SET09121 Games Engineering

Workbook 2018

Dr Kevin Chalmers
Sam Serrels

Edinburgh Napier University

January 16, 2018

Version 0.92

Contents

List of Figures

List of Algorithms

Lesson 1

Getting Started

Welcome to Games Engineering!

The pre-requisite knowledge you require is:

- a working knowledge of object-oriented programming in a high-level language. It an ideal world this is C++ but Java and C# programmers should be able to cope with the material.
- at least a grasp of mathematical concepts such as trigonometry, algebra, and geometry. Linear algebra and matrix mathematics are very advantageous.
- a willingness to spend time solving the problems presented in this workbook. Some of the exercises are challenging and require effort. There is no avoiding this. However, solving these problems will significantly aid your understanding.
- Git version control basics

1.0.1 Git Repos

The practical content is available from a Git repository and via Moodle.

```
git clone https://github.com/edinburgh-napier/set09121
```

or

```
git clone git@gitgud.napier.ac.uk:set/set09121.git
```

You could make a fork of this, or create your own code repo from scratch. (We will walk through this now). Later on you will need to create a separate repository for your coursework, which you will share with one other developer. But for now, we are just dealing with practical content. As you work through the practicals, commit your work to your repo.

1.0.2 Development Environment

You will be able to work on this module on any operating system that has a modern C++ compiler. We will be using the CMake build system, so you won't need to use a specific IDE. We recommend Visual Studio 2017 on Windows, and Clion on Mac/Linux. This module expects that you follow best practices when it comes to software development: i.e use version control

effectively and properly, maintain a clean codebase, and implement testing procedures(more on this later). You will also need Git and CMake installed, and added to your PATH.

SFML You may be glad to know that for this module, the Rendering of your games projects will be handled for you. We will be making use of the 2D Games/Graphics library SFML www.sfml-dev.org. This handles many of the low-level grunt-work that goes into making games, allowing us to focus on the high level design. There is also a wide array of tutorials and pre-existing support available around the internet if you run into a SFML-specific problem.

1.0.3 Workbook Style

The workbook is task and lesson based, requiring you to solve problems in each chapter to continue to the next. Unlike other games modules, no code will be given for you to start with, some listings will be provided in the workbook that you may copy verbatim, but even these may require you to fix them. For clarity; missing sections you have to complete are highlighted in stars:

```
1 int score = 3;
2 while(true){
3 // **********************************
4 // You have to complete this problem
5
6
7 // **********************************
8 }
```

In some cases, there may not be "missing code", but rather larger sections of logic that you will have to solve in your own way, rather than filling in the blanks.

```
1 auto h = renderer::window->GetHeight();
2 auto w = renderer::window->GetWidth()
3 //Calculate aspect ratio
4 //Scale Ui accordingly
```

The problems faced generally build on previous lessons until you become familiar with the work involved.

1.1 Getting started for real

1.1.1 Step 1 - Create a Github Repo for your project

Let's create our repo to work in, you can do this either via github first and pull down, or create locally and then push up. We'll start via github (or any repository hosting website of your choice i.e bitbucket / gitgud.anpier.ac.uk).

If you have not already created a github account, create one and sign in.

(As a student you get some cool swag from github: https://education.github.com/pack)

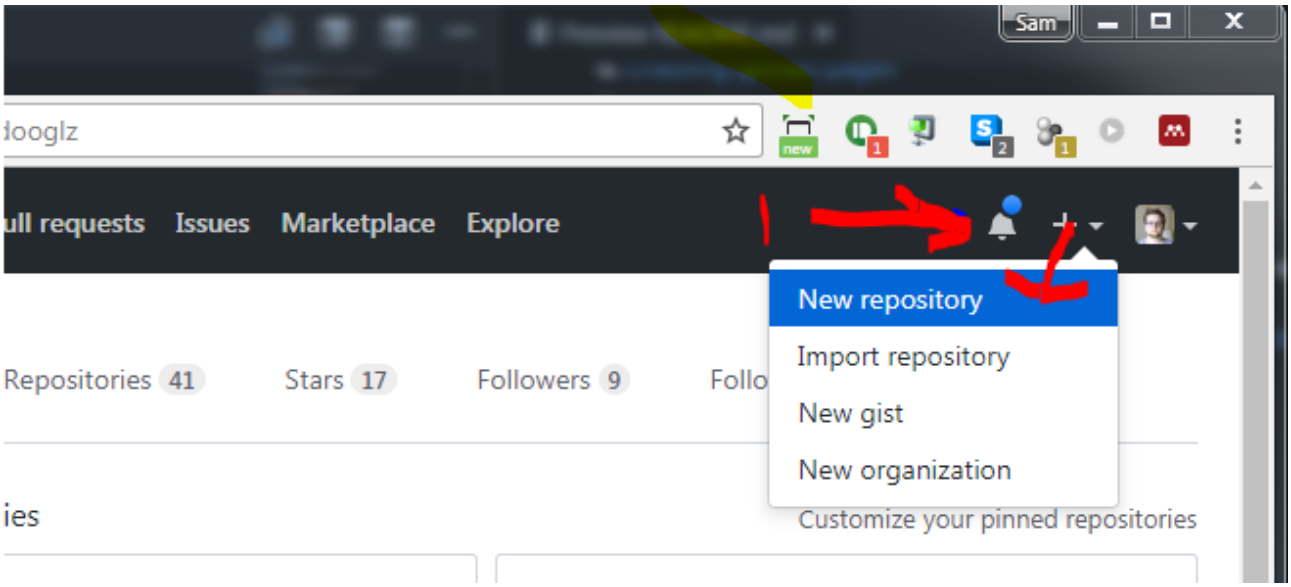


Figure 1.1: **New Repo process**

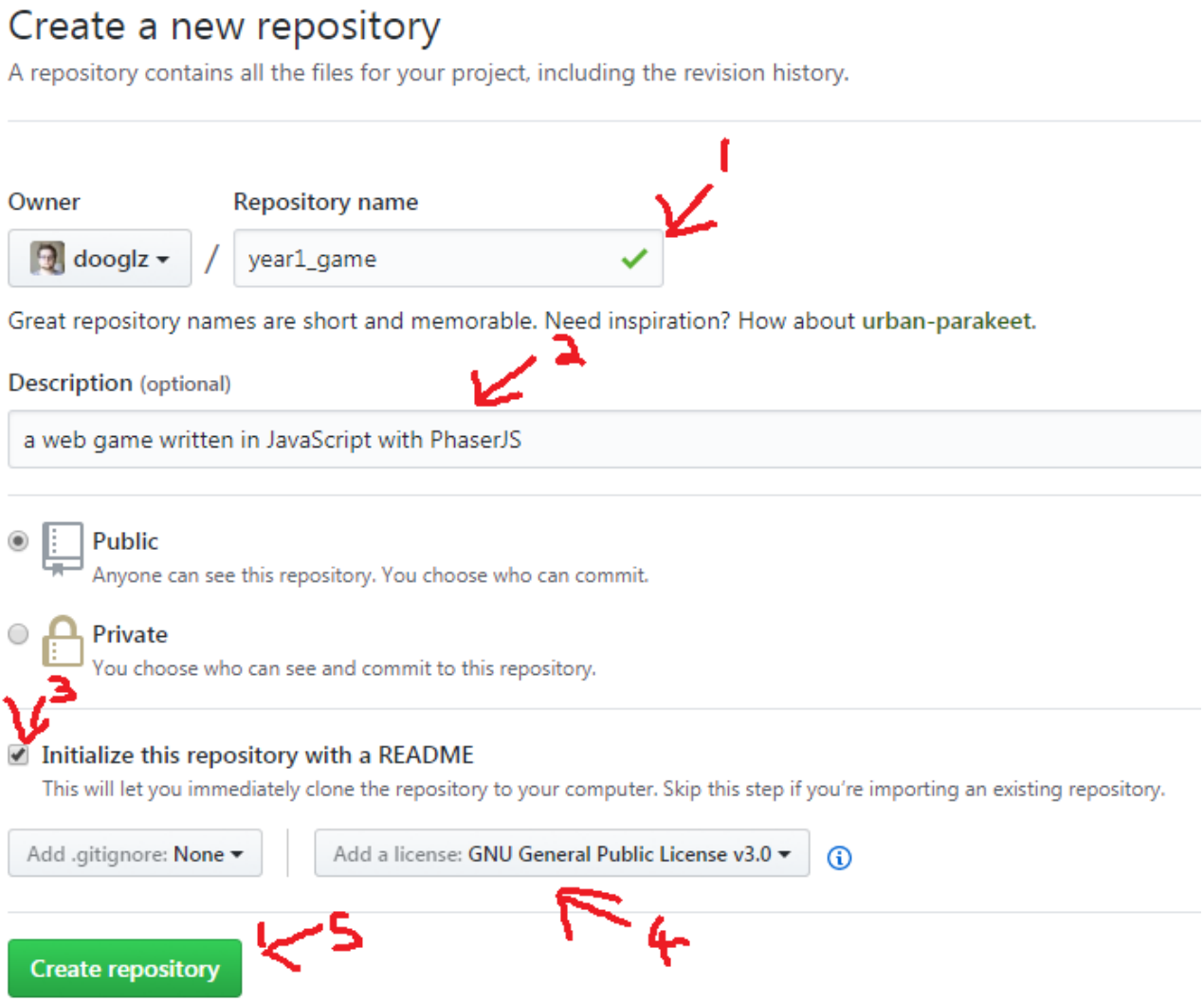


Figure 1.2: **New Repo Options**

Give your repo a simple name and descriptive description

Check - Initialize with a readme

Choose an open source license, so people cant legally steal your work without crediting you. The inbuilt guide from github covers this neatly, when in doubt: choose MIT license.

