Arcade

Project Presentation

Arcade is a gaming platform: a program that lets the user choose a game to play and keeps a register of player scores.

Arcade implements different displays libraries and game libraries, loaded at runtime. You can play your favourite retro games like Nibbler or Pacman, while being able to switch to the display of your liking, whether its text mode with Ncurses, or graphical mode with SDL2, SFML, Allegro…

This document presents how you can add one of these libraries to the Arcade with minimum development time.

Project conventions

In order to keep the code structured, and ease third-party integration, this project follows some software development conventions:

- The control version system is **Git**, following the conventional commit convention
- The coding style is the **Google C++ Coding Style**
- The testing framework is **GTest**
- The build system is **make**
- The documentation system is **Doxygen**

Please follow these conventions if you wish to add a library to the Arcade.

How to create Display Libraries

1. Create a folder under lib/Display/the_name_of_your_library

Ex (from root): mkdir ./lib/Display/Allegro

2. Copy the template Makefile for displays and fill in the blanks

Template Makefile :

```
##
## EPITECH PROJECT, 2021
## Arcade
## File description:
## Makefile
##

CC          =   g++

SRC         =   NAME OF YOUR LIBRARYDisplay.cpp
OBJ         =   $(SRC:.cpp=.o)

SRC_UT      =   $(addprefix $(SRC_UT_D), $(SRC_UT_F))
OBJ_UT      =   $(SRC_UT:.cpp=.o)
SRC_UT_D    =   tests/
SRC_UT_F    =   NAME OF YOUR TESTS FILES, UNDER tests/

INC         =   -I../../inc

CXXFLAGS    =   -std=c++17 -fPIC $(WFLAGS) $(INC)

WFLAGS      =   -W -Wall -Wextra -Werror

DBFLAGS     =   -g -g3 -ggdb

LDFLAGS     =   YOUR LIBRARY FLAGS

LDFLAGS_UT  =   -lgtest -lgtest_main -lpthread

NAME        =   arcade_name of your library in lowercase.so

NAME_UT     =   unit_tests

all: $(NAME)

%.o: %.cpp
    $(CC) $(CXXFLAGS) -fPIC -c $< -o $@ $(LDFLAGS)

$(NAME): $(OBJ)
    $(CC) $(CXXFLAGS) --shared -o $(NAME) $(OBJ) $(LDFLAGS)
    mv $(NAME) ../../

debug: $(OBJ)
    $(CC) $(CXXFLAGS) $(DBFLAGS) -o $(NAME) $(OBJ)

tests_run: clean $(OBJ) $(OBJ_UT)
    $(CC) $(CXXFLAGS) -o $(NAME_UT) $(OBJ) $(OBJ_UT) $(LDFLAGS_UT)
    ./$(NAME_UT)
```

```
coverage:
    gcovr -s --exclude tests/

clean:
    rm -f $(OBJ)
    rm -f $(OBJ_UT)
    rm -f *.gc*

fclean: clean
    rm -f ../../$(NAME)
    rm -f $(NAME_UT)

re: fclean all
```

You are free to modify this Makefile if you know what you are doing.

3. *Create your display class*

First, create the source and header file of the class. The naming convention of display libraries for this project is the name of the library + “Display”.

Ex: touch AllegroDisplay.hpp AllegroDisplay.cpp

For now, it is generally sufficient to put only one source and header file at the root of your display directory.

Next, create your display class. The class should be within the arc namespace, and inherit from the IDisplay Interface.

Ex: In AllegroDisplay.hpp

```
#include "IDisplay.hpp"

namespace arc {
class AllegroDisplay : public IDisplay {
        …
}
```

Copy the following functions after your library definition, still within the arc namespace

```
#include "IDisplay.hpp"

namespace arc {
…

extern "C" LibType getLibType(void) {
   return DISPLAY;
}

extern "C" IDisplay *create(void) {
   return new THE NAME OF YOUR CLASS; // ex: return new AllegroDisplay;
}

extern "C" void destroy(IDisplay *display) {
   delete display;
}
}  // namespace arc
```

These functions allow you to be part of the arcade !

getLibType specifies that your library is a Display library rather than a game, and the two other functions, called class factory functions, allows the arcade to create and destroy your display class.

It is also a good idea to create your display class error class. If you wish to do so, import the header file “Error.hpp” and inherit from the DisplayError class.

Ex: in Allegro.hpp

```
#include “Error.hpp”

class AllegroError : public DisplayError {
 public:
    explicit AllegroError(std::string const &message)
        : DisplayError(std::string("allegro: ") + message) {}
};
}  // namespace arc
```

4. Implement the IDisplay functions

Once you have finished these steps, you can now start the implementation of your display library !

