SET09121 Games Engineering

Workbook 2018

Dr Kevin Chalmers

Sam Serrels

Edinburgh Napier University

February 27, 2018

Version: 0.95

Contents

List of Figures

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. 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

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:

```cpp
int score = 3;
while(true){
// *******************************
// You have to complete this problem


// *******************************
}
```

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.

```cpp
auto h = renderer::window->GetHeight();
auto w = renderer::window->GetWidth()
//Calculate aspect ratio
//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)
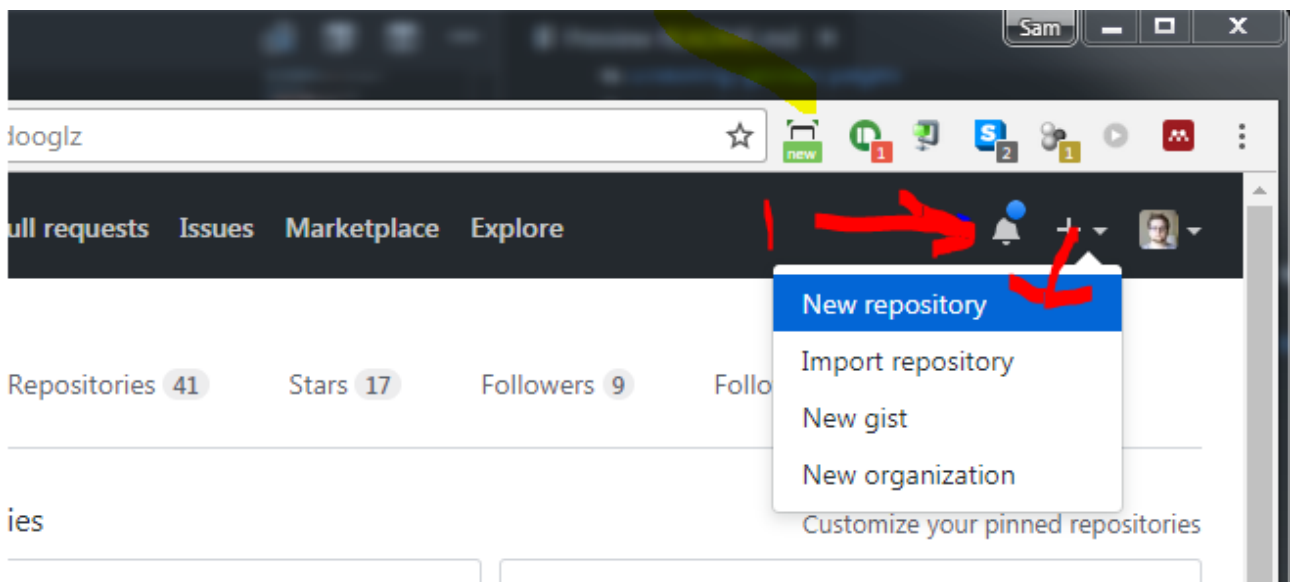
Figure 1: **New Repo process**

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
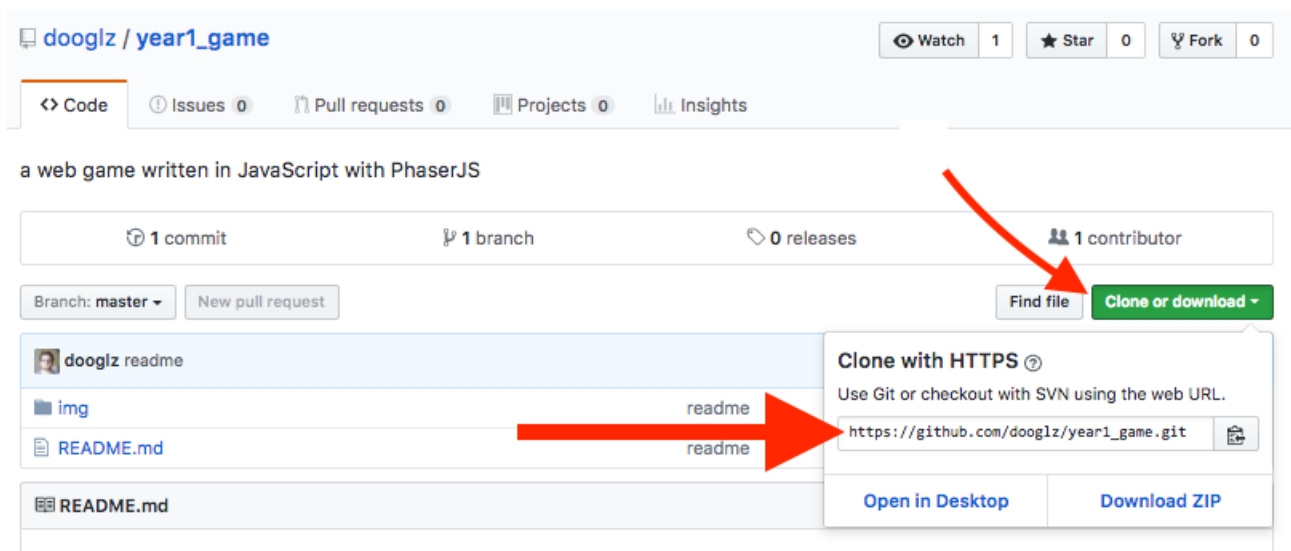
Figure 3: **Clone from Github**

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

```
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

```
cd set09121_labs
```

Now if you run

```
git status
```

You should see something similar to

```
On branch master
Your branch is up-to-date with 'origin/master'.
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.

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