After this stage Github will create the repo for you and you should see something like the following image, now it's time to **clone** the repo down so you can start to work within it. Click the green clone button and copy the link within the box

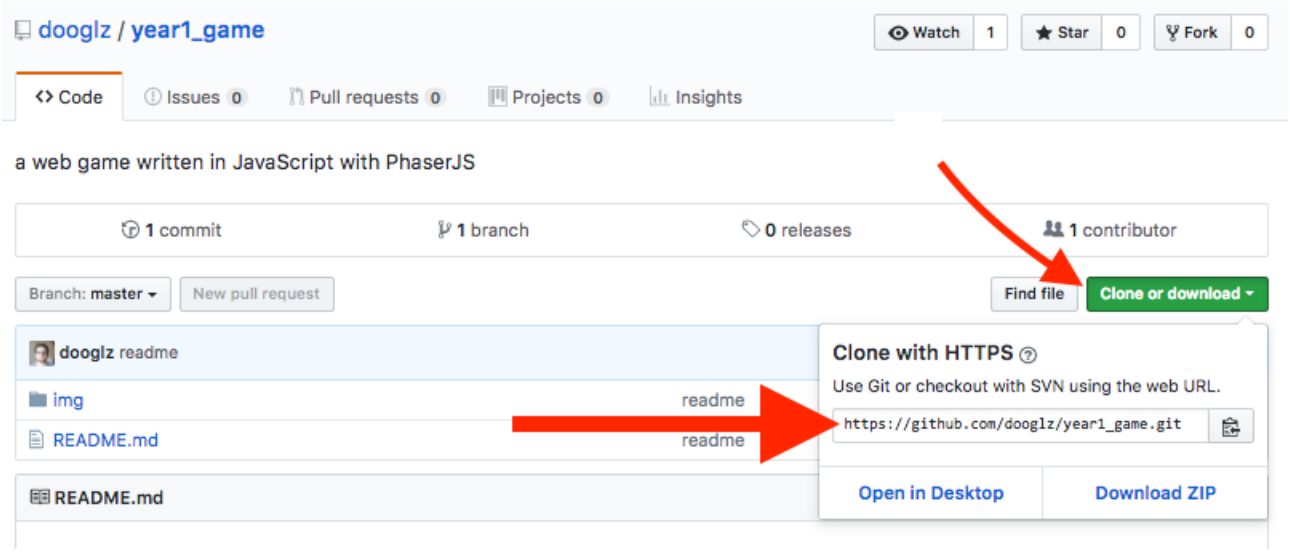


Figure 1.3: **Clone from Github**

1.1.2 Step 2 - Clone your repo

If you haven't installed Git on your pc yet, git-scm.com/downloads

Open a cmd (or git-bash) window somewhere (desktop is best). Now clone your repo down

```
1 git clone https://github.com/you/set09121_labs
```

This will create a folder named set09121_labs (or whatever you called you repo). Let's move into that folder in the terminal

```
1 cd set09121_labs
```

Now if you run

```
1 git status
```

You should see something similar to

```
1 On branch master
2 Your branch is up-to-date with 'origin/master'.
3 nothing to commit, working tree clean
```

Now we can Start to get to work properly.

1.1.3 Step 3 - Setup project structure

Create the following empty folders.

- **res** (where resources go, like images and fonts)
- **lib** (libraries that we need)
- **practical_1** (source code for practical 1)

Next, create a **.gitignore** file, open with a text editor. Navigate to www.gitignore.io and create an ignore file for "C++" and the IDE you intend to use, i.e "Visual Studio". Copy the generated file into your gitignore file and save it. Git will not look at or track any files that match the rules in the gitignore file, keeping junk you don't need out of your repo.

1.1.4 Step 4 - Get SFML

We will be building SFML from source, which means we need to get the code. We will be doing this via Git Submodules, which makes it look like the SFML code is now copied into your repo, but is actually saved a virtual link to the separate SFML repo.

```
git submodule add https://github.com/SFML/SFML.git lib/sfml
git submodule init
git submodule update
```

1.1.5 Step 5 - Write some test code

With a simple text editor, create a **main.cpp** file in the practical_1 folder, input the following code:

```
#include <SFML/Graphics.hpp>

int main(){
  sf::RenderWindow window(sf::VideoMode(200, 200), "SFML works!");
  sf::CircleShape shape(100.f);
  shape.setFillColor(sf::Color::Green);

  while (window.isOpen()){
      sf::Event event;
      while (window.pollEvent(event)){
      if (event.type == sf::Event::Closed){
        window.close();
      }
    }
    window.clear();
    window.draw(shape);
    window.display();
  }
  return 0;
}
```

Listing 1.1: practical_1/main.cpp

1.1.6 Step 6 - Create the CMake script

With a simple text editor, create a **CMakeLists.txt** file in the root folder, input the following code:

```
cmake_minimum_required(VERSION 3.9)
# Require modern C++
set(CMAKE_CXX_STANDARD 14)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

project(Games_Engineering)

#### Setup Directories ####
#Main output directory
SET(OUTPUT_DIRECTORY "${CMAKE_BINARY_DIR}/bin/")
# Ouput all DLLs from all libs into main build folder
SET (CMAKE_RUNTIME_OUTPUT_DIRECTORY ${OUTPUT_DIRECTORY})

#### Add External Dependencies ####
add_subdirectory("lib/sfml")
set(SFML_INCS "lib/sfml/include")
link_directories("${CMAKE_BINARY_DIR}/lib/sfml/lib")

#### Practical 1 ####
file(GLOB_RECURSE SOURCES practical_1/*.cpp practical_1/*.h)
add_executable(PRACTICAL_1 ${SOURCES})
target_include_directories(PRACTICAL_1  PRIVATE ${SFML_INCS})
target_link_libraries(PRACTICAL_1 sfml-graphics)
```

Listing 1.2: CMakeLists.txt

Creating the Solution, with CMake If you are unfamiliar with CMake; Follow this guide: https://github.com/edinburgh-napier/aux_guides/blob/master/cmake_guide.pdf

Remember to place the build folder outside of the set09121 folder. Preferably your desktop, NOT your H drive

NEVER Build from your H drive!

Or a memorystick / External HDD

The build folder will never contain work you need to save or commit. All code resides in the source directory.

Once configured and generated, you can open the .sln file in the build folder. You should not need to touch any solution or project settings form within Visual Studio. The solution is set up so you don't have to do much work yourself or even understand Visual Studio settings.

1.1.7 Step 7 - Run The solution

Cmake should have generated a solution project for you in your build folder, open it. Practical_1 should be available as a project within it. Compile and run it! You should see a green

circle.

1.1.8 Step 8 - Saving your work

You should take this opportunity to commit and push your work. If you know the basics of git, this is nothing new.

```
1 git status
```

Running git status should show you all the files you have modified so far. We need to "Stage" or "add" these files.

```
1 git add .
```

This is a shorthand to tell git that we want to commit everything.

```
1 git commit -m "SFML hello world working"
```

Now we run the actual commit, which will store the current version of all your ("Staged")files to the local repo. Note that this is only local, you now need ot push it up to github.

```
1 git push
```

This is a light-speed gloss over what version control can do for you. If this is new and strange to you, you really should take some time to look through some online git tutorials and guides to get comfortable with what it does and how it works.

1.1.9 Starting from scratch

If you want to work work on another pc, or at home. You obviously don't need to create a new repo. The steps you need to do are simply

1. Clone/Pull the repo down from your github
2. Get/update SFML by running:

   ```
   git submodule update --init --recursive
   ```

3. Run Cmake to generate your build folder

The key here is that you only need to version control your source folder, which only contains source files. The build folder, generated by CMake is full of large junk that visual studio needs, you don't need to save this or even really care about what's in there. You can re-generate it anytime anywhere using CMake.

Lesson 2

Pong

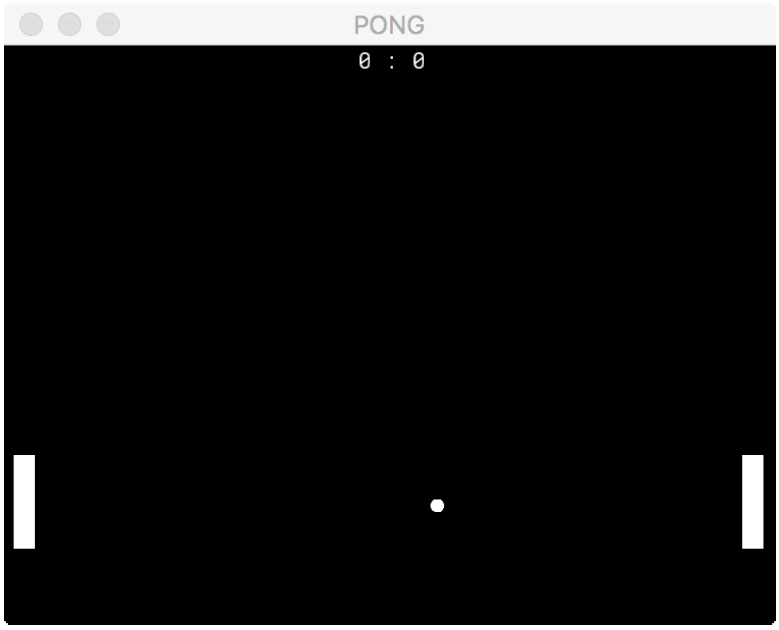


Figure 2.1: **Completed PONG**

For your first practical, we will be going back to the most basic of basic, Creating the classic arcade game PONG. For this we will be chucking some of the software practises you may be used to out the window. We will be using one single main.cpp file, no classes, no OO at all. You should be able to have a working pong game, with AI, in under 200 lines of code.