Be careful when implementing the getName() function, it should always return a string in lowercase:

```
/**
* @brief Get the name of the library
*
* @return "allegro"
*/
std::string getName(void) const override {
    return "allegro"; <- ALWAYS IN LOWERCASE !
}
```

Check the doxygen documentation of the IDisplay Interface for more information on how the member functions should behave, and don’t hesitate to check on the other libraries to get more insight on the implementation.

Since this project uses doxygen, please make documentation for your own implementations.

How to create Games libraries

1. Create a folder under lib/Games/the_name_of_your_library

Ex (from root): mkdir ./lib/Games/Qix

2. Copy the template Makefile for displays and fill in the blanks

Template Makefile :

```
##
## EPITECH PROJECT, 2021
## Arcade
## File description:
## Makefile
##

CC          =   g++

SRC         =   NAME_OF_YOUR_GAME.cpp
OBJ         =   $(SRC:.cpp=.o)

SRC_UT      =   $(addprefix $(SRC_UT_D), $(SRC_UT_F))
OBJ_UT      =   $(SRC_UT:.cpp=.o)
SRC_UT_D    =   tests/
SRC_UT_F    =   NAME OF YOUR TESTS FILES, UNDER ./tests

INC         =   -I../../inc -I./

CXXFLAGS    =   -std=c++17 $(WFLAGS) $(INC)

WFLAGS      =   -W -Wall -Wextra -Werror

DBFLAGS     =   -g -g3 -ggdb

LDFLAGS     =   -L../../ -lparser

LDFLAGS_UT  =   -lgtest -lgtest_main -lpthread

NAME        =   arcade_name_of_your_game_in_lowercase.so

NAME_UT     =   unit_tests

all: $(NAME)

%.o: %.cpp
    $(CC) $(CXXFLAGS) -fPIC -c $< -o $@

$(NAME): $(OBJ)
```

```
    $(CC) $(CXXFLAGS) --shared -o $(NAME) $(OBJ) $(LDFLAGS)
    mv $(NAME) ../../

debug: $(OBJ)
    $(CC) $(CXXFLAGS) $(DBFLAGS) -o $(NAME) $(OBJ)

tests_run: clean $(OBJ) $(OBJ_UT)
    $(CC) $(CXXFLAGS) -o $(NAME_UT) $(OBJ) $(OBJ_UT) $(LDFLAGS) $(LDFLAGS_UT)
    ./$(NAME_UT)

coverage:
    gcovr -s --exclude tests/

clean:
    rm -f $(OBJ)
    rm -f $(OBJ_UT)
    rm -f *.gc*

fclean: clean
    rm -f ../../$(NAME)
    rm -f $(NAME_UT)

re: fclean all
```

You are free to modify this Makefile if you know what you are doing.

3. Create your game class

First, create the source and header file of the class.

Ex: touch Qix.hpp Qix.cpp

For now, it is generally sufficient to put only one source and header file at the root of your game directory.

Next, create your game class. The class should be within the arc namespace, and inherit from the IGame Interface.

Ex: In Qix.hpp

```
#include "Qix.hpp"

namespace arc {
class Qix : public IGame {
        ...
}
```

Copy the following functions after your library definition, still within the arc namespace

```
#include "IGame.hpp"

namespace arc {
…

extern "C" LibType getLibType(void) {
    return GAME;
}

extern "C" IGame *create(void) {
    return new THE NAME OF YOUR CLASS; // ex: return new Qix;
}

extern "C" void destroy(IGame *game) {
    delete game;
}
}  // namespace arc
```

These functions allow you to be part of the arcade !

getLibType specifies that your library is a game library rather than a display, and the two other functions, called class factory functions, allows the arcade to create and destroy your game class.

It is also a good idea to create your game's error class. If you wish to do so, import the header file "Error.hpp" and inherit from the GameError class.

Ex: in Qix.hpp

```
#include "Error.hpp"

class QixError : public GameError {
 public:
    explicit QixError(std::string const &message)
        : QixError(std::string("qix: ") + message) {}
};
}  // namespace arc
```

4. Implement the IGame functions

Once you have finished these steps, you can now start the implementation of your game !

Just be careful when implementing the getName() function, it should always return a string in lowercase:

```
/**
* @brief Get the name of the library
*
* @return "qix"
*/
std::string getName(void) const override {
    return "qix"; <- ALWAYS IN LOWERCASE !
}
```

Check the doxygen documentation of the IGame Interface for more information on how the member functions should behave, and don’t hesitate to check on the other libraries to get more insight on the implementation.

Since this project uses doxygen, please make documentation for your own implementations.