```cpp
#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: 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:

```cmake
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 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 <u>outside</u> of the set09121 folder. Preferably your desktop, <u>NOT your H drive</u>

<u>NEVER Build from your H drive!</u>
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 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.

0.1.9   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 1

Pong
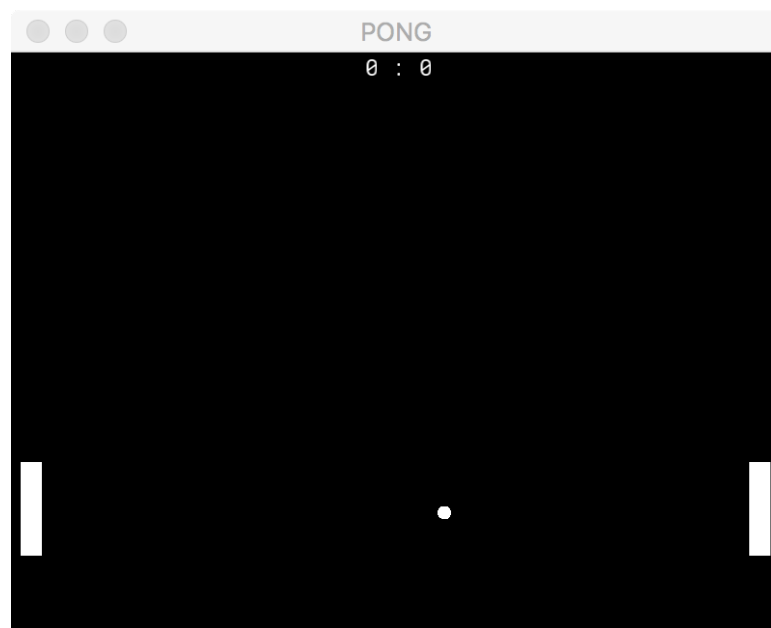
Figure 1.1: **Completed PONG**

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.

```cpp
void Update(double dt) {
  // Update Everything
}

void Render() {
  // Draw Everything
}

int main () {
  //initialise and load
  while(!shouldQuit){
    //Caluclate DT
    Update(dt);
    Render();
    //Wait for Vsync
  }
  //Unload and shutdown
}
```

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.

```cpp
static sf::Clock clock;
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.

```cpp
const float playerSpeed = 10.f;
const float idealFps = 60.f;

void Update(double dt) {
 player.move(playerSpeed / idealFps);
}
```

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 solutions is to **Always include DT in calculations that involve time. i.e speed, movement, acceleration.** Here is the proper way to do it:

```cpp
const float playerSpeed = 10.f;

void Update(double dt) {
  player.move(playerSpeed * dt);
}
```

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 = 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.

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 @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.

```
//End the frame - send to monitor - do this every frame
window.display();

// enable or disable vsync -
// do at start of game, or via options menu
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 ro 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.

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

using namespace sf;
using namespace std;

const Keyboard::Key controls[4] = {
    Keyboard::A,    // Player1 UP
    Keyboard::Z,    // Player1 Down
    Keyboard::Up,   // Player2 UP
    Keyboard::Down  // Player2 Down
};
const Vector2f paddleSize(25.f, 100.f);
const float ballRadius = 10.f;
const int gameWidth = 800;
const int gameHeight = 600;
const float paddleSpeed = 400.f;

CircleShape ball;
RectangleShape paddles[2];

void Load() {
  // Set size and origin of paddles
  for (auto &p : paddles) {
    p.setSize(paddleSize - Vector2f(3, 3));
    p.setOrigin(paddleSize / 2.f);
  }
  // Set size and origin of ball
  ball.setRadius(ballRadius - 3);
  ball.setOrigin(ballRadius / 2, ballRadius / 2);
  // reset paddle position
  paddles[0].setPosition(10 + paddleSize.x / 2, gameHeight / 2);
  paddles[1].setPosition(...);
  // reset Ball Position
  ball.setPosition(...);
}
```

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

```cpp
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 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.

```cpp
void Render(RenderWindow &window) {
  // Draw Everything
  window.draw(paddles[0]);
  window.draw(paddles[1]);
  window.draw(ball);
}

int main() {
  RenderWindow window(VideoMode(gameWidth, gameHeight), "PONG");
  Load();
  while (window.isOpen()) {
    window.clear();
    Update(window);
    Render(window);
    window.display();
  }
  return 0;
}
```

Listing 1.8: practical_1/main.cpp

Once you've typed all of the above in: you should have something like this:
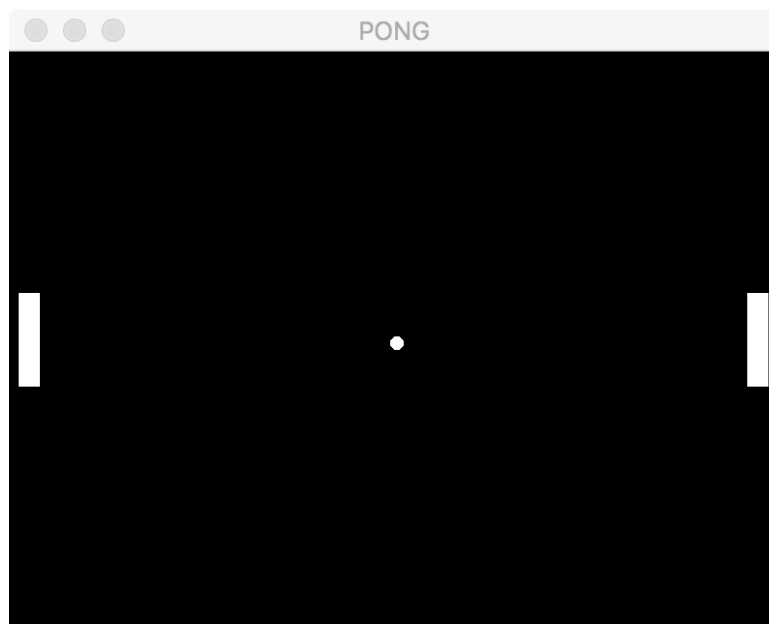
Figure 1.2: **PONG hello world**

1.2.1 Moving the ball

Add the following to the declarations at the start 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.

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 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.

```
// check ball collision
const float bx = ball.getPosition().x;
const float by = ball.getPosition().y;
if (by > gameHeight) {
  // bottom wall
  ballVelocity.x *= 1.1f;
  ballVelocity.y *= -1.1f;
  ball.move(0, -10);
} else if (by < 0) {
  // top wall
  ballVelocity.x *= 1.1f;
  ballVelocity.y *= -1.1f;
  ball.move(0, 10);
}
```

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.

