PROAV - Four in a Row in C++

This Lab consists in programming the popular game Four in a Row in C++, in a simple graphical environment. The library that will be used for this purpose is called piksel, whose code is freely available in a github repository.

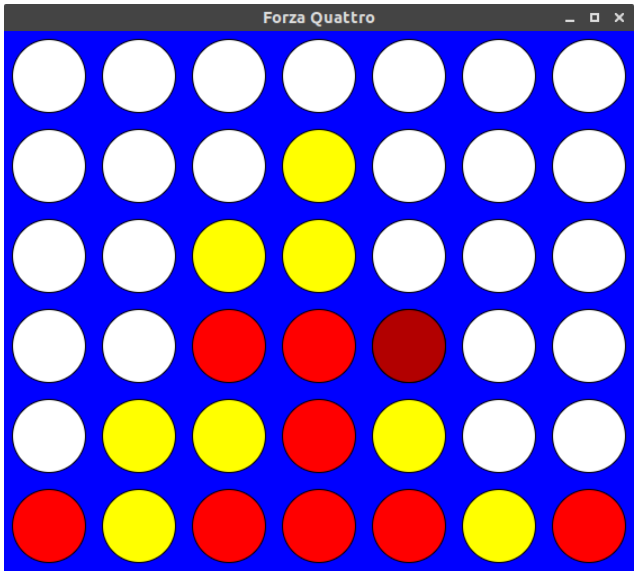

Figure 1: The Four in a Row game created using piksel.

1 Using piksel

To make sure that you will be able to do the lab, we will first download and test the piksel library. Afterwards, you will be given information about few useful methods that you will need to create the four in a row application.

1.1 Compiling the library

Using a terminal, move to the main folder of the assignment and download piksel using:

```
git clone --recursive https://github.com/bernhardfritz/piksel.git
```

To test if everything works fine, create in the project root a new folder named `src`. Add the new file `piksel_example.cpp` and paste the following code inside it[1]:

```cpp
#include <piksel/baseapp.hpp>

class App : public piksel::BaseApp {
public:
  App() : piksel::BaseApp(640, 480, "Hello Piksel") {}

  // called once on startup
  void setup() override {
    // select random background color
    std::srand(time(NULL));
    double r = (std::rand() % 256)/255.;
    double g = (std::rand() % 256)/255.;
    double b = (std::rand() % 256)/255.;
    color_ = glm::vec4(r, g, b, 1);
  }

  // main loop function
  void draw(piksel::Graphics& g) override {
    // set background and draw a circle
    g.background(color_);
    g.ellipse(mouse_x_, mouse_y_, 60, 60);
  }

  // called whenever the mouse is moved
  void mouseMoved(int x, int y) override {
    mouse_x_ = x;
    mouse_y_ = y;
  }

private:
  float mouse_x_; // last known x-coordinate of the mouse
  float mouse_y_; // last known y-coordinate of the mouse
  glm::vec4 color_; // background color
};



int main() {
  // create the app and run it
  App app;
  app.start();
}
```

To compile the code, add a `CMakeLists.txt` file inside the project root directory, and include the following lines of code[2]:

[1] The file can be copied directly from `solution/src/piksel_example.cpp`

[2] The name of the project corresponds to the Italian version of the game, feel free to name it differently.

```
cmake_minimum_required(VERSION 3.1)
project(forza_quattro)

# use C++11 features
set(CMAKE_CXX_STANDARD 11)

# tell CMake to compile piksel as well
add_subdirectory(piksel)

# compile the example
add_executable(piksel_example src/piksel_example.cpp)
target_include_directories(piksel_example PUBLIC piksel/src)
target_link_libraries(piksel_example piksel)
```

Do not bother understanding it for now, we will come back to it in the next section. Time to compile: create a new folder named `build` and move inside it, then type the command `cmake ..` to start configuring. You might get an error concerning the version of CMake (piksel requires 3.12), but do not worry! The official solution is to install the correct version of CMake, which does not require a lot of work. For simplicity, the steps are reported here:

```
mkdir -p ~/programs_sources && cd ~/programs_sources
version=3.12
build=1
wget https://cmake.org/files/v$version/cmake-$version.$build.tar.gz
tar -xzvf cmake-$version.$build.tar.gz
cd cmake-$version.$build/
./bootstrap
make -j4
sudo make install
```

To check if the installation went fine, try to type `cmake --version` on your console: you should see the newly installed version 3.12.1. Also, as a good rule, do not delete the folder that contains the compiled CMake sources, *e.g.*, `~/program_sources/cmake_3.12.1`. In fact, if in the future you wish to remove the installed version, it will suffice to move inside such directory and type `sudo make uninstall`. As a plus, from now on you can use the folder `program_sources` to install programs from their sources.