The purpose of this exercise is to get you acquainted with SFML. We will come back to the exercise often as an example of simple software design, as it useful as a base to compare more complex approaches.

Before we get stuck in, let's cover some of the fundamentals.

2.1 The game Loop

The fundamental core of all games, is the game loop. While some engines may hide this away, behind the scenes you can be sure that the code for any game can be stripped away to the fundamental game loop. It looks like this.

```
1  void Update(double dt) {
2    // Update Everything
3  }
4
5  void Render() {
6    // Draw Everything
7  }
8
9  int main () {
10   //initialise and load
11   while(!shouldQuit){
12     //Caluclate DT
13     Update(dt);
14     Render();
15     //Wait for Vsync
16   }
17   //Unload and shutdown
18 }
```

Listing 2.1: Standard Game Loop

The program starts as all programs do, with the main() function. Here we would load in anything we need at the start of the game, create the window, load settings, and all the usual start-up things you can imagine. If we weren't using SFML we would have to detect the capabilities of the system and enable or disable certain code paths accordingly (or throw an error and quit). SFML does all this work for us, and so we only need to care about loading things directly relevant to our game.

From there we enter a while loop of some kind. This will be based on some boolean value that determines if the game should quit or not. The game logic will set this from true to false when some event has happened (e.g, ESC key pressed, or player presses quit button). This loop will continuously run two functions, Update and Render. Update is where all game-logic code will go. This includes input processing, physics, content loading/unloading, networking.. etc. The rate at which we loop through this loop is the games framerate.

Once the Game update has been completed, the game can render a frame. No rendering will take place during the update function. The simple way of thinking is that the Update function determines where everything is and what it's doing. The render function then draws the results of the update.

Before calling Update, the Delta-Time (DT) is calculated. This is the amount of time that has passed between now and the previous frame. With a game updating at a steady 60fps, dt should be approximately 16ms (1/60). To actually calculate DT, you can use inbuilt C++ timers, or just use the handy SFML Clock.

```
1 static sf::Clock clock;
2 const float dt = clock.restart().asSeconds();
```

Listing 2.2: SFML DT

2.1.1 Update(dt)

This is also commonly called the game "Tick". While in our games we only do one "Tick" per frame, we could do more. If the game's logic could be executed quickly, and the game relies on fast action, it may be beneficial to do as many updates as you can between frames. While we aren't going to implement this, what you should take away is that: The Update function should be decoupled from Render(), so that multiple calls to Update() before a call to Render() should not break your game.

Framerate Independence The use of DT is to allow time related logic to take place (i.e a countdown timer) accurately, and for movement and physics code to work properly - independent of the framerate. A problem that arises with games is that the frametime can vary, sometime not by much, sometimes drastically (lower end PCs). Ideally we want a solid 16ms, but we won't always get that. A stuttering game doesn't look good graphically, but if we don't account for it within the game-logic, it can actually cause major problems. Imagine a player moving left at a speed of 10 units a second. With an ideal frame-rate of 60fps. We are calling Update() 60 times per second. So we should move the player by 0.16 (10/60) units each update.

```
1 const float playerSpeed = 10.f;
2 const float idealFps = 60.f;
3
4 void Update(double dt) {
5  player.move(playerSpeed / idealFps);
6 }
```

Listing 2.3: Shonky movement code

This would work, but if the frametime suddenly drops to 30fps, or starts varying between 30 or 60, the update() function will be called less, and so the player will move slower. Conversely, if the fps skyrockets, then our player will start moving at light-speed. The solutions is to **Always include DT in calculations that involve time. i.e speed, movement, acceleration.** Here is the proper way to do it:

```
1 const float playerSpeed = 10.f;
2
3 void Update(double dt) {
4   player.move(playerSpeed * dt);
5 }
```

Listing 2.4: Framerate Independent movement code

Let's look at the maths here.

- when the game is running slower – DT gets larger (it's the time between frames).
- When the game is running faster – DT gets smaller.
- When the game is running at our ideal rate of 60fps – DT = 60.
- If the game drops down to 30fps – DT = 120
- When the game is running slower – the player moves faster.
- When the game is running faster – the player moves slower.

2.1.2 Render()

The render function does what you would expect, renders everything in the game to screen. There may be additional logic that goes on to do with optimisation and sending things back and forth between the GPU, but we are not dealing with any of this for a @D game in SFML. To us, the render function is simply where we tell SFML to draw to the screen. In a multi-threaded engine, this could be happening alongside an update function (more on this later).

Vsync One other piece of logic that is important the game loop is the "buffer-swap" or "swap-chain" or "Vsync". This is a function that we can call that let's us know that the rendering has finished, and therefore it's the end of the frame, and time to start a new one.

```
1 //End the frame - send to monitor - do this every frame
2 window.display();
3
4 // enable or disable vsync -
5 // do at start of game, or via options menu
6 window.setVerticalSyncEnabled(true/false);
```

Listing 2.5: Swapchain in SFML

If we have enabled Vertical-sync(Vsync), the game will limit itself to the refresh rate of the connected monitor (usually 60hz, so 60fps). In this scenario once we have finished rendering everything in under 16ms, and we call window.display(), the game will wait for the remaining time of the frame before continuing. This is a carry-over from low level graphics programming where you don't want to send a new image to the monitor before it has finished drawing the previous (this causes visual Tearing). So if we are rendering faster than 60fps, the game will wait at the end of the render function while the monitor catches up. Before starting again the next frame. With vsync disabled, once we have finished rendering a frame, window.display() does not pause after sending the image and we continue to render the next frame immediately.

An important gotcha that can happen here is that the graphics drivers can manually override and forcefully enable ro disable Vsync. So don't depend on it always being in the state that you set it.

2.2 Starting PONG

I'll get you started off with the top of your file. It's the usual imports and namespaces, followed by some variables we will use for game rules, and then 3 shapes, 1 circle for the ball, and 2 rectangles stored in an array for the paddles.

We then move onto the Load() function, we would load in assets here if we had any, and then we set-up the game by resizing and moving our shapes.

Line 32 and 34 are left incomplete for you to complete later. We can come back to it after looking at our other functions.

```
1  #include <SFML/Graphics.hpp>
2
3  using namespace sf;
4  using namespace std;
5
6  const Keyboard::Key controls[4] = {
7      Keyboard::A,    // Player1 UP
8      Keyboard::Z,    // Player1 Down
9      Keyboard::Up,   // Player2 UP
10     Keyboard::Down // Player2 Down
11 };
12 const Vector2f paddleSize(25.f, 100.f);
13 const float ballRadius = 10.f;
14 const int gameWidth = 800;
15 const int gameHeight = 600;
16 const float paddleSpeed = 400.f;
17
18 CircleShape ball;
19 RectangleShape paddles[2];
20
21 void Load() {
22   // Set size and origin of paddles
23   for (auto &p : paddles) {
24     p.setSize(paddleSize - Vector2f(3, 3));
25     p.setOrigin(paddleSize / 2.f);
26   }
27   // Set size and origin of ball
28   ball.setRadius(ballRadius - 3);
29   ball.setOrigin(ballRadius / 2, ballRadius / 2);
30   // reset paddle position
31   paddles[0].setPosition(10 + paddleSize.x / 2, gameHeight / 2);
32   paddles[1].setPosition(...);
33   // reset Ball Position
34   ball.setPosition(...);
35 }
```

Listing 2.6: practical_1/main.cpp

The Update Here – as we have covered previously – is where our gamelogic goes. This runs every frame. Firstly we calculate DT, then process any events that sfml passes to us. From there we are free to do whatever we want, and what we want to do is make PONG. We will come back and add to this, you don't need to edit anything just now

```
36 void Update(RenderWindow &window) {
37   // Reset clock, recalculate deltatime
38   static Clock clock;
39   float dt = clock.restart().asSeconds();
40   // check and consume events
41   Event event;
42   while (window.pollEvent(event)) {
43     if (event.type == Event::Closed) {
44       window.close();
45       return;
46     }
47   }
48
49   // Quit Via ESC Key
50   if (Keyboard::isKeyPressed(Keyboard::Escape)) {
51     window.close();
52   }
53
54   // handle paddle movement
55   float direction = 0.0f;
56   if (Keyboard::isKeyPressed(controls[0])) {
57     direction--;
58   }
59   if (Keyboard::isKeyPressed(controls[1])) {
60     direction++;
61   }
62   paddles[0].move(0, direction * paddleSpeed * dt);
63 }
```