```
else if (bx > gameWidth) {
  // right wall
  reset();
} else if (bx < 0) {
  // left wall
  reset();
}
```

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()

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

Add the following to your Reset()

```
// Update Score Text
text.setString(...);
// Keep Score Text Centered
text.setPosition((gameWidth * .5f) - (text.getLocalBounds().width * .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
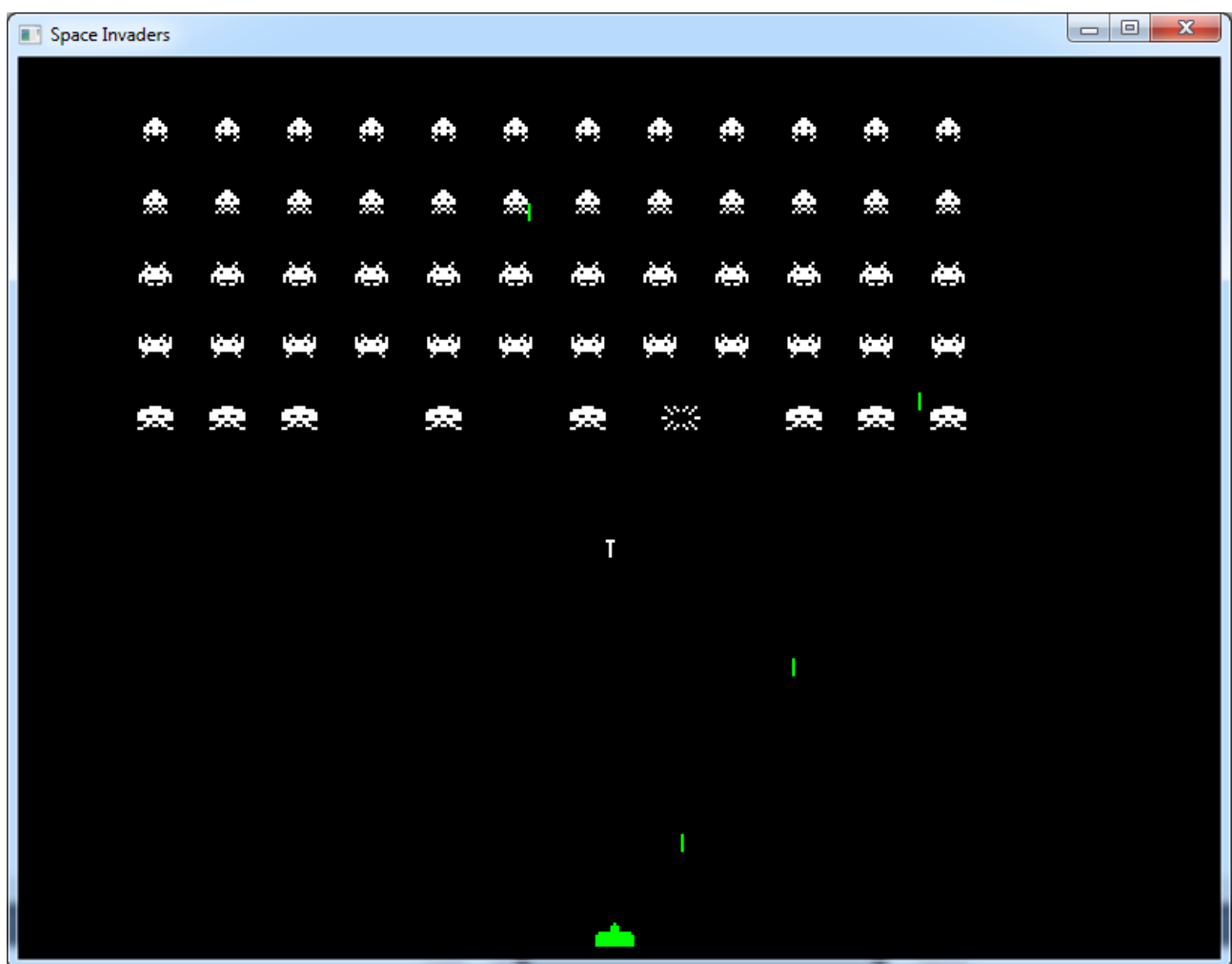
Figure 2.1: **Completed Space Invaders**

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.

```
## Space invaders
file(GLOB_RECURSE SOURCES 2_invaders/*.cpp 2_invaders/*.h)
add_executable(2_INVADERS ${SOURCES} )
target_include_directories(2_INVADERS SYSTEM PRIVATE ${SFML_INCS})
target_link_libraries(2_INVADERS sfml-graphics)
```

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 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.

2.2 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 2.2: **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
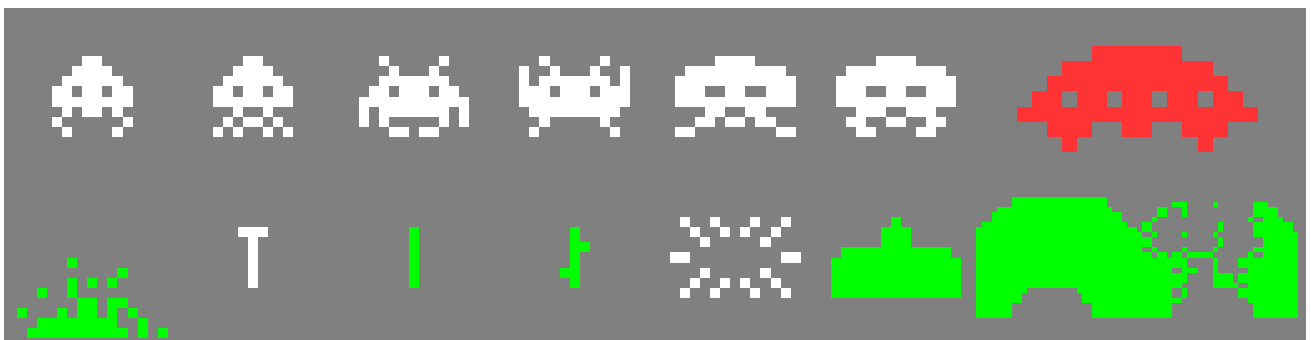
Figure 2.3: **Our Space Invaders Sprite-sheet** It's actually a transparent image - grey background just for clarity.

