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.

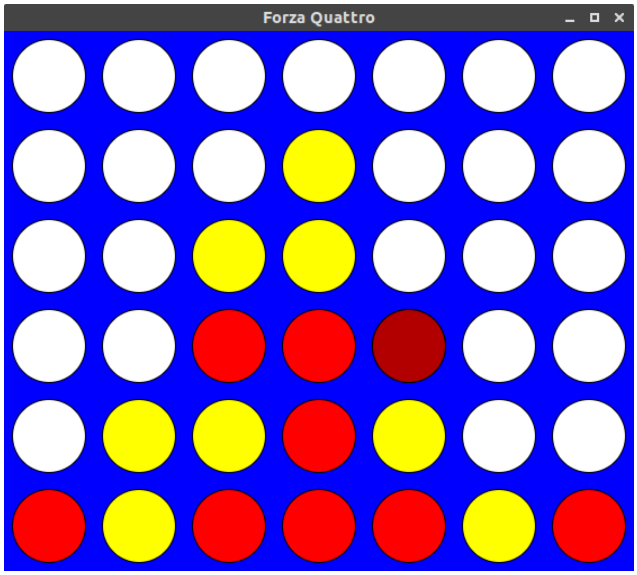

Figure 1: The Four in a Row game created using piksel.

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 choice must always be made)
- 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.

In the following, you will be given some suggestions to implement these classes. As almost always in programming, there exist many different ways to accomplish the same task, and thus you might decide to use different strategies. If you feel like there is a better way to implement the code, you are encouraged to give it a try!

2.1 Class `Board` - Basics

This section concerns the creation of the basic functionalities of the `Board` class, *i.e.*, those not requiring the class `Player`. While adding and implementing new methods, remember to compile and run the code frequently in order to spot errors as soon as possible.

First of all, define a new empty class that inherits from `piksel::BaseApp`. Create the header and source file in the usual way: declarations go in the `.h` file, while implementations (the instructions of each method) should be in the `.cpp`. Remember to add the proper `#ifndef ... #define` guards and to include relevant headers.

2.1.1 The `Cell` Enum

In order to represent the content of a cell in the board, we will use an internal enum named `Cell`. Its values should reflect the possibility of a cell being empty or filled with a red/yellow chip. You can declare it – *inside* the class definition – as:

```
enum class Cell {
  EMPTY,
  RED,
  YELLOW
};
```

Being an internal class of `Board`, it is accessible via the syntax `Board::Cell`, and one of its value by writing, *e.g.*, `Board::Cell::EMPTY`. Remember to add the declaration in the *public* scope of the class, so that external code fragments can use it as well.

2.1.2 Member Variables

To avoid using hard-coded numbers here and there in the code, let's store the geometric parameters of the grid inside two *static* members:

```
static const int CELL_SIZE;
static const int CHIP_DIAMETER;
```

The former is used to store the size of the grid (or, equivalently, the distance between two chips) while the latter gives the diameter of a single chip (it should be smaller than the size of the cell). Remember to initialize them in the beginning of the `board.cpp` file!

In addition, we will need some variables to store the grid. We can do that by adding the following *private* members:

```
const unsigned int rows_, cols_;
std::vector<Cell> grid_;
```

As their names suggest, `rows_` and `cols_` contain the size of the grid, *i.e.*, R and C. The content of the grid is stored in the vector `grid_`, whose size will be set later to $R \times C$. Note that we use a "linear" structure (the vector) to store a matrix-like content (the grid). This is a pretty common choice in programming, and we will see later how to provide utility methods to access its content in a simple way.

2.1.3 Constructors

Declare three constructors with the following signatures:

```
Board();
Board(unsigned int size);
Board(unsigned int rows, unsigned int columns);
```

The first one should build a 6×7 grid using *constructor delegation*. Similarly, the second version will generate a squared grid with `size` rows and columns. The last constructor is the one that actually performs all the initialization: it should call `BaseApp`'s constructor[3], save the number of rows/columns in the grid and initialize `grid_` to the proper size, with all cells being initially empty.