Listing 2.7: practical_1/main.cpp

Render and Main Our last section of the file is out super simple render function. I mean, just look at it. Isn't SFML awesome? Then we have our standard Main entrypoint, with our gameloop code. Not much to see here.

```
64 void Render(RenderWindow &window) {
65   // Draw Everything
66   window.draw(paddles[0]);
67   window.draw(paddles[1]);
68   window.draw(ball);
69 }
70
71 int main() {
72   RenderWindow window(VideoMode(gameWidth, gameHeight), "PONG");
73   Load();
74   while (window.isOpen()) {
75     window.clear();
76     Update(window);
77     Render(window);
78     window.display();
79   }
80   return 0;
81 }
```

Listing 2.8: practical_1/main.cpp

Once you've typed all of the above in: you should have something like this:

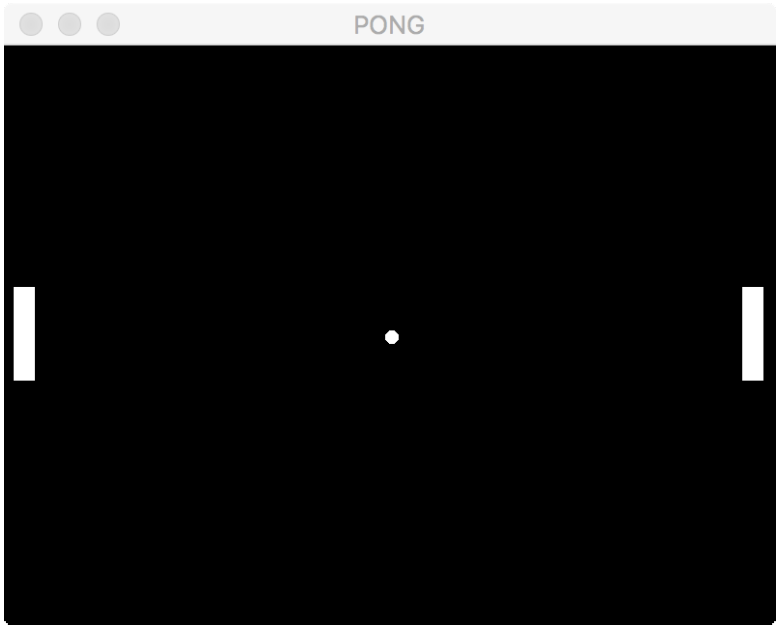


Figure 2.2: **PONG hello world**

2.2.1 Moving the ball

Add the following to the declarations at the start fo the file.

```
1 Vector2f ballVelocity;
2 bool server = false;
```

Add this to the Load() function

```
1 ballVelocity = {(server ? 100.0f : -100.0f), 60.0f};
```

Add this to the Update() function

```
1 ball.move(ballVelocity * dt);
```

What we have done here is store a 2D vector of the **velocity**(speed+direction) of the ball. SFML stores the **position** of the ball internally so we don't need an extra variable for that. In our update function we then move the ball by it's velocity. If we were coding the move function manually it would look like this

$$newBallPosition = oldBallPosition + (ballVelocity * TimePassed)$$

Which is just simple physics. I've introduced another piece of logic, the 'server' boolean. This decided which direction the ball starts moving at the start of the game. The weird piece of code that you added to the load function is an 'inline IF statement'. It's just the same as a IF block, but in one line. If the statement before the ? is true, then ballVelocity.x is set to -100, if server is false, ballVelocity.x = 60.0f.

2.2.2 Ball collision

"Bouncing" the ball is one of those cheap tricks that looks harder than it is. As we are storing the ball velocity as a vector. Negating it with respect to wall it bounced off of is as easy as multiplying the right element (X or Y) by -1. We actually multiply by -1.1, and 1.1 here so the ball not only bounces, but gets faster. All in two lines of code. Maths *is* fun.

In a perfect word that would be fine, but if the ball was flying super fast, in might get "stuck within the wall" where the collision code will constantly negate it's velocity, while speeding it up. This will cause the ball to stop, wiggle, then form a black hole of big numbers. This is bad. To get around this we chat and teleport the ball out of teh wall by 10 units. Pong is a fast paced game so no one will notice. (This foreshadows some of the nastiness that happens when trying to write good physics code, we will talk about much later)

Add the following to the Update(). There is no functional purpose for the bx and by variables, but as we are going to use them a lot it's nicer to have them to keep our code small.

```
1  // check ball collision
2  const float bx = ball.getPosition().x;
3  const float by = ball.getPosition().y;
4  if (by > gameHeight) {
5    // bottom wall
6    ballVelocity.x *= 1.1f;
7    ballVelocity.y *= -1.1f;
8    ball.move(0, -10);
9  } else if (by < 0) {
10   // top wall
11   ballVelocity.x *= 1.1f;
12   ballVelocity.y *= -1.1f;
13   ball.move(0, 10);
14 }
```

Left and right "Score walls" This is a very easy check. We will be extending on the previous section of code with more else if statements. What is new here is that if we have collided with these walls, then we don't bounce. We reset the balls and the paddles, and increment the score. Ignore score for now, but you should implement the reset function. You should pull out some of the logic form the load() function and then call Reset() at the beginning of the game. Repeated code is bad code.

```
1  else if (bx > gameWidth) {
2    // right wall
3    reset();
4  } else if (bx < 0) {
5    // left wall
6    reset();
7  }
```

Collision with paddles This is a simple "Circle - Rectangle" collision check. But as we know that the paddles only move in the Y axis, we can take some shortcuts, we only need to check if the ball is within the top and bottom edge of the paddle.

```
else if (
  //ball is inline or behind paddle
  bx < paddleSize.x &&
  //AND ball is below top edge of paddle
  by > paddles[0].getPosition().y - (paddleSize.y * 0.5) &&
  //AND ball is above bottom edge of paddle
  by < paddles[0].getPosition().y + (paddleSize.y * 0.5)
  ) {
    // bounce off left paddle
  } else if (...) {
    // bounce off right paddle
  }
```

2.2.3 Two Player and Validated moves

You should now extend your game logic code to allow for moving both paddles (use the controls array defined at the top of the file). This should simply be a case of adding two extra IF statements to the Update(). Further to this, players shouldn't be able to move off of the screen. Given you have already done collision code, this should be a simple addition for you to complete.

2.2.4 AI

At this stage, we want to keep the code simple, so for AI, the AI paddle will try to match it's position to be the same height as the ball. However, keep it fair, AI should always play by the same rules as the player. The AI paddles should move at the same speed as the player, and not teleport to the correct position.

I've not given you any code for this, you should try to implement this yourself.

2.2.5 Adding score text

Loading and displaying text is super easy with SFML. Firstly you will need to find a font.ttf file somewhere, and use the following code

Add the following to Load()

```
1  // Load font-face from res dir
2 font.loadFromFile("res/fonts/RobotoMono-Regular.ttf");
3 // Set text element to use font
4 text.setFont(font);
5 // set the character size to 24 pixels
6 text.setCharacterSize(24);
```

Add the following to your Reset()

```
1  // Update Score Text
2 text.setString(...);
3 // Keep Score Text Centered
4 text.setPosition((gameWidth * 0.5f) - (text.getLocalBounds().width *↩
      0.5f), 0);
```

Finally, you need to add to your Render() function. I'll let you figure out what.

Runtime Resources If you have written your code correctly, you will get an error during runtime, that it can't find the font file specified. You will need to have your font file relative to the "working directory" of your game. Traditionally this is wherever the .exe is (In the build folder), but IDEs can change this to be other places. There is a proper way to do this, and we will cover this in the next practical. For now you should investigate where your program looks for files and place the font there manually.

2.2.6 Done

At this point you should have a fully featured PONG game, feel free to add more features, but the point of this exercise is to make a simple game with as few lines as possible of simple code. It's refreshing to be able to work in these conditions, without having to think about software design and large scale functionality. The next project we will work though will get progressively more complex until we are building an entire games engine. So keep you simple PONG game around, as something to look back on fondly as a statement of how easy it can be sometime to make great games.

