Arcade Documentation

This is a document is the documentation for our Arcade project.
It will include an explanation of the Interfaces, the file structure
how to build a new game/graphical library and a how to add
prebuild libraries using the same interfaces.

Group Members: Alexander Prosch, Jonas Conrad and Johannes Timmreck

Overview

1. Interfaces

1.1. COMMAND

```
enum COMMAND {
    NO_INPUT,
  // change libraries
    PREV_GRAPH,
    NEXT_GRAPH,
    PREV_GAME,
    NEXT_GAME,
  // change game state
    RESTART,
    PAUSE,
    MAIN_MENU,
    GAME,
    EXIT,
  // game relevant user input
    UP,
    RIGHT,
    DOWN,
    LEFT,
  // TYPE BACK_SPACE KEY
    ACTION
};
```

This is the enumeration for the player input which get returned by the IGraphicalInterface::getInput function and applied in the core and the IGameInterface::applyInput.
This enumeration is for standardising the input so not every game has to implement every single key for every library.

1.2. IObjectToDraw

```
class IObjectToDraw {
  public:
    enum Type {
        objects,
        text
    };

    virtual ~IObjectToDraw() = default;
    // get id which instances of this type get referenced to
    virtual std::string getId() const = 0;
    // path to directory of assets [terminal.txt, texture.png, mesh.*]
    virtual std::string getPath() const = 0;
    // get enum Type [objects|text]
    virtual Type getType() const = 0;
    // get string to display for texts
    virtual std::string getValue() const = 0;
    // get the coordinates of the current instance (x, y)
    virtual std::pair<long int, long int> getCoords() const = 0;
    // get position of sprite in texture for animation(x, y, width, height)
    virtual std::tuple <uint, uint, uint, uint> getSpritePos() const = 0;
    // get color of the asset(0-255, 0-255, 0-255, 0-255)
    virtual std::tuple<uint, uint, uint, uint> getColor() const = 0;
};
```

This interface is a standardized form of what gets sent between the game library and the graphics library. Its the Interface which describes the different objects which should be displayed on the screen.

1.3. IGameInterface

```
class IGameInterface
{
  public:
    virtual ~IGameInterface () = default;

    // initialize game
        virtual bool start() = 0;
    // end game
        virtual bool end() = 0;
    // return the assets for display
        virtual std::vector<IObjectToDraw *> getAssets() = 0;
    // apply user input
        virtual COMMAND applyInput (COMMAND userInput) = 0;
    // calculate next game frame and return it
        virtual std::vector<IObjectToDraw *> compute() = 0;
};
```

This is the interface through which the arcade core interacts with the game libraries.

start()

function which gets called directly after the constructor,
before any other interaction with this interface

end()

function which gets called directly before the destructor,
after any other interaction with this interface

getAssets()

function which gets called to receive one IObjectToDraw of each asset
which should later be displayed for this game

applyInput()

function which applies the user input to the game and returns the COMMAND
unchanged if the core should apply the change, otherwise it return NO_INPUT

compute()

function which holds the complete game logic independent from the
user input, calculates the next frame and returns the IObjectToDraw which
should be displayed

1.4. IGraphicalInterface

```
class IGraphicalInterface
{
  public:
    virtual ~IGraphicalInterface() = default;

  // initialize window
    virtual bool init() = 0;
  // initialize assets for each type of object
    virtual bool initAssets(std::vector<IObjectToDraw *> assets) = 0;
  // destroy assets for each type of object
    virtual bool destroyAssets() = 0;
  // destroy window
    virtual bool destroy() = 0;
  // get User Input
    virtual COMMAND getInput() = 0;
  // draw next frame
    virtual bool draw(std::vector<IObjectToDraw *> objects) = 0;
};
```

This is the interface through which the arcade core interacts with the graphics libraries.