BE CAREFUL: by default, the new CMake will be installed in `/usr/local`. If you have already a CMake binary installed there, the procedures above might override it completely. For a safer procedure, make a backup before typing `sudo make install`. You can do that simply by running:

```
sudo cp -p /usr/local/bin/cmake.3.10.1
```

The example assumes you have CMake 3.10.1 installed, change it to reflect the correct version!

A "quick-n-dirty" workaround is also possible: simply open the file `piksel/CMakeLists.txt` and change the required version of CMake to the one that you have, *e.g.*, 3.1. Note that with this solution the build might fail completely or that some piksel's functionalities might not work as intended.

Coming back to compiling piksel, after a successful configuration type `make` to build the executable and finally `./piksel_example` to run it. You should see a rectangular window appearing in your screen, with a solid colored background. In addition, when moving the mouse inside the window, a white circle should be drawn following you. If so, then congratulations, piksel has been correctly compiled!

1.2 Understanding piksel's fundamental classes

Applications in piksel are rather simple: they all inherit from the base class `BaseApp`, which exposes a couple of virtual methods that can be overridden to customize the behavior of the program. In particular, the methods we are interested in are:

`BaseApp(int,int,std::string)` Constructor, takes the size (in pixels) of the window and optionally a string representing the "title" of the application.

`virtual void setup()` Called once as soon as the application starts.

`virtual void draw(piksel::Graphics&)` Called repeatedly by the application, each call can be used to "render" a single frame. The `Graphics` object can be used to add shapes inside the window.

`virtual void mousePressed(int)` Called when a mouse button is pushed down (not "clicked"). The integer parameter allows to distinguish between different buttons on the mouse.

`virtual void mouseMoved(int,int)` Executed when the mouse changes position inside the window. The two input variables correspond to the current horizontal and vertical coordinates (in pixels) of the device.

void run() This is the "entry-point" of the application (generally invoked in `main`). You can think of it as a call to `setup` followed by an infinite loop that repeatedly calls `draw`.

The other fundamental class used in this library is `Graphics`, which is used to draw inside the canvas. We will not go into details, and we will just focus on few methods:

void background(glm::vec4) Allows to set the color of the background to the given one.

void ellipse(float,float,float,float) Draws an ellipse in the specified coordinates (first two parameters of the function) with given width and height (third and fourth variables).

void fill(glm::vec4) Changes the color used to draw shapes, such as ellipses.

void text(std::string,float,float) Prints some text at the given coordinates.

void textSize(float) Changes the size for following calls to `text`.

Finally, colors are all declared using the class `glm::vec4`, whose constructor accepts four `float` parameters, representing the RGBa (red, green, blue and alpha) components of the color, all normalized between 0 and 1.

Looking at the example code compiled in the previous section, it should be rather easy to understand what happens. First of all, we create a new class `App`, inheriting from `BaseApp`. The

constructor simply initializes the canvas by calling the constructor of the base class. `App` has three private members: the background color of the canvas and the "recorded" mouse position. The color is initialized to a random one in the `setup` method. We also override the virtual method `mouseMoved`, whose default implementation is empty, so that the new coordinates of the mouse are saved whenever it is moved. Finally, the `draw` method will fill the background with the random color generated in `setup` and also draw a circle of fixed size in the recorded mouse position. To actually run the code, in the `main` method we create an instance of `App` and "execute" it by calling `start` (inherited from `BaseApp`).

Hoping that everything is clear up to this point, we are ready to start the actual assignment!

2 Four in a Row

Let's start by introducing the rules of the game:

- A rectangular grid of $R \times C$ cells is initially empty
- Each of the two players is assigned a color, either red or yellow
- Players alternate in selecting one column out of the C available ones and they are not allowed to pass
- A fully filled column cannot be selected
- Once a column is selected, a chip is inserted in it reaching the lowest free cell
- The first player who aligns four or more chips in any direction (horizontal, vertical, diagonal) wins the game

With these rules in mind, it is decided to implement the game as follows:

- An *abstract* class `Player` provides the *interface* for a player. It will expose a *pure virtual method* called `choose` that asks to select a column. Concrete implementations of this class will define different strategies, *e.g.*, using the mouse to select a column (human player) or randomly selecting a valid column (a very bad AI).
- The class `Board` will serve two objectives at the same time: on one hand, it will store and handle the grid by exposing methods to query/update cells content, and on the other hand it will take care of graphics management by inheriting from `BaseApp` and overriding some of its methods.