Lesson 3

Space Invaders

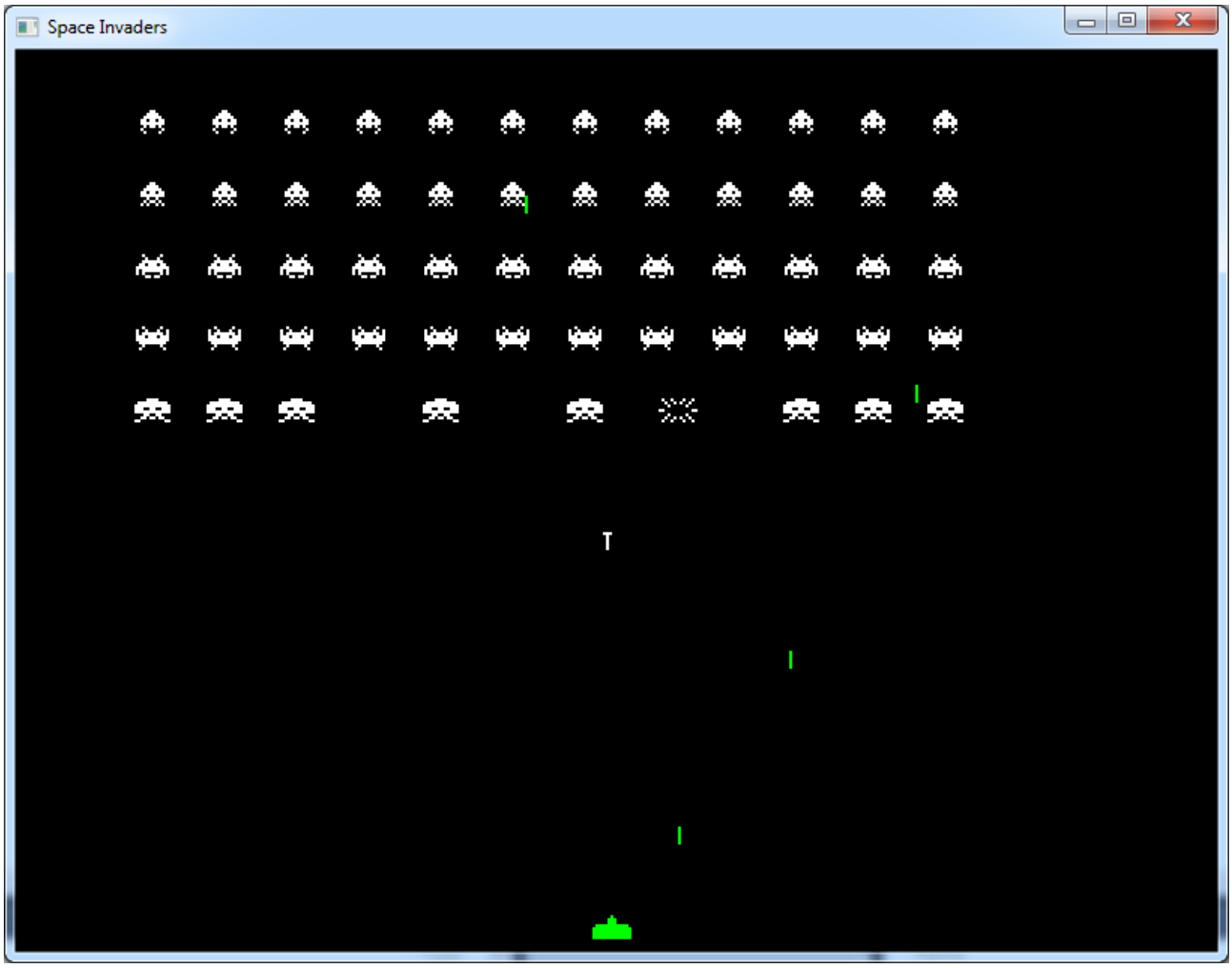


Figure 3.1: **Completed Space Invaders**

This lab is design to introduce you to: Object Orientation(OO) in C++, Working with C++ header files, and a small amount of memory and resource management.

3.1 Adding another project

We will be adding this lab a 'project' to our already existing 'Games Engineering solution'. Note: this is the vernacular of Visual studio. In CMake Terms we are adding another 'Executable' to the 'project'.

- Create a "practical_2" folder inside your repo.
- Within that, create a main.cpp, feel free to copy some boilerplate SFML code into it.

```
1 ## Space invaders
2 file(GLOB_RECURSE SOURCES practical_2/*.cpp practical_2/*.h)
3 add_executable(PRACTICAL_2_INVADERS ${SOURCES} )
4 target_include_directories(practical_2 SYSTEM PRIVATE ${SFML_INCS})
```

Listing 3.1: Addition to CMakeLists.txt

- Configure and generate from CMake.

Helpful hint: on Re-configuring CMake Whenever we alter the CMake script, or add/remove source files from the source repo, we must configure and generate from CMake again. There is a short-cut do doing this. In your open solution in visual studio, CMake builds a helper project called "ZERO_CHECK". Building this project runs a script to configure and regenerate in the background. So we can edit and rebuild the CMakelists.txt without leaving Visual studio.

A note on creating additional files As you know, we have all our source code in the source 'repo' folder, and all the project and build files in the ephemeral 'build' directory that CMake generates for us. CMake links the source files directly in the project files. When you edit a .cpp file in Visual Studio, it is editing the real file in the repo directory, even though all of visual studios files are in the 'build' directory.

One annoying caveat of this is that if you try to create a new file from visual studio, it incorrectly puts it in the build directory. You can manually type in the correct directory in the create file dialogue, or create the files manually and re-run CMake. Note: you will have to re-run CMake anyway when adding or removing files in the source directory.

3.2 Sprite-sheets

Before we get stuck in, you should have already have a standard boiler plate gameloop written to open a window and poll for events.

A common technique for 2D art assets in games is to combine what would be many separate images into one "sprite sheet". This saves time when loading in files, and a small amount of graphics memory. Images are places into tiles within a larger image, sprites are rendered by taking a 'cut out' of the larger image. In terms of optimisation this makes life very easy for the GPU – as it doesn't have to switch texture units.

The primary benefit of sprite sheets however is sprite-animation. It is commonplace to have each tile in the sheet be the same square size. With this restriction in place, picking the dimensions to 'cut' are a simple multiplication operation. Rendering a sequence of frames as an animation is as simple as moving the 'cut' windows to the right each frame.

Figure 3.2: **Animation Frames** Sonic will be rendered using different frames from the sprite-sheet each time, looping to the start once all frame have been rendered. Creating the illusion of fluid movement

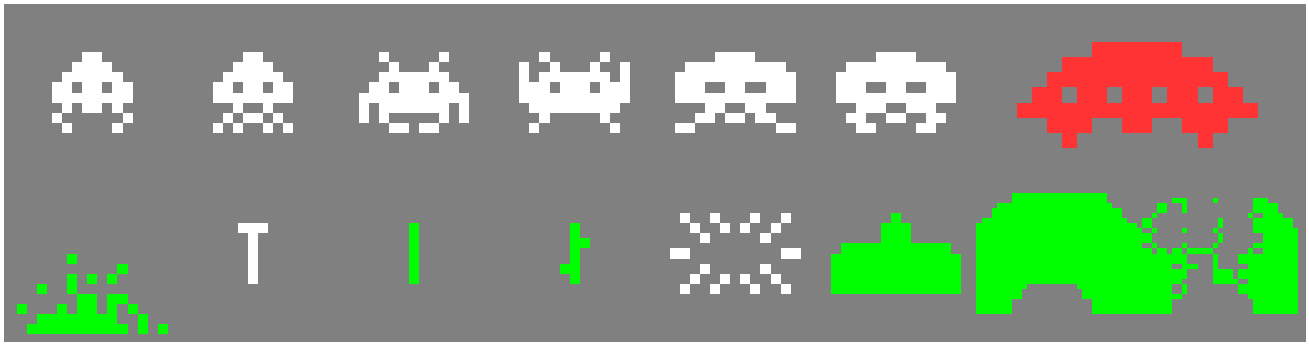


Figure 3.3: **Our Space Invaders Sprite-sheet** It's actually a transparent image - grey background just for clarity.

Figure 3.4: **Minecraft's Textures** Having all the textures in one image allows for easy modability. The multiple squares of water and lava are animation frames.

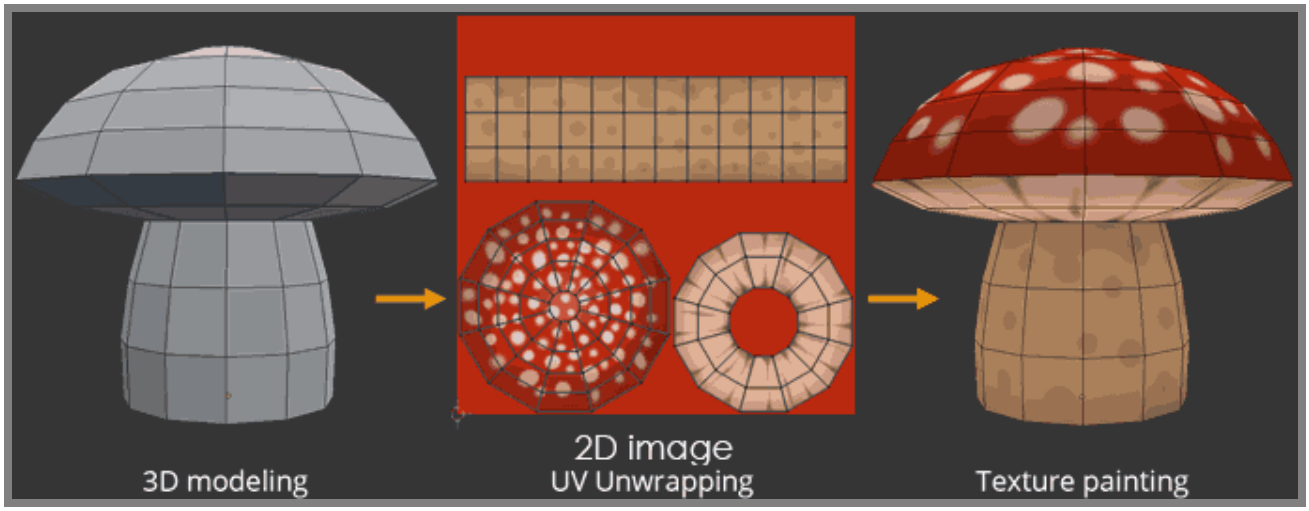


Figure 3.5: **Sprite sheets are not UV maps** Although visually similar - don't confuse sprite-sheets for unwrapped UV textures. These are used to apply a texture to a 3d model. Sprite-sheets are for 2d sprites

3.3 Loading a sprite-sheet

Workign with Sprite-sheets in SFML couldn't be easier. Take a look at this:

```
1  sf::Texture spritesheet;
2  sf::Sprite invader;
3
4
5  void Load() {
6    if (!spritesheet.loadFromFile("res/img/invaders_sheet.png")) {
7      cerr << "Failed to load spritesheet!" << std::endl;
8    }
9    invader.setTexture(spritesheet);
10   sprite.setTextureRect(sf::IntRect(0, 0, 32, 32));
11 }
12
13
14 void Render() {
15   window.draw(invader);
16 }
```

Listing 3.2: Sprite sheet code

Totally easy. Take note of this line:

```
1  sprite.setTextureRect(sf::IntRect(0, 0, 32, 32));
```

The rectangle structure takes the form of (Left, Top, Width, Height). Our sprite-sheet is dived into squares of32x32 pixels. So this line of code set the 'cut' do be the top left square in the image, aka. The first invader sprite.

