Moullec (thomas.le-moullec@epitech.eu)
- Julie Bodart (julie.bodart@epitech.eu)
- Corentin Cailleaud (corentin.cailleaud@epitech.eu)

Documentation

This document contains **four parts** :

- 1 - Add a new game
- 2 - Add a new graphic library
- 3 - Make your own core
- 4 - Make your own scoreboard system

1 - Add a new game

If you want to add a new game, you must create a directory with the name of your game in the `games` directory. You must also provide a shared library in the `games` directory with the syntax : `lib_arcade_GAMENAME.so`.

Your game class must inherit from the IGame interface available in the `IGame.hpp` file.

Methods :

Each method has a specific behavior described down below :

`void Update(CommandType type, bool debug)`

This function is called when an input is pressed and when your game must play a cycle.

Parameters :

> type : the command type that is sent from the Core
>
> debug : the game must send this flag to the GetMap and GetPlayer function

`struct GetMap* GetMap(bool debug) const`

This function makes a new GetMap structure and fills it with the tile of the game. If the flag `debug` is on, the GetMap function displays the structure on the standart output. In the other case, the function return the structure without displaying it.

Parameters :

> debug : This flag changes the behavior and allows you to test your game with the `karcade` binary.

`struct WhereAmI* GetPlayer(bool debug) const`

This function makes a new WhereAmI structure and fills it with the position of the player. If the flag `debug` is on, the GetPlayer function displays the structure on the standart output. In the other case, the function return the structure without displaying it.

Parameters :

> debug : This flag changes the behavior and allows you to test your game with the `karcade` binary.

`bool IsGameOver() const`

Return `true` if the player has lost the game.

`Status GetStatus() const`

Return the status of the game. It can be either `WIN` or `LOSE` if the game is over or `RUNNING` if the player is playing. If the status is `PAUSE`, this function won't be called by the core.

`Assets& GetAssets()`

This function allows you to customize the color and the texture used to render your game. If the texture can't be loaded, the graphic library will use the Color structures.

> Be careful of the loadMap, loadPlayer and loadBg fields. Even if you don't have any textures, your constructor must initalize your assets structure with

> those three fields set to `false` in order to unset previous texture.

```
struct               Assets
{
  Color              c_player;       // The color of the player's body
  Color              c_map[8];       // The color of each tile (use TileType
  SoundType          sound;          // The sound that must be played at the
  bool               loadMap;        // true when you want to load a new map
  std::string        t_map;          // File name without the path
  bool               loadPlayer;     // true when you want to load a new play
  std::string        t_player;       // File name without the path
  bool               loadBg;         // true when you want to load a new back
  std::string        t_bg;           // File name without the path
  CommandType        dir;            // The player's orientation
};
```

`unsigned int GetScore() const`

Return the score of the player.

2 - Add a new graphic library

If you want to add a new graphic library, you must create a directory with the name of your library in the `lib` directory. You must also provide a shared library in the `lib` directory with the syntax : `lib_arcade_LIBNAME.so`.

Your graphic library class must inherit from the IGraphic interface available in the `IGraphic.hpp` file. Your constructor must initialize your window and the specific parameters that must be applied once.

Methods :

Each method of the interface has a specific behavior described down below :

`void ShowGame(WhereAmI *player, GetMap *map, Assets& assets)`

This method is called at each frame and displays the **player** and the **map** on the window. The **assets** structure give you information about textures and colors if the library can't load those textures.

Parameters :

> player : pointer to a WhereAmI structure, containing all the positions of the player
>
> map : pointer to a GetMap structure, containing all the tile that have to be displayed
>
> assets : reference on an Assets structure filled by the game that give you all the textures and colors informations.

`void ShowMenu(const std::vector<std::string>& games, size_t idgame, const std::vector<std::string>& graphics, size_t idgraphic, Button selected, const playerName& name)`

This method displays the menu on the window. You must be able to select a game and a graphic library on the menu. You can also set the name of the player.

Parameters :

> games : list of the name of all the game found
>
> idgame : the id of the selected game
>
> graphics : list of the name of all the graphic libraries found
>
> idgame : the id of the selected graphic library
>
> selected : the selected button
>
> name : a structure that contains the name of the player and the index of the letter selected. (file `ICore.hpp`)

`void ShowScore(IScore* current, const std::vector<IScore*>& bests)`

This method is called just after de **ShowGame** method. You can display the current score of the player and the highscores of the current game.