init()

function which gets called directly after the constructor before any other interaction with this interface happens

initAssets()

function which loads the assets as IObjectToDraw the next game will want to display

destroyAssets()

function which destroys the assets of the last game so new ones can be created or the object can be destroyed

destroy()

function which gets called directly before the destructor after any interaction with this interface happens

getInput()

function which returns the COMMAND given by the user

draw()

function which draws the vector IObjectToDraw it gets to the screen

2. File Structure

2.1. Core

The core is the main logic of the arcade program. It contains the program loop and handles interactions with the libraries.

```
├──── core
│    ├──── assets
│    ├──── include
│    ├──── Makefile
│    └──── src
```

assets

directory with the assets for the main menu

include

directory with core header files

Makefile

file which contains the build information for the core

src

directory with the core source code files

2.2. doc

```
├──── doc
│    ├──── Documentation.pdf
│    └──── src
```

Documentation.pdf

this documentation in pdf format

Documentation.html

this documentation in html format

src

the source for this documentation

2.3. games

```
├──── games
│    ├──── IGameInterface.hpp
│    ├──── Nibbler
│    │    ├──── assets
│    │    ├──── include
│    │    ├──── Makefile
│    │    └──── src
│    └──── Pacman
│         ├──── assets
│         ├──── include
│         ├──── Makefile
│         └──── src
```

The directory which includes all the game directories
and will contain the shared libraries once a game is built.

IGameInterface.hpp

contains the IGameInterface interface

Nibbler

directory of our first game Nibbler

Pacman

directory of our second game Pacman

2.3.1. structure of a game library

assets

directory for the assets for the game

include

directory for the header file of the game

Makefile

file with build information

src

directory for the source files of the game

```
├──── GlobalHeaders.hpp
```

Global headers.hpp

File containing the COMMAND enum and the IObjectToDraw interface

---

2.4. lib

```
├──── lib
│    ├──── IGraphicalInterface.hpp
│    ├──── nCurses
│    │    ├──── include
│    │    ├──── Makefile
│    │    └──── src
│    └──── SFML
│         ├──── include
│         ├──── Makefile
│         └──── src
```

The directory which includes all the graphical library directories
and will contain the shared libraries once these are built.

IGameInterface.hpp

contains the IGameInterface interface

nCurses

directory of our terminal library nCurses

SFML

directory of our multimedia library SFML

2.4.1. structure of a graphical library

include

directory for the header file of the library

Makefile

file with build information

src

directory for the source files of the library

```
├──── Makefile
└──── README
```

Makefile

file containg the rules to built the core, all the games
and all the graphical libraries

README

file containing the email addresses of the repositories our group shared the interfaces with

3. Adding new graphical library

3.1. build new library

completely build a new graphics library module

Step 1
create a directory in the /lib/ directory

Step 2
create a class which inherits from the IGraphicalInterface and implement it

Step 3
implement an entryPoint to the library so it can get loaded by the core

Step 4
add a Makefile to built the shared library and move to the /lib/ folder

3.2. add a shared library

add a shared grapphics library which uses the same interfaces

Step 1
add the shared library into the /lib/ directory

4. Adding new game library

4.1. build a new library

completely build a game library

Step 1
create a directory in the /game/ directory

Step 2
create a class which inherits from the IGameInterface and
create a class which inherits from the IObjectToDraw and
implement them, take care that the tiles of your tilemap should
have a size of 32x32 pixel so they work both with terminal and
other graphical libraries

Step 3
create an assets directory inside of your game directory created in **Step 1**

Step 4
implement an entryPoint to the library so it can get loaded by the core

Step 5
add a Makefile to built the shared library and move to the /games/ directory

4.2. add new library

add a shared game library which uses the same interfaces

Step 1
add the shared library into the /games/ directory

Step 2
create a directory with the name of the game in the /games/ directory
and add the assets in there, for each asset there needs to be a director
having a terminal.txt, a texture.png and optionally a mesh file

Authors Notice

version v1.2 written on 26.03.2020 by Johannes Timmreck