Note that the invader doesn't take up the whole 32x32 square, it's surrounded by transparent pixels. SFML takes care of doing the rendering with correct modes so as to cuts out the background, but we may have to be careful when it comes to physics and collision code.

3.3.1 Runtime Resources Done Right

Before we can get this code running, we need to get the spritesheet image to somewhere where the running game can find it. In the previous practical we just dropped the image into the build folder, now we are going to do it properly.

Method 1 : Altering the working directory A commonly used practise for getting your runtime resources where you need them is to alter the debugging "Working Directory" settings in visual studio to point to the source folder. This runs your game out of the build directory, but it looks for files in the source directory.

This works great for development but it means you can't run your program outside of Visual studio, and you have to remember to copy all the required files form your source directory when it comes time to package up an installer.

There is an even better way.

Method 2 : Copy all resources via POST_BUILD Visual studio allows you to input simple command-line scripts to run after certain events are triggered, such as Before-Build and Post-Build. This means we can write a script to automatically copy everything in source/resources to correct build folder whenever we build the application. This would involve digging through Visual studio windows, but thankfully CMake can do this for us.

CMake code just an example, don't actually write this

```
add_custom_command(TARGET YOUR_EXECUTABLE POST_BUILD
  COMMAND ${CMAKE_COMMAND} -E copy_if_different "resource_dir"
    $<TARGET_FILE_DIR:${YOUR_EXECUTABLE}>/res
)
```

Listing 3.3: POST Build script

This is a pretty good solution, but come with ing downsides. Visual studio only triggers POST_BUILD on build events, not on RUN events. So if you change a resource file in any way, you will have to rebuild your game to get it to trigger the copy script. There is one final step we need to do to perfect our workflow:

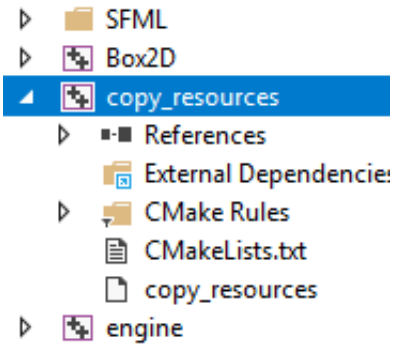
Method 3 : Copy all resources via Custom Target CMake can add more than just executables to the solution, it can also add "custom targets". We just aht below, and add a slightly modified version of our copy script within it.

```
add_custom_target(copy_resources ALL COMMAND ${CMAKE_COMMAND}
  -E copy_directory
    "${PROJECT_SOURCE_DIR}/res"
    ${CMAKE_RUNTIME_OUTPUT_DIRECTORY}/$<CONFIGURATION>/res
)
```

Listing 3.4: custom target script

[If you are on osx, see Appendix A.1.]

And there we have it, a project in Visual studio we can 'Build' at any time that will copy everything we need from the source directory to the build directory



Oh, I almost forgot, there's one last thing to do in our CMake:

```
add_dependencies(PRACTICAL_2_INVADERS copy_resources)
```

Listing 3.5: custom target dependancy

Now whenever we build Space_Invaders, Visual studio will make sure that 'copy_resources' has been run. Excellent.

3.4 Creating the Ship Class

It's been a long journey since we wrote some space invaders code, let's get back to it. As with all software projects, as the complexity of the software grows, so do the potential different ways to implement it. That is to say, this may not be the best way to implement space invaders, we like to think it's at least a 'good' way. The point of building it this way is to expose you to many different aspects of C++ OO. Any programmer worth their salt will have an opinion on how they could improve someone else's code, and if you feel at the end of this that you have some ideas, then this lesson was successful.

Let's go OO We are going to need at least two different entities for our invaders game. Invaders and the player. Invaders are all identical other than their starting position and sprite. They also exhibit some non-trivial individual logic. Your software engineering brain should be starting to form the basis for properties and methods of the invader class by this point. To add to the fun, consider the player, and how similar it is also to the invader. they both shoot bullets, move, and can explode. This sounds like inheritance should be joining this party.

The way we are going to go about this to have an *abstract base class* **Ship**, which is inherited from by a **Player** class and an **Invader** Class.

Functionality of the Ship The ship class will contain all logic that is common for both the player and invaders. Primarily this will be "moving around". We could go with the full entity model and have Ship be a base class, with variables for it's position and rotation and such. We would then also have a sf::Sprite member attached where we would call upon all the SFML render logic. This is a good idea – for a larger game. For space invaders that would involve lot's of code to keep the sprite in sync with the ship Entity. Instead we are going to take a super short cut, and inherit for sf::sprite.

This means that Ship will have all teh same methods as a sf::sprite, including all the usual 'SetPostition()' and 'move()' commands we have been using already. It also means we can pass a ship object directly to window.draw().

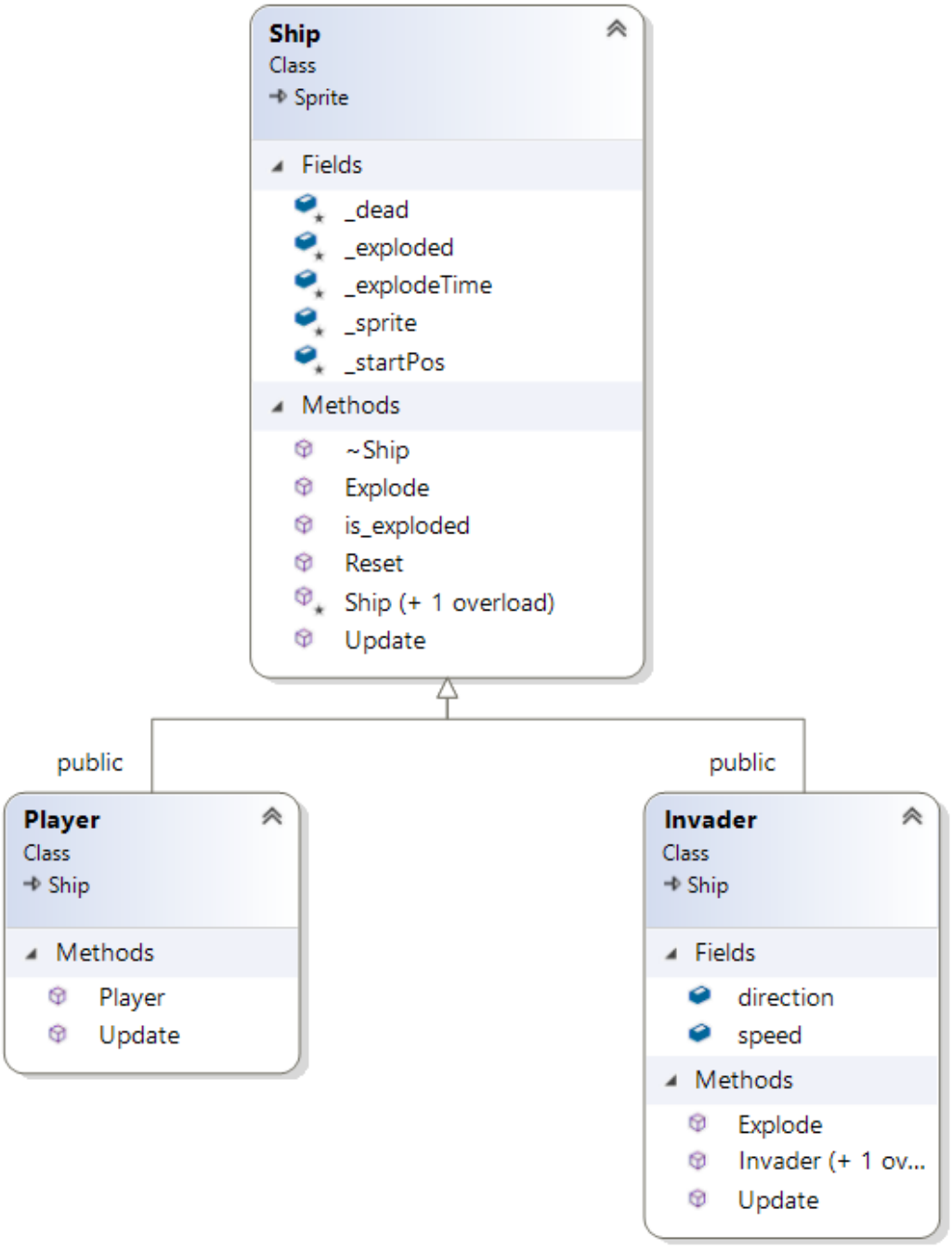


Figure 3.6: **Completed Space Invaders Class Diagram** don't worry about implmenting all these properties just yet

Create Ship.h Create a file inside the invaders source folder called "ship.h". This will be our Headder file for the Ship class. Headder files contain the declaration of our class, i.e only the function declarations. Headders shouldn't contain any code (some common exemptions apply). the reason we do this is to keep the logic of the class stored inside a .cpp file, any peice of code that want's to access this functionality only needs the headder. This concept does not exist inside java or C#, wherein you provide the full definition of a function inside a class in one file. the code runtime parses this and allows other classes to link to it. C++ is not so nice, and while this is totally possible to work in this fashion, we get into issues regarding name-space collisions, scope issues, multiple declarations, and code bloat. Ask in the lab if you would like to know more.

