



SET09121 Games Engineering

Workbook 2018

Dr Kevin Chalmers
Sam Serrels

Edinburgh Napier University

January 15, 2018

Version 0.92

Contents

0.0.1	Git Repos	5
0.0.2	Development Environment	5
0.0.3	Workbook Style	6
0.1	Getting started for real	6
0.1.1	Step 1 - Create a Github Repo for your project	6
0.1.2	Step 2 - Clone your repo	8
0.1.3	Step 3 - Setup project structure	9
0.1.4	Step 4 - Get SFML	9
0.1.5	Step 5 - Write some test code	9
0.1.6	Step 6 - Create the CMake script	10
0.1.7	Step 7 - Run The solution	10
0.1.8	Step 8 - Saving your work	11
0.1.9	Starting from scratch	11
	Page	
1	Pong	12
1.1	The game Loop	13
1.1.1	Update(dt)	14
1.1.2	Render()	15
1.2	Starting PONG	16
1.2.1	Moving the ball	19
1.2.2	Ball collision	20
1.2.3	Two Player and Validated moves	21
1.2.4	AI	21
1.2.5	Adding score text	22
1.2.6	Done	22
2	Space Invaders	23
2.1	Adding another project	24

List of Figures

1	New Repo process	7
2	New Repo Options	7
3	Clone from Github	8
1.1	Completed PONG	12
1.2	PONG hello world	18
2.1	Completed Space Invaders	23

List of Algorithms

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. In 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

0.0.1 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.

0.0.2 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.

0.0.3 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.

0.1 Getting started for real

0.1.1 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>)

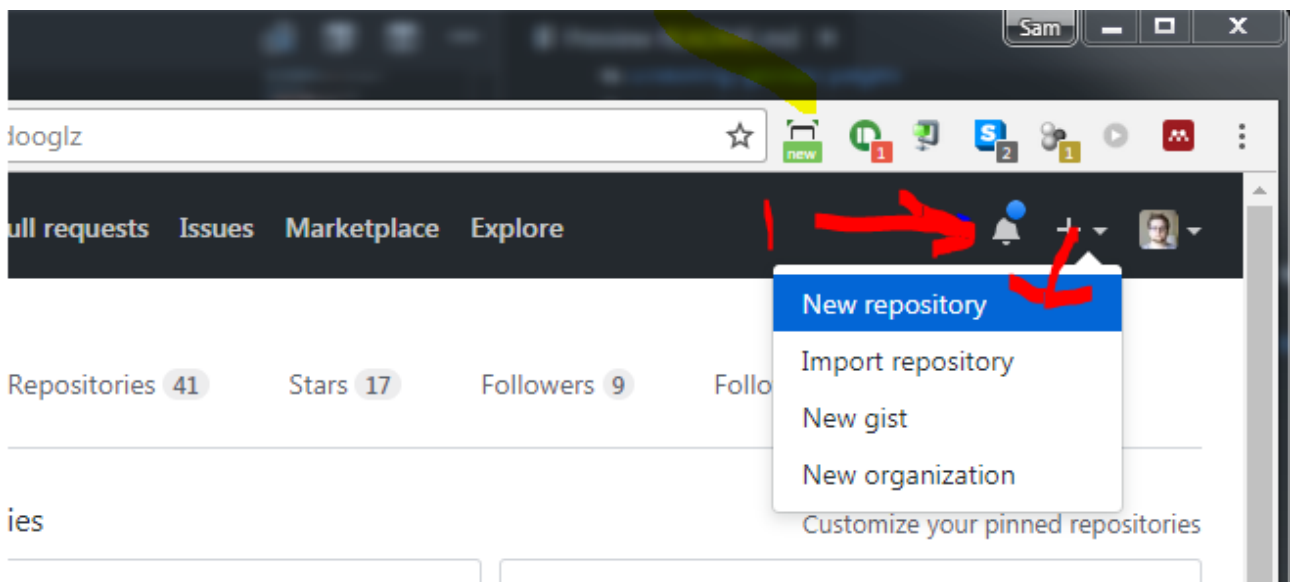



Figure 1: New Repo process

Create a new repository

A repository contains all the files for your project, including the revision history.

Owner:  dooglz / Repository name: ✓

Great repository names are short and memorable. Need inspiration? How about **urban-parakeet**.

Description (optional):

☒ Public
Anyone can see this repository. You choose who can commit.

☐ Private
You choose who can see and commit to this repository.