Parameters :

> current : current score of the player
>
> bests : list of the top three scores of the current game

`void GetInput(ICore *core)`

Get inputs and send it to the core using the core's methods `NotifyScene` and `NotifyGame`. This function is called at each frame and events are then supported by the core.

The method NotifyScene uses **CommandType**. You must only send GO_UP, GO_DOWN, GO_LEFT, GO_RIGHT or SHOOT. The method NotifyCore uses **CoreCommand**. **CoreCommand** is an enum defined in `ICore.hpp`.

Parameters :

> core : pointer to the core, allows you to call NotifyScene and NotifyGame

`void PrintGameOver(Status status)`

Show a message, either "YOU WIN!" or "GAME OVER", regarding the status.

Parameters :

> Status : can be WIN or LOSE

3 - Make your own core

You can implement your own core. This is the main part of the program. The constructor must save the name of the games and the libs respectively located on the `games` and the `lib` directories.

Methods :

Each method has a specific behavior described down below :

`void RunArcade()`

This method is called once one the main, after loading the graphic library given in parameter. It is the main loop of the arcade.

At each cycle, you must call the GetInput method of the graphic library and then redirect to the good **ShowSceneName** functions described below.

`void NotifyCore(CoreCommand type)`

This method is called when an input matching with a CoreCommand is pressed. Each CoreCommand matches a function that is prototype this way : `void funcName()`.

Parameters :

> type : the type of the command that must be executed

`void LoadGame(const std::string& gameName)`

This method load the game sent in parameter.

Parameters :

> gameName : name of the game that must be loaded

`void LoadGraphic(const std::string& libName)`

This method load the graphic library sent in parameter.

Parameters :

> libName : name of the library that must be loaded

`void LoadPrevGraphic()`

This method informs the core to load the previous graphic library before the next cycle

`void LoadNextGraphic()`

This method informs the core to load the next graphic library before the next cycle

`void LoadPrevGame()`

This method informs the core to load the previous game library

`void LoadNextGame()`

This method informs the core to load the next game library

`void Pause()`

This method informs the core to toggle on and off the cycle process.

`void Restart()`

This method unloads the current game and reloads it.

`void Menu()`

This method informs the core to show the menu instead of the game

> You can use the Scene enum to change the status

`void Quit()`

This method informs the core to stop the program.

> You can use the Scene enum to change the status

`void ShowSceneMenu()`

This method sends informations to the graphic library to display the menu.

`void ShowSceneGame()`

This method sends informations to the graphic library to display the game and depending of the Status.

> You can use the Status enum to switch from one state to another

`void ShowSceneScoreboard()`

This method sends informations to the graphic library to display the scoreboard scene.

`void NotifyScene(CommandType type)`

This method is called when an input matching with a CommandType is pressed (arrows and shoot button). Each CommandType matches a function that is prototype this way : `void NotifySceneNAME()`.

Parameters :

> type : the type of the command that must be executed

`void NotifySceneMenu(CommandType type)`

This method is called by NotifyScene and interprets the CommandType on the menu.

Parameters :

> type : the type of the command that must be executed

`void NotifySceneGame(CommandType type)`

This method is called by NotifyScene and interprets the CommandType while playing.

Parameters :

> type : the type of the command that must be executed

`void NotifySceneScoreboard(CommandType type)`

This method is called by NotifyScene and interprets the CommandType while displaying the entier scoreboard.

Parameters :

> type : the type of the command that must be executed

4 - Make your own scoreboard system

You can change the scoreboard system by creating your own scoreboard system that inherit from IScoreBoard (available in the `IScoreBoard.hpp` file)

Methods :

Each method has a specific behavior described down below :

`void PostScore(const std::string& game, const std::string& name, unsigned int score)`

`void PostScore(IScore* s)`

This function push a new score in the scoreboard.

Parameters :

> game : game's name
>
> name : player's name
>
> score : player's score
>
> s : pointer to a structure that inherits from IScore.

```
class IScore
{
public:
  virtual ~IScore() {};
  virtual const std::string&  getGame() const = 0;
  virtual const std::string&  getName() const = 0;
  virtual unsigned int        getScore() const = 0;
  virtual void                setGame(const std::string&) = 0;
  virtual void                setName(const std::string&) = 0;
  virtual void                setScore(unsigned int) = 0;
};
```

`const std::vector<IScore*>& GetBestScores(const std::string& game, size_t size)`

This function return a sorted vector that contains the X(`size`) best scores of the game `game` .

Parameters :

> game : game's name
>
> size : number of best scores wanted

`void GetFileBestScores()`

Load the score from a file or database in the class.