Anyway, Inside Ship.h get this written down:

```
1  #pragma once
2  #include <SFML/Graphics.hpp>
3
4  class Ship : public sf::Sprite {
5  protected:
6  sf::IntRect _sprite;
7  Ship();
8  public:
9
10 explicit Ship(sf::IntRect ir);
11
12 virtual ~Ship() = 0;
13
14 virtual void Update(const float &dt);
15 };
```

Listing 3.6: Initial Ship.h

Create Ship.cpp Next to our ship, create ship.cpp

```
1  #include "ship.h"
2
3  using namespace sf;
4  using namespace std;
5
6  Ship::Ship() {};
7
8  Ship::Ship(IntRect ir) : Sprite() {
9    _sprite = ir;
10   setTexture(spritesheet);
11   setTextureRect(_sprite);
12 };
13
14 void Ship::Update(const float &dt) {}
15
16 Ship::~Ship() = default;
```

Listing 3.7: Initial Ship.cpp

Access to global variables The code above needs access to some variables we have in our main.cpp (spritesheet). There are multiple ways to go about this. A rather simple yet hacky way is to have these variables as 'extern' in a headder file. To do this, create another headder, called "game.h" and insert:

```
#pragma once
#include <SFML/Graphics.hpp>
constexpr uint16_t gameWidth = 800;
constexpr uint16_t gameHeight = 600;
constexpr uint16_t invaders_rows = 5;
constexpr uint16_t invaders_columns = 12;

extern sf::Texture spritesheet;
```

Listing 3.8: game.h

We are defining some common variables here as constant, which is fine. The interesting bit is the 'extern spritesheet', this tells anyone that includes game.h that a sprite-sheet exists 'somewhere". That somewhere is main.cpp, and the compiler will figure this out for us when we need to access it from ship.cpp.

Remember to reload CMake via Zero_check to add our new files to the build

Test out the code As Ship is an abstract class, we can't create one. Ee can only create a concrete class derived from it. We can reference it as pointer however, due to how c++ polymorphism works. Add the following to the top of your main.cpp

```
#include "ship.h"
//...
std::vector<Ship *> ships;
```

Listing 3.9: main.cpp

This should compile without errors.

3.5 Making the Invader class

We could create a new invader.h and invader .cpp to house the invader class. Generally speaking separate files for separate classes is a good idea, although unlike Java we don't *have* to do this. In some situations when certain classes are very similar or just slightly different version of each other it makes sense to host them in the same Headder file.

Add the following to code

```
class Invader : public Ship {
public:
  Invader(sf::IntRect ir, sf::Vector2f pos);
  Invader();
  void Update(const float &dt) override;
};
```

Listing 3.10: Ship.h with inital Invader class

```
1  Invader::Invader() : Ship() {}
2
3  Invader::Invader(sf::IntRect ir, sf::Vector2f pos) : Ship(ir) {
4      setOrigin(16, 16);
5    setPosition(pos);
6  }
7
8  void Invader::Update(const float &dt) {
9    Ship::Update(dt);
10 }
```

Listing 3.11: Ship.cpp with inital Invader class

Now that we have a concrete implementation of a Ship we can create one.

```
1 Load(){...
2 Invader* inv = new Invader(sf::IntRect(0, 0, 32, 32), {100,100});
3 ships.push_back(inv);
```

Listing 3.12: main.cpp

Important note, we used the New() operator here, which created the ship on the heap. If we wanted a stack version, we omit the new New and would use Invader 'inv = Invader()'.

As we are storing the invader into a vector of ships, which will also later contain the player, this vector must be Ship Pointers. The way we have set this up we couldn't even create a vector<ship>, as that would try to construct an abstract class.

We need to call the update function of all our ships every frame, due to polymorphism this is very simple. Update() is a virtual function so when we call update() on a ship pointer that points to an invader, the invader's update() is called.

```
1 Update(){...
2   for (auto &s : ships) {
3     s->Update(dt);
4   };
```

Listing 3.13: main.cpp

The same goes for rendering, as we have inherited from sprite, SFML can render our ships natively.

```
1 Render() {...
2   for (const auto s : ships) {
3     window.draw(*s);
4   }
```

Listing 3.14: main.cpp

3.5.1 Invader movement

A quirk of space invaders is that all the invaders move as one, when any of the invaders touches the edge of the screen: all invaders drop down and reverse direction. When invaders are killed, the remaining invaders speed up. From this we can gather that we need some form of

communication medium between all the invaders so they can communicate when it's time to drop down and when to speed up. We are going to store these parameters as two variables: direction and speed. We could store these as properties in each invader, but as the contents will be identical for each invader we should do something better. The "something better" is static properties.

```
1 class Invader : public Ship {
2 public:
3   static bool direction;
4   static float speed;
5   ...
```

Listing 3.15: Ship.h

Top Hint: Any declared static variable **must** be defined somewhere. Which means we do the following in ship.cpp

```
1 bool Invader::direction;
2 float Invader::speed;
```

Listing 3.16: Ship.cpp

We can access these variables anywhere like so 'invader::speed = 20.f'.

Invader Update It's about time we had something moving on screen. We should modify the Invaders Update() to include some movement code.

```
1  void Invader::Update(const float &dt) {
2    Ship::Update(dt);
3
4    move(dt * (direction ? 1.0f : -1.0f) * speed, 0);
5
6    if ((direction && getPosition().x > gameWidth - 16) ||
7      (!direction && getPosition().x < 16)) {
8        direction = !direction;
9          for (int i = 0; i < ships.size(); ++i) {
10           ships[i]->move(0, 24);
11         }
12   }
13 }
```

Listing 3.17: Ship.h

The first two lines are simple, we call the base ship::update() to run any logic that is generic for all ships (none right now). Then we move either left or right, at the speed dictated by the static speed variable. The next few lines of code is the logic to detect weather it's time to drop and reverse. Direction is involved in the check to stop a feedback loop occurring of one invader triggering the reverse, then in the same frame another invader re-reversing it. So long as the invaders are updated sequentially (i.e not in threads) then this will work.

3.5.2 Spawning Invaders

Now we need to see this in action, lets create some more invaders. I'll start you off with this hint:

```
Load(){...
    for (int r = 0; r < invaders_rows; ++r) {
  auto rect = IntRect(...);
  for (int c = 0; c < invaders_columns; ++c) {
    Vector2f position = ...;
    auto inv = new Invader(rect, position);
    ships.push_back(inv);
  }
 }
```

Listing 3.18: main.cpp

3.6 The Player Class

Compared to the invader, the player is actually a very simple class. The only real logic it brings to the party is moving left and right based on keyboard inputs. Let's get to it by adding to the ship.h file

```
class Player : public Ship {
public:
  Player();
  void Update(const float &dt) override;
};
```

Listing 3.19: ship.h

Pretty basic. Over in the ship.cpp we now define this code.

```
Player::Player() : Ship(IntRect(160, 32, 32, 32)) {
  setPosition({gameHeight * .5f, gameHeight - 32.f});
}

void Player::Update(const float &dt) {
  Ship::Update(dt);
  //Move left
  ...
  //Move Right
  ..
}
```

Listing 3.20: ship.cpp

You should know how to add in the movement code, it's almost identical to pong. Bonus points for not allowing it to move off-screen. You should construct one player at load time and add it to the vector of ships.

3.7 Bullets

The game wouldn't be very difficult (or possible) without bullets firing around. Let's look at our requirements:

- Invaders shoot green bullets downwards
- The player shoots white bullets upwards
- The bullets explode any ship they touch
- After exploding the bullets disappear

If we look at our sprite-sheet we have two different bullet sprites. SFML can do some colour replacement, so if we wanted we could use the same white sprites for both bullet types and get SFML to colour tehm differently. However we want the two bullet types to look physically different so we will use two different sprites. So for wahtevr we decide to go with for our software design, we will be inheriting from sf::sprite again.

That's rendering out of the way, now just movement and explosions to figure out. It would be tempting to do as we did with Ship and have two sublcasses for invader and player bullets. But as they are both so similar, the only difference being the direction they travel and who they collide with, having a three class structure would be overkill. Sometimes rigorously following OO patterns isn't the best way forward. So instead we will build just one bullet class.