☒ Initialize this repository with a README
This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: | Add a license: ⓘ

Figure 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

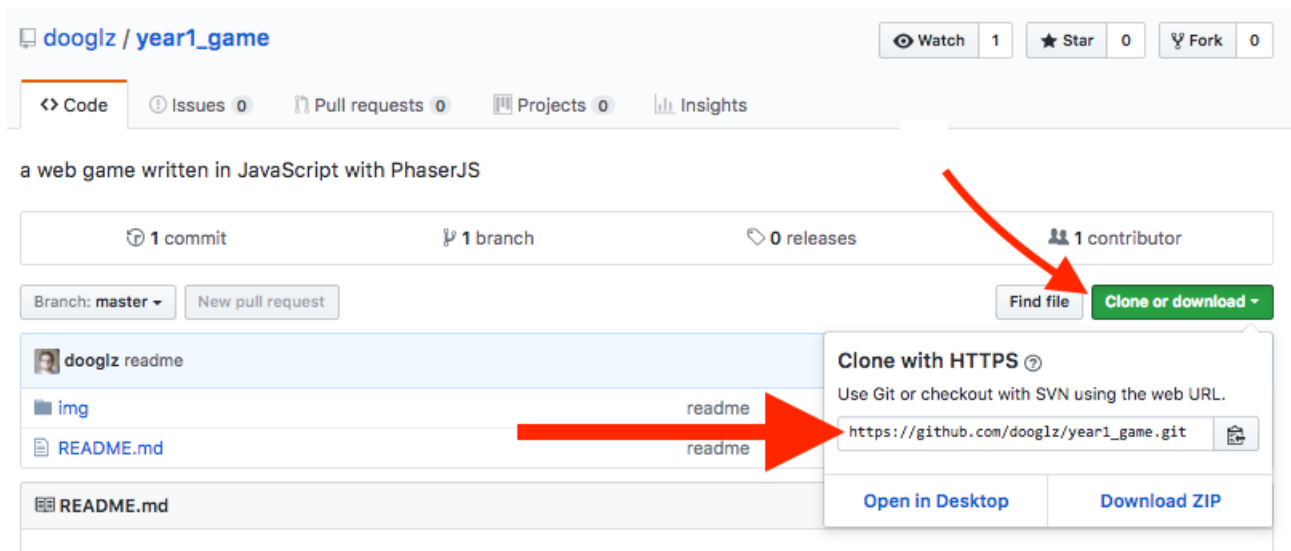


Figure 3: Clone from Github

0.1.2 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.

0.1.3 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.

0.1.4 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.

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

0.1.5 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:

```
1 #include <SFML/Graphics.hpp>
2
3 int main(){
4     sf::RenderWindow window(sf::VideoMode(200, 200), "SFML works!");
5     sf::CircleShape shape(100.f);
6     shape.setFillColor(sf::Color::Green);
7
8     while (window.isOpen()){
9         sf::Event event;
10        while (window.pollEvent(event)){
11            if (event.type == sf::Event::Closed){
12                window.close();
13            }
14        }
15        window.clear();
16        window.draw(shape);
17        window.display();
18    }
19    return 0;
20 }
```

Listing 1: practical_1/main.cpp

0.1.6 Step 6 - Create the CMake script

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

```

1 cmake_minimum_required(VERSION 3.9)
2 # Require modern C++
3 set(CMAKE_CXX_STANDARD 14)
4 set(CMAKE_CXX_STANDARD_REQUIRED ON)
5
6 project(Games_Engineering)
7
8 ##### Setup Directories #####
9 #Main output directory
10 SET(OUTPUT_DIRECTORY "${CMAKE_BINARY_DIR}/bin/")
11 # Output all DLLs from all libs into main build folder
12 SET(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${OUTPUT_DIRECTORY})
13
14 ##### Add External Dependencies #####
15 add_subdirectory("lib/sfml")
16 set(SFML_INCS "lib/sfml/include")
17 link_directories("${CMAKE_BINARY_DIR}/lib/sfml/lib")
18
19 ##### Practical 1 #####
20 file(GLOB_RECURSE SOURCES practical_1/*.cpp practical_1/*.h)
21 add_executable(PRACTICAL_1 ${SOURCES})
22 target_include_directories(PRACTICAL_1 PRIVATE ${SFML_INCS})
23 target_link_libraries(PRACTICAL_1 sfml-graphics)

```

Listing 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.