Even with this thin definition, you can already try executing some code to start debugging the class. Create a new source file containing a `main` that instanciates a `Board` object and runs the application.

2.1.4 Simple accessors

We can proceed by providing simple accessors to the grid properties. Add the following declarations to the public scope of your class:

```
const unsigned int& rows() const;
const unsigned int& cols() const;
const Cell& cell(unsigned int r, unsigned int c) const;
```

The first two methods should simply return the internal members `rows_` and `cols_`. Please note the `const` qualifiers both before and after the method declaration: do you understand why they are there? If not, ask the teacher!

The method `cell` should be used to simplify the access to elements contained inside the vector `grid_`. In practice, it should return the value of the cell located at the given row `r` and column `c`, and its implementation depends on how you choose to handle the content of the vector. To give an example, a matrix-like structure can be stored in a vector by writing it "row by row", also known as row major ordering. In this case, the index k corresponding to a given row i and column j can be evaluated as

$$k = i \times C + j$$

2.1.5 Drawing the board

We can now display the content of the grid, using the two methods below:

[3]The size of the canvas should be related to the number of rows and columns in the grid, as well as to the `CELL_SIZE`

```
void cellCenter(
  unsigned int r,
  unsigned int c,
  float& x,
  float& y
);

void draw(piksel::Graphics& g) override;
```

Both methods can be put in the private scope of the class. The first one is an auxiliary method that takes as an input the row and column indices of a cell and evaluate the coordinates of its center in the canvas.

The method `draw` overrides `BaseApp`'s one and should render the grid. To do that, you can simply loop over all rows and columns of the grid and draw an ellipse for each cell depending on its content (use the methods `cell` and `cellCenter`, altogether with those provided by piksel's `Graphics` class).

2.1.6 Additional cells methods

This basic version of the `Board` class is almost finished. We just need two methods to facilitate accessing and changing the cells in the grid:

```
public: bool next(unsigned int c, unsigned int& r) const;
private: Cell& cell(unsigned int r, unsigned int c);
```

The method `next` takes the index `c` of a column and returns `false` if no additional chips can be inserted there. If the column has still some space instead, the index of the first free row (from the bottom of the grid!) is stored in `r` and `true` is returned instead.

The second method is almost identical to the `cell` accessor defined in section 2.1.4. The difference is that the returned reference is not constant, *i.e.*, its content can be changed. This is the main reason to put the method in the private scope: anybody can read the value of one cell, but we want the `Board` class only to be able to change the state of the grid.

Having added these methods, you can try to manually change the initial state of the grid in the constructor of the class. Use the method `next` to evaluate valid rows for the given columns and use `cell` to change their value. Clearly, this manual change is just for temporary testing and we will remove it in the next sections.

2.2 Class `Player`

Let's leave the `Board` class for a moment – we will come back to it later – so that we can start implementing the `Player` class. We will see in the following how to define it and how to implement new player types for our game.

2.2.1 Protected member variables

The attributes of the `Player` class will be just two, the color used by the player and a reference to the board used for the game:

```
Board::Cell color_;
Board& board_;
```

Thanks to the `Board` reference, the player will be able to query the current state of the grid and thus to make decisions for the following move.

Place the two variables in the *protected* scope of the class, so that they are not directly accessible from outside, but still visible to children classes.

Finally, provide an accessor and a mutator for the attribute `color_`.

2.2.2 Constructor and destructor

Add a constructor that takes a reference to a `Board` instance. In the `.cpp` file, make sure to properly initialize the internal `board_` reference. In addition, add a virtual destructor as follows:

```
virtual ~Player() = default;
```

To understand the declaration, let's first say that writing `~Player() = default;` is equivalent to providing an empty destructor, *i.e.*, a destructor whose code is just `{}`.[4] However, I believe it is clearer and nicer to write explicitly `~Player() = default;` rather than `~Player() {}`. That being said, having a virtual destructor is not always necessary, and in our specific case would not be mandatory. However, it is part of best practices to add a (even empty) virtual destructor *whenever your class has at least one virtual method*. More information about this can be found in the C++ FAQ entry "*When should my destructor be virtual?*". It is not a big deal if you do not fully understand this, but at least be aware that if you plan to do inheritance, there are chances that you will need to provide a virtual destructor to your base class, even if empty.