Figure 2.4: **Minecraft's Textures** Having all the textures in one image allows for easy modability. The multiple squares of water and lava are animation frames.
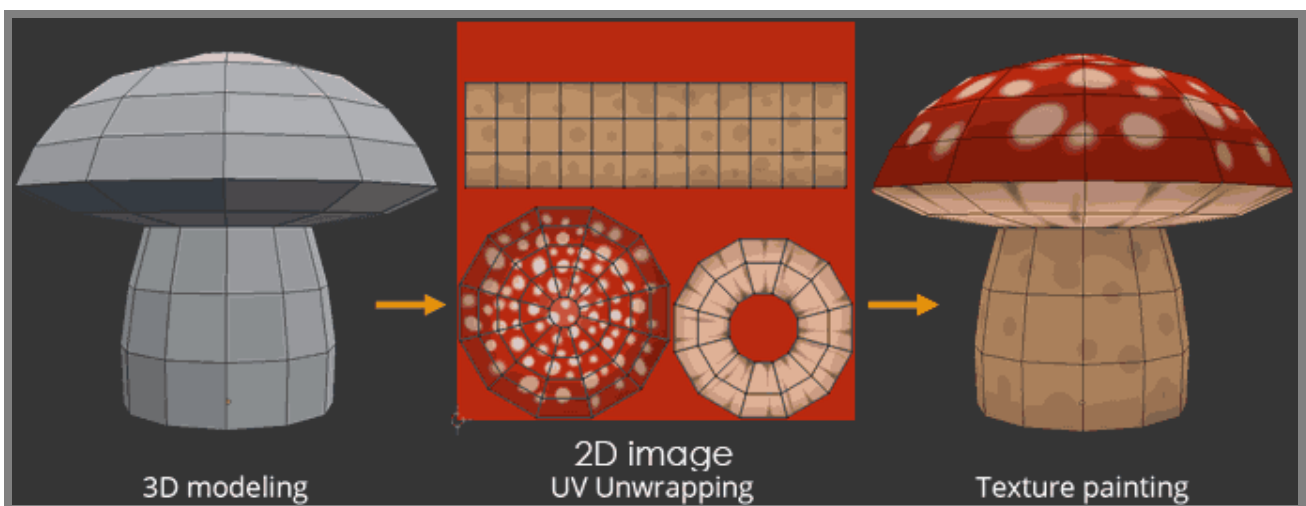
Figure 2.5: **Sprite sheets are not UV maps** Although visually similar - don't confuse sprite-sheets for unwrapped UV textures. These are used to apply a texture to a 3d model. Sprite-sheets are for 2d sprites

28

2.3 Loading a sprite-sheet

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

```
sf::Texture spritesheet;
sf::Sprite invader;


void Load() {
  if (!spritesheet.loadFromFile("res/img/invaders_sheet.png")) {
    cerr << "Failed to load spritesheet!" << std::endl;
  }
  invader.setTexture(spritesheet);
  invader.setTextureRect(sf::IntRect(0, 0, 32, 32));
}


void Render() {
  window.draw(invader);
}
```

Listing 2.2: Sprite sheet code

Totally easy. Take note of this line:

```
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.

2.3.1 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.

Firstly, get Visual studio running in the right place Before we know where to put out resources, we need to know where Visual studio is running our program from. This could be the source folder, it could be somewhere in the build folder. This cmake script will alter VS to run your program out of the folder it was built into. i.e. "build/bin/debug".

```
set_target_properties(PRACTICAL_2_INVADERS
  PROPERTIES VS_DEBUGGER_WORKING_DIRECTORY
  ${CMAKE_RUNTIME_OUTPUT_DIRECTORY}/$(Configuration)
)
```

Listing 2.3: Setting VS working directory

Now we know that's where we need to put our resources.

Method 1 : Altering the working directory   A commonly used practise for getting your runtime resources where you need them is altering the debugging "Working Directory" to point to the source folder (like we did just above, but set to the source directory). 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

```
1  add_custom_command(TARGET YOUR_EXECUTABLE POST_BUILD
2    COMMAND ${CMAKE_COMMAND} -E copy_if_different "resource_dir"
3      $<TARGET_FILE_DIR:${YOUR_EXECUTABLE}>/res
4  )
```

Listing 2.4: 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.

```
1  add_custom_target(copy_resources ALL COMMAND ${CMAKE_COMMAND}
2    -E copy_directory
3      "${PROJECT_SOURCE_DIR}/res"
4      ${CMAKE_RUNTIME_OUTPUT_DIRECTORY}/$<CONFIGURATION>/res
5  )
```

Listing 2.5: 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:

```
1  add_dependencies(PRACTICAL_2_INVADERS copy_resources)
```

Listing 2.6: custom target dependancy
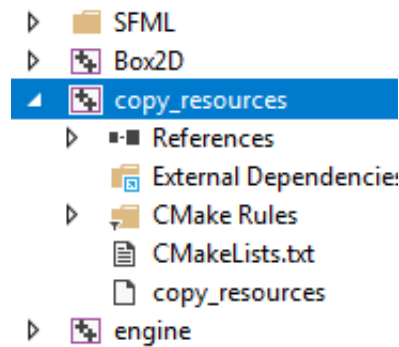
Now whenever we build Space_Invaders, Visual studio will make sure that 'copy_resources' has been run. Excellent.

Shortcut for linking libraries   Every project we will need will add to our CMake will have some identical options. Mainly linking with SFML and running our copy_resources target. Duplicated code is bad code. Let's fix that with some additional CMake coolness.

Add this to the end of your CMake script :

```
#### Link Dependencies ####
foreach ( exe ${EXECUTABLES})
  #Set working directory to build dir in visual studio
  set_target_properties(${exe} PROPERTIES VS_DEBUGGER_WORKING_DIRECTORY
    ${CMAKE_RUNTIME_OUTPUT_DIRECTORY}/$(Configuration)
  )

  target_link_libraries(${exe}
    #lib_maths # We'll make this later.
    optimized sfml-graphics debug sfml-graphics-d
    optimized sfml-audio debug sfml-audio-d
    optimized sfml-window debug sfml-window-d
    optimized sfml-system debug sfml-system-d
    optimized sfml-graphics debug sfml-graphics-d
    optimized sfml-main
  )

  add_dependencies(${exe} sfml-graphics sfml-audio copy_resources)
endforeach ()
```

Listing 2.7: Addition to CMakeLists.txt

So our final listing for Space invaders now looks like this:

```
## Space invaders
file(GLOB_RECURSE SOURCES 2_space_invaders/*.cpp 2_space_invaders/*.h)
add_executable(2_INVADERS ${SOURCES})
target_include_directories(2_INVADERS SYSTEM PRIVATE ${SFML_INCS})
set(EXECUTABLES ${EXECUTABLES} 2_INVADERS)
```

Listing 2.8: Addition to CMakeLists.txt

2.4 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().
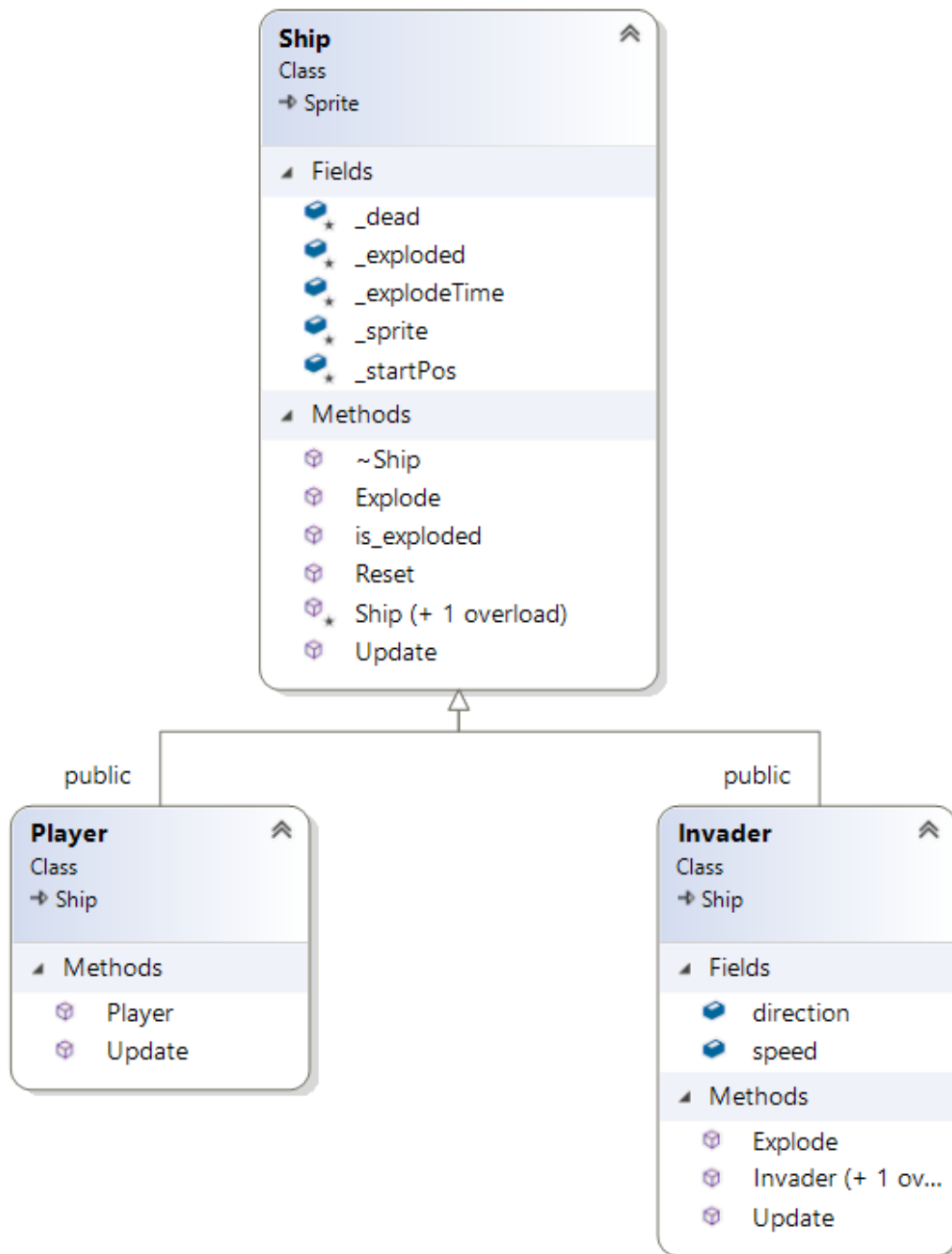
Figure 2.6: **Completed Space Invaders Class Diagram** don't worry about implmenting all these properties just yet

Create Ship.h Create a file inside the invaders source folder called "ship.h". This will be our Header file for the Ship class. Header files contain the declaration of our class, i.e only the function declarations. Headers 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 header. 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    //Default constructor is hidden
8    Ship();
9  public:
10   //Constructor that takes a sprite
11   explicit Ship(sf::IntRect ir);
12   //Pure virtual deconstructor -- makes this an abstract class
13   virtual ~Ship() = 0;
14   //Update, virtual so can be overrided, but not pure virtual
15   virtual void Update(const float &dt);
16 };
```

Listing 2.9: Initial Ship.h

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

```
1  #include "ship.h"
2  using namespace sf;
3  using namespace std;
4
5  Ship::Ship() {};
6
7  Ship::Ship(IntRect ir) : Sprite() {
8    _sprite = ir;
9    setTexture(spritesheet);
10   setTextureRect(_sprite);
11 };
12
13 void Ship::Update(const float &dt) {}
14
15 //Define the ship deconstructor.
16 //Although we set this to pure virtual, we still have to define it.
17 Ship::~Ship() = default;
```

Listing 2.10: 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 header file. To do this, create another header, 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 2.11: 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 2.12: main.cpp

This should compile without errors.

2.5   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 Header 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 2.13: Ship.h with inital Invader class

```cpp
Invader::Invader() : Ship() {}

Invader::Invader(sf::IntRect ir, sf::Vector2f pos) : Ship(ir) {
    setOrigin(16, 16);
  setPosition(pos);
}

void Invader::Update(const float &dt) {
  Ship::Update(dt);
}
```

Listing 2.14: Ship.cpp with inital Invader class

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

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

Listing 2.15: 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 <u>must</u> be <u>Ship Pointers</u>. 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.

```cpp
Update(){...
  for (auto &s : ships) {
    s->Update(dt);
  };
```

Listing 2.16: main.cpp

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

```cpp
Render() {...
  for (const auto s : ships) {
    window.draw(*s);
  }
```

Listing 2.17: main.cpp

2.5.1 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.

```cpp
class Invader : public Ship {
public:
  static bool direction;
  static float speed;
  ...
```

Listing 2.18: Ship.h

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

```cpp
bool Invader::direction;
float Invader::speed;
```

Listing 2.19: 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.

```cpp
void Invader::Update(const float &dt) {
  Ship::Update(dt);

  move(dt * (direction ? 1.0f : -1.0f) * speed, 0);

  if ((direction && getPosition().x > gameWidth - 16) ||
    (!direction && getPosition().x < 16)) {
      direction = !direction;
        for (int i = 0; i < ships.size(); ++i) {
          ships[i]->move(0, 24);
        }
  }
}
```

Listing 2.20: 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.