0.1.7 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.

0.1.8 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 to 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.

0.1.9 Starting from scratch

If you want to 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 1

Pong

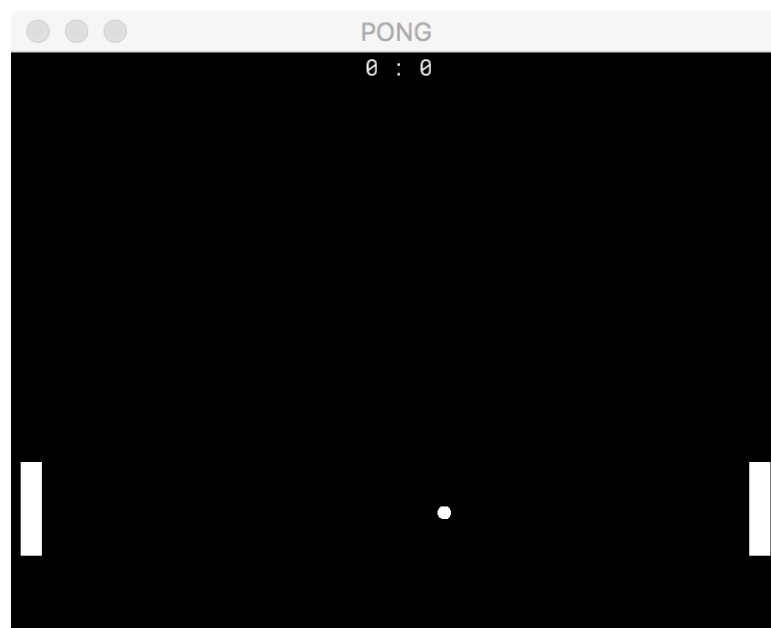


Figure 1.1: **Completed PONG**

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.

1.1 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 1.1: 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 1.2: SFML DT

1.1.1 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 1.3: 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 solution 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 1.4: 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 – $DT = 1/60$.
- If the game drops down to 30fps – $DT = 1/30$
- When the game is running slower – the player moves faster.
- When the game is running faster – the player moves slower.

1.1.2 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 2D 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 1.5: 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 or disable Vsync. So don't depend on it always being in the state that you set it.

1.2 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 1.6: 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 1.7: practical_1/main.cpp

Render and Main Our last section of the file is our 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 1.8: practical_1/main.cpp

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

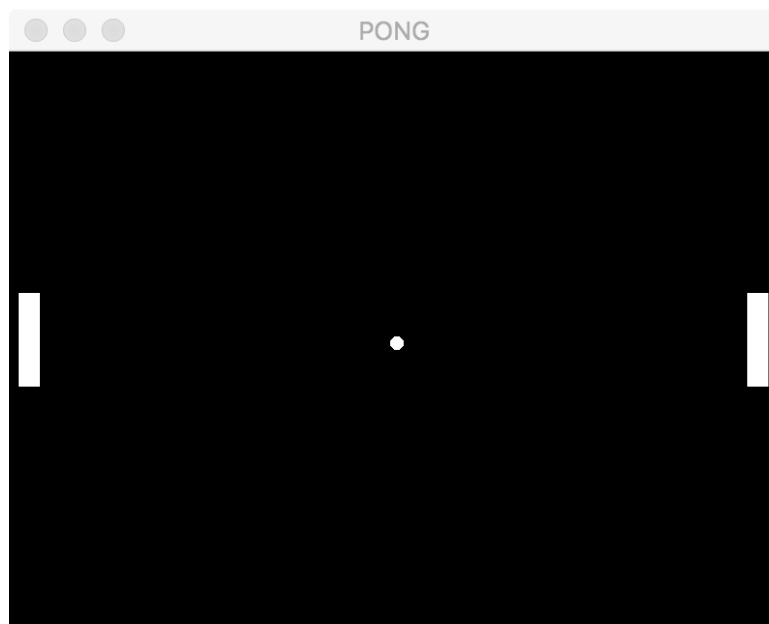


Figure 1.2: PONG hello world

1.2.1 Moving the ball

Add the following to the declarations at the start of 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 its 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.