2.2.3 Virtual methods

Add the two following public virtual methods to the class:

```
virtual bool choose(unsigned int& col) = 0;
virtual void start() {}
```

The first one is a *pure virtual* method, thus making the class *abstract*. Its goal is to choose the next move by storing in `col` where to insert the chip. The method will return `true` if a choice is made, and `false` if the player "needs more time to think" (it will be useful when implementing a "human" player).

The method `start` can instead be useful to "reset a player" before starting a game – a single player might play multiple times! However, a default (empty) implementation is already provided for simplicity: it is up to child classes to decide whether it is useful to override it.

2.2.4 Wait method

Finally, we can add a simple auxiliary function (make it as protected):

[4] For this reason, you do not need to add the definition of the destructor in the `.cpp` file!

```
void wait(int ms);
```

The function should wait for the given amount of `ms` milliseconds. To implement it, you can use the `BaseApp::millis` method, which returns the elapsed time (in milliseconds) since the start of the application. Such pausing function will be useful when implementing "AI players".

2.2.5 Class `RandomPlayer`

We have enough tools now to implement our first (rather inefficient) player type. Create the class `RandomPlayer` inheriting from `Player`, and override the `choose` method so that a valid column is chosen at random. If all columns are full, the method should return `false` since a choice cannot be made!

2.3 Class `Board` – integrating players

We can now add players to our "Four in a Row" game. We will need to introduce new members to the existing `Board` class, to store the player and to let him/her/it choose columns during the game.

2.3.1 Adding players

To add a player, let's start by including the relevant header and by adding two variables to the class `Board`:

```
Player* red_player_;
Player* yellow_player_;
```

Make sure to initialize the two pointers to `nullptr` in the constructor, so that in the beginning they point to a known invalid address – they could otherwise contain any value and it would be impossible to know whether they are valid or not.

Note that if you compile the code you will almost certainly get a compilation error like:

```
error: 'Player' does not name a type: Player* red_player_;
```

This is due to something that is (sometimes) denoted as "circular dependency". This happens when a class `A` uses/has a member of type `B`, which in turn uses/has a member of type `A`. The problem can be solved using a *forward declaration*, which in our case corresponds to adding right before the declaration of `Board` the "incomplete" declaration of `Player`:

```
class Player; // forward declaration

class Board {
  ...
  // The compiler does not really know how the 'Player' class works
  // but at least it now knows that something like that exists
  Player* red_player_;
};
```

Up to now, the pointers to the `Player` instances are simply `nullptr`. To allow assigning players properly, let's add the public method

```
void setPlayers(Player& red, Player& yellow);
```

which should copy the address of the two given objects into the internal pointers. Make sure to check that the addresses are different, *i.e.*, that the user did not call `setPlayers(x,x)`. You can now modify the main program to include the players before the game starts.

2.3.2 Making players choose

Add two private member variables to `Board`:

```
bool red_playing_;
Cell winner_;
```

the former used to know whose turn it is now, the latter to know who is the winner (we can use `Cell::EMPTY` to indicate that no one won yet).

Now add the private method `void update()` which will be used to ask the current player to pick a column. To implement it, try to follow the given pseudo-code:

```
void Board::update() {
  // select the current player depending on the value of 'red_playing_'
  // (store the active player in a reference called 'p'

  // call p.choose(column): if it returns false, exit immediately

  // if a column was selected, check that it has space:
  // if not, throw an exception

  // get the row in which the chip should be inserted, then use
  // the method 'cell' to insert the chip in the grid - the color
  // to be used is p.color()

  // TODO LATER: check if the player won

  // switch the current player
}
```

Finally, add the following at the very beginning of the `draw` method:

```
if(winner_ == Cell::EMPTY)
  update();
```

which ensures that, if no one won yet, the players are continuously asked to make a choice.