2.5.2 Spawning Invaders

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

```cpp
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 2.21: main.cpp

2.6 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

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

Listing 2.22: ship.h

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

```cpp
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 2.23: 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.

2.7 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 <u>one</u> bullet class.

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

```cpp
#pragma once
#include <SFML/Graphics.hpp>

class Bullet : public sf::Sprite {
public:
  void Update(const float &dt);
  Bullet(const sf::Vector2f &pos, const bool mode);
  ~Bullet()=default;
protected:
  Bullet();
  //false=player bullet, true=Enemy bullet
  bool _mode;
};
```

Listing 2.24: bullet.h

```cpp
#include "bullet.h"
#include "game.h"
using namespace sf;
using namespace std;

//Create definition for the constructor
//...

void Bullet::Update(const float &dt) {
    move(0, dt * 200.0f * (_mode ? 1.0f : -1.0f));
}
```

Listing 2.25: bullet.cpp

2.7.1 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:

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

Listing 2.26: 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**.

2.7.2 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:

```
Player::Update(){...
  static vector<bullet*> bullets;
  if (Keyboard::isKeyPressed(...)) {
    bullets.push_back(new Bullet(getPosition(), false));
  }
  for (const auto s : bullets) {
    bullets.Update(dt);
  }
```

Listing 2.27: 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.

```
class Bullet : public sf::Sprite {
...
protected:
static unsigned char bulletPointer;
static Bullet bullets[256];
...
```

Listing 2.28: 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.

```
class Bullet : public sf::Sprite {
public:
  //updates All bullets
  static void Update(const float &dt);
  //Render's All bullets
  static void Render(sf::RenderWindow &window);
  //Chose an inactive bullet and use it.
  static Fire(const sf::Vector2f &pos, const bool mode);

  ~Bullet()=default;
Protected:
  static unsigned char bulletPointer;
  static Bullet bullets[256];
  //Called by the static Update()
  void _Update(const float &dt);
  //Never called by our code
  Bullet();
  //false=player bullet, true=Enemy bullet
  bool _mode;
};
```

Listing 2.29: 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.

2.8   Exploding Things

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

```cpp
void Bullet::_Update(const float &dt) {
    if (getPosition().y < -32 || getPosition().y > gameHeight + 32) {
        //off screen - do nothing
        return;
    } else {
        move(0, dt * 200.0f * (_mode ? 1.0f : -1.0f));
        const FloatRect boundingBox = getGlobalBounds();

        for (auto s : ships) {
            if (!_mode && s == player) {
                //player bulelts don't collide with player
                continue;
            }
            if (_mode && s != player) {
                //invader bullets don't collide with other invaders
                continue;
            }
            if (!s->is_exploded() &&
                s->getGlobalBounds().intersects(boundingBox)) {
                //Explode the ship
                s->Explode();
                //warp bullet off-screen
                setPosition(-100, -100);
                return;
            }
        }
    }
};
```

Listing 2.30: 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.

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

Listing 2.31: ship.h

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

Listing 2.32: bullet.cpp

2.9 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.

```cpp
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 2.33: 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.

2.9.1   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:

```
void Invader::Update(const float &dt) {
   ...
   static float firetime = 0.0f;
   firetime -= dt;
   ...
   f (firetime <= 0 && rand() % 100 == 0) {
      Bullet::fire(getPosition(), true);
      firetime = 4.0f + (rand() % 60);
   }
}
```

Listing 2.34: 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.

2.9.2   Fading the Explosion sprite

At 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.

2.10   Future

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

Lesson 3

Tile Engine
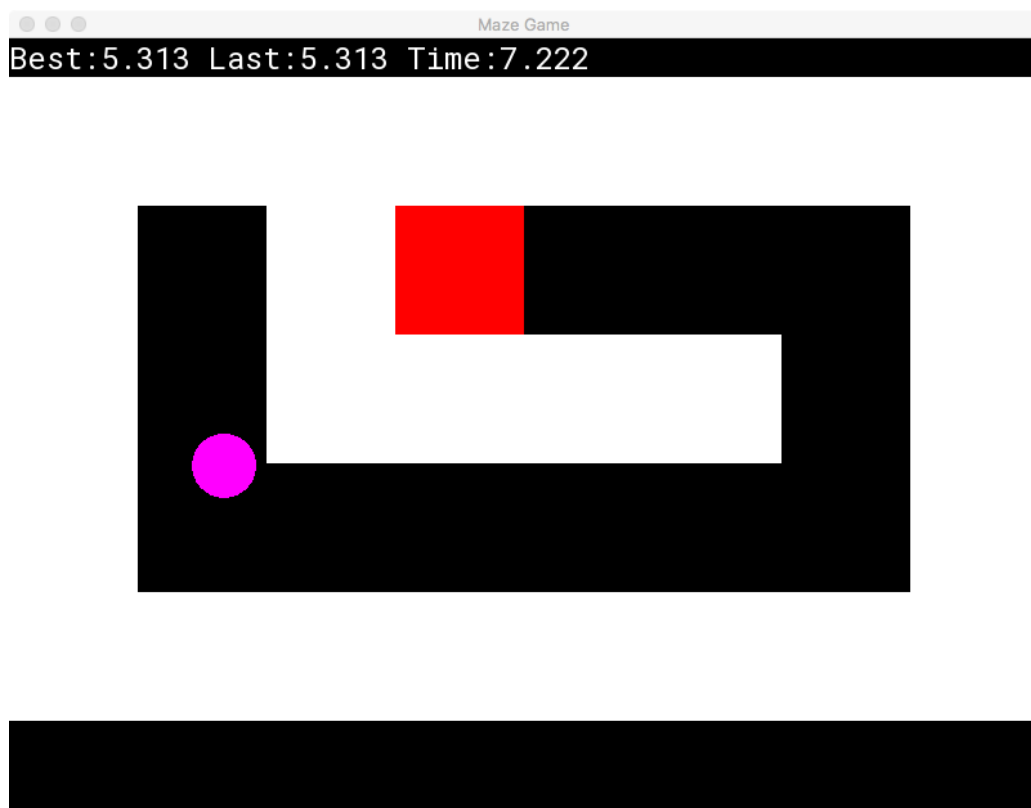
Figure 3.1: **Completed Maze Game**

This lab is design to introduce you to: Writing C++ Helper Libraries, Tile Based logic and 2D coordinate code.

3.1 Get Started

We will be starting this practical from the usual game loop basis. Create the main.cpp, and add the new project to CMake.

3.1.1 The Goal

The Game we are about to make is a maze game. The player moves a purple circle around from a starting point and tries to get to a determined end point in the shortest amount of time.

The Maze will be loaded from a text file with a super simple format. The logic for loading, rendering, and managing the maze data will be handled by a standalone "LevelSystem" library that we will build. This means we can use this code again in future projects. While we are building libraries, we will also make a small Maths helper library to cover some of the functions that SFML doesn't provide with its vector class.

The maze file, along with all other resources can be found in the Repo (section 0.0.1)