1.2.2 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 world that would be fine, but if the ball was flying super fast, it 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 the 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 from 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.

```
1 else if (  
2     //ball is inline or behind paddle  
3     bx < paddleSize.x &&  
4     //AND ball is below top edge of paddle  
5     by > paddles[0].getPosition().y - (paddleSize.y * 0.5) &&  
6     //AND ball is above bottom edge of paddle  
7     by < paddles[0].getPosition().y + (paddleSize.y * 0.5)  
8 ) {  
9     // bounce off left paddle  
10 } else if (...) {  
11     // bounce off right paddle  
12 }
```

1.2.3 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.

1.2.4 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.

1.2.5 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.

1.2.6 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 2

Space Invaders

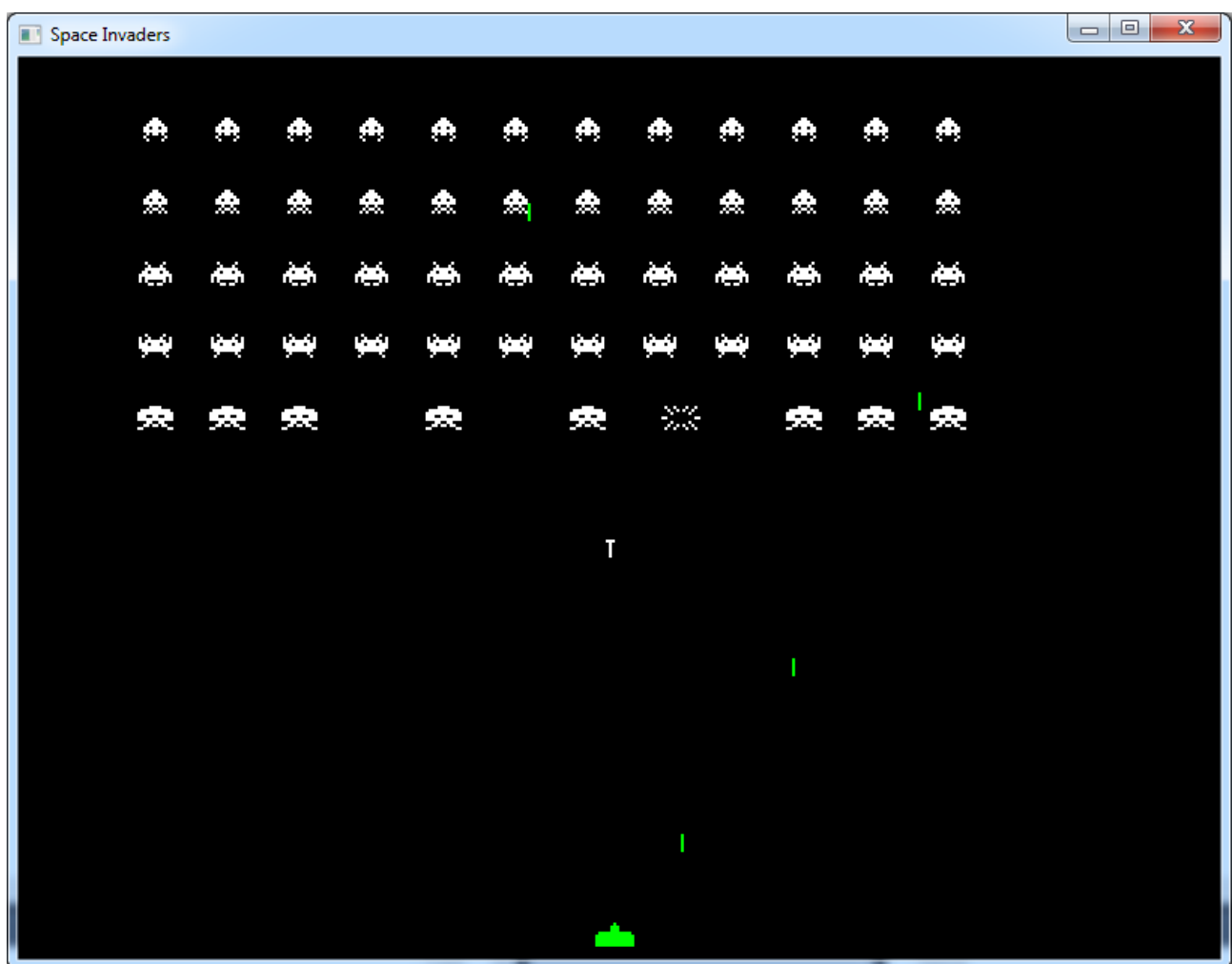


Figure 2.1: Completed Space Invaders

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.

2.1 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 2.1: 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 to 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 inside 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.