2.3.3 Right before starting a game

To make sure that everything is properly setup before starting a game, we can override the method `setup` from `BaseApp`. Inside it, you should do a couple of operations:

In addition, make the method assign the correct colors to the players, *e.g.*, by calling

```
void Board::setup() {
  // clear all the values in the grid, to make sure they are EMPTY

  // initialize red_playing_

  // initialize winner_

  // check that the Player pointers are not nullptr
  // (throw an exception if they are)

  // set the color of the red and yellow player
  // using Player::setColor

  // call Player::start
}
```

And that's it! Well... almost!

2.3.4 Checking who won

You can start the game with two random players, but they will keep on playing until the grid is entirely filled. This is because we did not check yet whether a player won after making a move. Go back to the `update` method, and try to figure out how to check if the current player won the game.

Hint: you can try to use a recursive auxiliary function that counts how many cells of the same color there are in a given direction (left, left+up, up, right+up, right, *etc.*).

In addition, in the `draw` method you can add a message that says which player won (if any).

2.3.5 Resetting the game with a "click"

As the very last step of this basic implementation, let's add the ability to reset a game by a mouse click. We can easily do that by overriding the method `mousePressed`, so that whenever the mouse is pressed the game is reset (just by calling `setup`) if the game has ended.

2.4 Support for human players

To implement a nicer interface allowing a human to play the game, we will add few functionalities to the existing `Board` class. In particular, we will add support to handle mouse motion, mouse clicks and a "preview mode" to help visualizing the effect of the next move.

2.4.1 Mouse motion support

When a human is playing, he/she will use the mouse to decide in which column the chip should be placed. For this purpose, we can add the following to our `Board` class:

```
private: int mx_;
private: void mouseMoved(int x, int y) override;
public: unsigned int mouseColumn() const;
```

The variable `mx_` will store the current horizontal position of the mouse in the canvas, which can be updated inside `mouseMoved` (using the input variable x only). The method `mouseColumn` can instead be used to return the current column selected using the mouse. To obtain such column, you need to perform a very simple operation using `mx_` and `CELL_SIZE`.

2.4.2 Handling mouse clicks

Mouse clicks can be handled in a similar way, by adding the following members to our class:

```
private: bool mouse_pressed_;
public: bool mousePressed();
```

The variable `mouse_pressed_` can be set to `true` inside the `mousePressed(int)` method added in section 2.3.5 – provided that the game is not reset! Then, the new method `mousePressed()` will return `true` if `mouse_pressed_` has been changed, and reset it to `false`. In practice, the method returns `true` if the mouse has been clicked at least once since its last call.

2.4.3 Previewing a move

We can implement a sort of "move preview" by adding few lines of code to our class:

```
private: unsigned int preview_col_;
public: void preview(unsigned int col);
```

The variable `preview_col_` can be used to store in which column we would like to place a chip, even if we are not sure about it yet – and this is why we want to see a preview! To "disable" the preview, we can decide that if the variable above is greater than (or equal to) the number of columns, then the preview should not be shown. The method `preview` is used to change `preview_col_` to a desired column index.

To actually display the preview, we can add in the `draw` method few lines of code that will add an ellipse in the desired location. By setting the color to either dark red or dark yellow, the "preview chip" will be distinguishable from those that have already been placed in the grid.

2.4.4 Adding the `HumanPlayer` class

You can now define a human player by adding a new class `HumanPlayer` inheriting from `Player`. The only thing you need is to override the `choose` method, whose pseudo-code is reported below:

```
bool HumanPlayer::choose(unsigned int& col) {
  // get currently selected column from the board

  // if the column is not a valid index, exit returning false

  // if the selected column is full, exit returning false

  // if the mouse has been pressed, exit returning true

  // if the mouse has not been pressed: only show a preview
  // and finally exit returning false
}
```

Change the main program so that it allows a human player to challenge the "random AI".

What next?

The assignment is over, but you can try to improve and/or extend the code in many ways. Some examples:

- When a player wins, show which chips are aligned in a row
- Create new (hopefully more powerful) AIs
- Add a "revert" functionality to the game, allowing to cancel the last move(s) when the backspace key is pressed

Have fun, and happy coding!