```
wwwwwww
wswe    w
w wwww  w
w       w
wwwwwww
```

Listing 3.1: Maze file

3.2 Step One - Player Entity

While we won't have any need for complex inheritance or large software patterns, for this practical we are still going to follow the Entity Model. This will be a slight change form Space invaders in that we are not going to inherit from an SFML class.

3.2.1 Entity.h and Entity.cpp

For this game we will be working with sf::shapes rather than sf::sprites. They are both sibling classes that derive from sf::drawable, but are incompatible with each other (you can't allocate a shape with a sprite.)

Our base Entity class will be abstract, due to having a pure virtual Render() function.

The default constructor is also deleted, meaning the only way to construct it is through the constructor that takes a sf::shape. Meaning that all classes that derive from Entity must provide an sf::shape when they are constructed.

```cpp
#pragma once

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

class Entity {
protected:
   std::unique_ptr<sf::Shape> _shape;
   sf::Vector2f _position;
   Entity(std::unique_ptr<sf::Shape> shp);

public:
   Entity() = delete;
   virtual ~Entity() = default;

   virtual void update(const double dt);
   virtual void render(sf::RenderWindow &window) const = 0;

   const sf::Vector2f getPosition();
   void setPosition(const sf::Vector2f &pos);
   void move(const sf::Vector2f &pos);
};
```

Listing 3.2: entity.h

Smart Pointers An important difference here is that the introduction of smart pointers (std::unique_ptr). So far we have been using "Raw" pointers, and creating them with New(), and deleting them with Delete().

Smart pointers use reference counting to know where a pointer is stored. If it isn't stored anywhere, the reference count is 0 and the smart pointer deletes the memory associated with the pointer. This means that we don't have to call Delete() ourselves, as we would with raw pointers.

This is similar but different to "Garbage Collection" in Java. Smart pointers de-allocates memory when the last reference goes out of scope. This is deterministic, and so you can know when it will happen. With Java, the garbage collection process runs separately and deallocated memory based on internal algorithms – separate from your code.

Entity.cpp   The definitions in the .cpp don't do anything fancy. We are just covering the basic movement functions that we no longer have from inhering from sf::sprite. Notice the Update function in particular. This is how we synchronise the state of the entity to SFML.

```cpp
#include "entity.h"
using namespace std;
using namespace sf;

const Vector2f Entity::getPosition() { return _position;}

void Entity::setPosition(const Vector2f &pos) { _position = pos;}

void Entity::move(const Vector2f &pos) { _position += pos;}

void Entity::update(const double dt) {
   _shape->setPosition(_position);
}

Entity::Entity(unique_ptr<Shape> s) : _shape(std::move(s)) {}
```

Listing 3.3: entity.cpp

3.2.2 Player.h and Player.cpp

Nothing really to see here for the player class, just a standard implementation of an Entity.

```cpp
#pragma once
#include "entity.h"

class Player : public Entity {
 private:
   float _speed;

 public:
   void update(double dt) override;
   Player();
   void render(sf::RenderWindow &window) const override;
};
```

Listing 3.4: player.h

player.cpp For the .cpp, for now we will keep this basic and come back to it. Implement code that will move the player around on screen in the update.

```cpp
#include "player.h"
using namespace sf;
using namespace std;

void Player::update(double dt) {
  //Move in for directions based on keys
  ...

  Entity::update(dt);
}

Player::Player()
    : _speed(200.0f), Entity(make_unique<CircleShape>(25.f)) {
  _shape->setFillColor(Color::Magenta);
  _shape->setOrigin(Vector2f(25.f, 25.f));
}

void Player::render(sf::RenderWindow &window) const {
  window.draw(*_shape);
}
```

Listing 3.5: player.h

At this point you should have a magenta circle moving around a screen. Do not continue on if you haven't got everything working so far.

WRITING LIBRARIES

3.3 Writing Libraries

For our next piece of work, we are going to write code that we are going to want to use again in future games. Standard practise would be to use good software engineering to isolate all the logic required for our Level loader code to it's own files and minimize coupling between itself and game code. Another approach is to build a "helper class" which is completely separate form the code of the game, but included in the build as headers and .cpp's. We could then easily move this code to a new project by just brining along the needed files. This is fairly common practise and most game programmers have a collection of small "helper" classes that they copy over from project to project.

The next step from this approach is to bundle up the code as a library. We are already using several libraries in our project - all the SFML components. Libraries are code that is complied separately to your program and linked in during the link stage. The primary benefit of this is that we don't have to compile libraries again once they are complied (imagine if we had to build SFML every time we needed to build our game code).

In certain situations you can download libraries pre-compiled from the internet. This isn't great for C++ as the compile settings need to be near identical for your application as for the downloaded library. Furthermore - we can't step down into source code when debugging. Providing a library with a well maintain ed CmakeLists.txt is by far the best way to distribute your code when building libraries for other programmers to use.

Static vs dynamic linking – .lib's and .dll's When we build a library - it will generate a .lib file. Depending on your build settings (Dynamic/Shared linking) it may also produce a .dll. The situation is more complicated but the simple explanation is that with dynamic linking" the code for our library lives inside the .dll (.so on linux/mac). When our application starts, it loads the code from the .dll into memory. This means that the .dll has to be somewhere the running program can access. We still need to link with a .lib file, this .lib will be an small file which only describes the .dll.

With static linking, the .lib is compiled into our executable. Meaning we don't need to bring any Dlls along. In this case the compiled .lib file contains all the code and will be significantly larger.

3.3.1 Setting this up

Finding and setting all the right settings in an IDE to set up building and linking libraries is a nightmare. With CMake it's laughably easier (and cross-platform!).