Here we go, create a bullet.h and bullet.cpp

```
1  #pragma once
2  #include <SFML/Graphics.hpp>
3
4  class Bullet : public sf::Sprite {
5  public:
6    void Update(const float &dt);
7    Bullet(const sf::Vector2f &pos, const bool mode);
8    ~Bullet()=default;
9  protected:
10   Bullet();
11   //false=player bullet, true=Enemy bullet
12   bool _mode;
13 };
```

Listing 3.21: bullet.h

```
1  #include "bullet.h"
2  #include "game.h"
3  using namespace sf;
4  using namespace std;
5
6  //Create definition for the constructor
7  //...
8
9  void Bullet::Update(const float &dt) {
10     move(0, dt * 200.0f * (_mode ? 1.0f : -1.0f));
11 }
```

Listing 3.22: bullet.cpp

3.7.1 Firing bullets

Alrighty, that's a barebones Bullet created, now to spawn one. The simplest way to do this would be to do something like this:

```
1 Player::Update(){...
2 if (Keyboard::isKeyPressed(...)) {
3   new Bullet(getPosition(), false);
4 }
```

Listing 3.23: ship.cpp player update

This is not a good idea however, there is three major problems.

Firstly, we will spawn thousands of bullets, we need a way to 'cooldown' the weapon of the player
Secondly, How do we update and render them? We should store our bullets somewhere.
Thirdly, and this is a big one, we put these bullets on the heap, and then forget about them, ayy'oh that's a **Memory Leak**.

3.7.2 Storing our bullets

Let's get our bullets at least updating and rendering before we worry about the other issues. What we could do is something like this:

```
1 Player::Update(){...
2   static vector<bullet*> bullets;
3   if (Keyboard::isKeyPressed(...)) {
4     bullets.push_back(new Bullet(getPosition(), false));
5   }
6   for (const auto s : bullets) {
7     bullets.Update(dt);
8   }
```

Listing 3.24: ship.cpp player update

So now the player is responsible for handling and keeping track of all the bullets it fires. We've stopped a runaway memory leak (for now, we still have to delete the bullets later). But how do we Render them? and while we're asking questions, won't this very get really, really big? We are going to fire a lot of bullets.

Yes, we will be firing loads of bullets, and we don't want to have keep track them all and delete them. In larger games this would cause performance issues. It won't here, but let's pretend we are running on original 80's hardware, we've gotta do better, or else the arcade will have to shut-down and the kids will be sad.

A different solution - Bullet Pools How about instead of creating bullets as and when we need them, we allocate a whole bunch at the start, and put them into a "pool of available bullets". When a player or an invader fires, an inactive bullet in the pool gets initialised to the correct position and mode and goes about it's bullet'y business. After this bullet has exploded or moved off-screen, it is moved back into the pool (or just set to "inactive").

This is a very common technique used in games with lots of expensive things coming into and out of existence. Almost every AAA UnityD game uses this with GameObjects – which take forever to allocate and construct. It's much quicker to allocate loads at the start and re-use them.

Storing the Bullet Pool Each Ship could have it's own pool of 3 or 4 bullets to re-use, but that's a lot of code to refactor. Instead let's store the bullet pool *inside* the bullet class.

```
1
2 class Bullet : public sf::Sprite {
3 ...
4 protected:
5 static unsigned char bulletPointer;
6 static Bullet bullets[256];
7 ...
```

Listing 3.25: bullet.h - "Yo dog I put a bullet inside your bullet"

We have statically allocated 256 bullets on the stack. We have brought along a sneaky unsigned char to do a clever trick to determine which bullet to use next. Unisgned chars go between 0 and 255, and then wrap round back to 0 and repeat. Therefore every time we Fire() a bullet we choose from the array like this "bullets[++bulletPointer]". If there were ever more than 256 bullets on screen we will run into trouble, but I won't worry about this if you don't.

We will need to change our Firing mechanism, we now don't want to ever construct a bullet, just Fire() one. We will have to change our class declaration around to suit this. Fire() will become a static function.

```
1  class Bullet : public sf::Sprite {
2  public:
3    //updates All bullets
4    static void Update(const float &dt);
5    //Render's All bullets
6    static void Render(sf::RenderWindow &window);
7    //Chose an inactive bullet and use it.
8    static Fire(const sf::Vector2f &pos, const bool mode);
9
10   ~Bullet()=default;
11 Protected:
12   static unsigned char bulletPointer;
13   static Bullet bullets[256];
14   //Called by the static Update()
15   void _Update(const float &dt);
16   //Never called by our code
17   Bullet();
18   //false=player bullet, true=Enemy bullet
19   bool _mode;
20 };
```

Listing 3.26: bullet.h

I'll let you figure out the changes to the bullet.cpp. Keep in mind the differences between static-and non static functions. The _update() function is given in the next section.

3.8 Exploding Things

We will be using SFML to do the collision checks for us this time.

```
1  void Bullet::_Update(const float &dt) {
2      if (getPosition().y < -32 || getPosition().y > gameHeight + 32) ↩
           {
3          //off screen - do nothing
4          return;
5      } else {
6          move(0, dt * 200.0f * (_mode ? 1.0f : -1.0f));
7          const FloatRect boundingBox = getGlobalBounds();
8
9          for (auto s : ships) {
10             if (!_mode && s == player) {
11                 //player bulelts don't collide with player
12                 continue;
13             }
14             if (_mode && s != player) {
15                 //invader bullets don't collide with other invaders
16                 continue;
17             }
18             if (!s->is_exploded() &&
19                 s->getGlobalBounds().intersects(boundingBox)) {
20                   //Explode the ship
21                   s->Explode();
22                   //warp bullet off-screen
23                   setPosition(-100, -100);
24                   return;
25             }
26         }
27     }
28 };
```

Listing 3.27: bullet.cpp

This code will require modification to our ship class. Also we need access to a pointer to the player ship so we can determine the types of collisions. My way of doing this would be to add it as another extern in game.h.

We need to introduce Explode behaviour into the ship classes. We will add the common functionality to the base Ship class - turning into the explosion sprite. The invader class will extend this by increasing the speed of other invaders, add removing the explosion sprite after a second. The player ship will end the game if explode is called on it. Which will trigger a game reset.

```
class Ship : public sf::Sprite {
    ...
    protected:
      ...
      bool _exploded;
    public:
      ...
      bool is_exploded() const;
      virtual void Explode();
};
```

Listing 3.28: ship.h

```
void Ship::Explode() {
    setTextureRect(IntRect(128, 32, 32, 32));
    _exploded = true;
}
```

Listing 3.29: bullet.cpp

3.9 Bullet Timing and Explosion fade

We noticed earlier that there is no limit to how fast a player could shoot – let's remedy that now.

My favourite way of doing this is keeping a 'cooldown' timer.

```
void Player::Update(const float &dt) {
  ...
  static float firetime = 0.0f;
  firetime -= dt;
  ...
  if (firetime <= 0 && Keyboard::isKeyPressed(controls[2])) {
      Bullet::fire(getPosition(), false);
      firetime = 0.7f;
    }
}
```

Listing 3.30: ship.cpp

Every time we fire, we put the timer up by .7 seconds. Every Update() we reduce this by dt. This means that we should only be able to fire from the player every .7 seconds. You could image some form of power-up that would modify this timer.

3.9.1 Invader shooting

Invaders should shoot somewhat randomly. How you do this is up to you, when it comes to something like this, it's usually a case of trying out magic numbers until you find something that looks good.

My hacky / beautiful solution was this:

```
1  void Invader::Update(const float &dt) {
2    ...
3    static float firetime = 0.0f;
4    firetime -= dt;
5    ...
6    f (firetime <= 0 && rand() % 100 == 0) {
7        Bullet::fire(getPosition(), true);
8        firetime = 4.0f + (rand() % 60);
9    }
10 }
```

Listing 3.31: ship.cpp

I've limited so an invader won't be able to fire more than once in four seconds. Bonus points for coming up with a solution wherein only the bottom row of invaders shoot, and as the invaders numbers dwindle, they fire more often.

3.9.2 Fading the Explosion sprite

St the moment our implementation turns a invader into the explosion sprite when it explodes, an the explosion remains in space. We need it to fade out over time. To do ths we will use a similar technique as the bullet timer, where we will have a cooldown timer that starts when the ship explodes, and once the timer hit's 0, the invader is moved off the screen or turned invisible.

3.10 Future

- Game over screen
- Restarting the Game
- animated sprites
- Green shield bases

Lesson 4

Tile Engine

Lesson 5

Pacman

Lesson 6

Physics

Lesson 7

AI: Steering and Pathfinding

Lesson 8

AI: Behaviours

Lesson 9

Deployment and Testing

Lesson 10

Performance Optimisation

Lesson 11

Scripting

Lesson 12

Networking

Appendix A

Appendix

A.1 Additional CMake scripts

Custom target dependancy - for mac mac, sfml , and cmake do weird things to do with frameworks and R-paths.

```
#### Target for copying resources to build dir####
if(APPLE)
  add_custom_target(copy_resources ALL
    COMMAND ${CMAKE_COMMAND} -E copy_directory
    "${PROJECT_SOURCE_DIR}/res"
    ${CMAKE_RUNTIME_OUTPUT_DIRECTORY}/res

    COMMAND ${CMAKE_COMMAND} -E copy_directory
    "${CMAKE_SOURCE_DIR}/lib/sfml/extlibs/libs-osx/Frameworks"
    ${CMAKE_BINARY_DIR}/lib/sfml/Frameworks
  )
else()
  add_custom_target(copy_resources ALL COMMAND ${CMAKE_COMMAND} -E
  copy_directory
    "${PROJECT_SOURCE_DIR}/res"
    ${CMAKE_RUNTIME_OUTPUT_DIRECTORY}/$<CONFIGURATION>/res
  )
endif()
```

Listing A.1: custom target dependancy (including mac)