```
## Tile loader lib
file(GLOB_RECURSE SOURCE_FILES
    lib_tile_level_loader/*.cpp lib_tile_level_loader/*.h)
add_library(lib_tile_level_loader STATIC ${SOURCE_FILES})
target_include_directories(lib_tile_level_loader INTERFACE
            "${CMAKE_SOURCE_DIR}/lib_tile_level_loader/" )
target_link_libraries(lib_tile_level_loader sfml-graphics)
```

Listing 3.6: Addition to CMakeLists.txt

The biggest difference here is the call to "add_library" rather than "add_executable"

From the CMake we can see that we need to put some code in a "lib_tile_level_loader" folder. Library code is exactly the same as wiring any other c++ file, but we don't need a Main() function.

3.4 Level System Code

Let's get started with our header.

```cpp
#pragma once

#include <SFML/Graphics.hpp>
#include <memory>
#include <string>
#include <vector>

#define ls LevelSystem

class LevelSystem {
public:
  enum TILE { EMPTY, START, END, WALL, ENEMY, WAYPOINT };

  static void loadLevelFile(const std::string&,float tileSize=100.f);
  static void render(sf::RenderWindow &window);
  static sf::Color getColor(TILE t);
  static void setColor(TILE t, sf::Color c);
  //Get Tile at grid coordinate
  static TILE getTile(sf::Vector2ul);
  //Get Screenspace coordinate of tile
  static sf::Vector2f getTilePosition(sf::Vector2ul);
  //get the tile at screenspace pos
  static TILE getTileAt(sf::Vector2f);

protected:
  static std::unique_ptr<TILE[]> _tiles; //Internal array of tiles
  static size_t _width; //how many tiles wide is level
  static size_t _height; //how many tile high is level
  static sf::Vector2f _offset; //Screenspace offset of level, when rendered.
  static float _tileSize; //Screenspace size of each tile, when rendered.
  static std::map<TILE, sf::Color> _colours; //color to render each tile type

  //array of sfml sprites of each tile
  static std::vector<std::unique_ptr<sf::RectangleShape>> _sprites;
  //generate the _sprites array
  static void buildSprites();

private:
  LevelSystem() = delete;
  ~LevelSystem() = delete;
};
```

Listing 3.7: levelsystem.h

That's quite a lot to begin with. pay attention to the public functions first. This is where we declare what our Library can do. The protected variables are internal state that we need for some calculation later on. The whole LevelSystem is a static class, everything is static so we can access everything within it from anywhere (Downside: we can't inherit from it). I've thrown in a handy #define macro so we can access everything like "ls::render()". Vector2ul will give you an error, this is something that doesn't exist yet, more on this later.

With our Levelsystem declared, let's get to defining it in LevelSystem.cpp

```cpp
#include "LevelSystem.h"
#include <fstream>

using namespace std;
using namespace sf;

std::unique_ptr<LevelSystem::TILE[]> LevelSystem::_tiles;
size_t LevelSystem::_width;
size_t LevelSystem::_height;
Vector2f LevelSystem::_offset(0.0f, 30.0f);

float LevelSystem::_tileSize(100.f);
vector<std::unique_ptr<sf::RectangleShape>> LevelSystem::_sprites;

std::map<LevelSystem::TILE, sf::Color> LevelSystem::_colours{
    {WALL, Color::White}, {END, Color::Red}};

sf::Color LevelSystem::getColor(LevelSystem::TILE t) {
   auto it = _colours.find(t);
   if (it == _colours.end()) {
      _colours[t] = Color::Transparent;
   }
   return _colours[t];
}

void LevelSystem::setColor(LevelSystem::TILE t, sf::Color c) {
   ...
}
```

Listing 3.8: levelsystem.cpp

We start off by defining all the static member variables declared in the header file. This brings in a C++ data structure that we've not dealt with before, the map. It's statically initialised with two colours, more can be added by the game later. This map is read by the "getColor" function which will return a transparent colour if an allocation is not within the map.

You should complete the setColor function. It's super simple, but you may have to look up the c++ docs on the std::map.

Next up, reading in and parsing the text file.

```cpp
void LevelSystem::loadLevelFile(const std::string& path, float tileSize) {
  _tileSize = tileSize;
  size_t w = 0, h = 0;
  string buffer;

  // Load in file to buffer
  ifstream f(path);
  if (f.good()) {
    f.seekg(0, std::ios::end);
    buffer.resize(f.tellg());
    f.seekg(0);
    f.read(&buffer[0], buffer.size());
    f.close();
  } else {
    throw string("Couldn't open level file: ") + path;
  }

  std::vector<TILE> temp_tiles;
  for (int i = 0; i < buffer.size(); ++i) {
    const char c = buffer[i];
    switch (c) {
    case 'w':
      temp_tiles.push_back(WALL);
      break;
    case 's':
      temp_tiles.push_back(START);
      break;
    case 'e':
      temp_tiles.push_back(END);
      break;
    case ' ':
      temp_tiles.push_back(EMPTY);
      break;
    case '+':
      temp_tiles.push_back(WAYPOINT);
      break;
    case 'n':
      temp_tiles.push_back(ENEMY);
      break;
    case '\n':      // end of line
      if (w == 0) { // if we haven't written width yet
        w = i;      // set width
      }
      h++; // increment height
      break;
    default:
      cout << c << endl; // Don't know what this tile type is
    }
  }
  if (temp_tiles.size() != (w * h)) {
    throw string("Can't parse level file") + path;
  }
  _tiles = std::make_unique<TILE[]>(w * h);
```

```
54    _width = w; //set static class vars
55    _height = h;
56    std::copy(temp_tiles.begin(), temp_tiles.end(), &_tiles[0]);
57    cout << "Level " << path << " Loaded. " << w << "x" << h << std::endl;
58    buildSprites();
59 }
```

Listing 3.9: levelsystem.cpp

If we had many more tile types we would switch out that switch statement for a lop of some kind, but for the limited tile types we need it will do.

The file handling code at the top is nothing special, we read the whole file into a string then close the open file. This may be a bad move if the level file was larger, but this where we can get away with saying "C++ is fast, it doesn't matter".

Once the level string has been parsed into a vector of tile types, an array is created with the final dimensions. Keeping within a vector could be valid, but we don't ever want to change the size of it, so an array seems a better fit.

Notice that while the level file is 2D, we store it in a 1D storage type. If we know the width of the level we can extrapolate a 2d position from the 1d array easily. We do this as C++ doesn't have a native 2D array type. We could create an array of arrays which would do the job, but makes the functions we need to write later slightly more difficult.

Our level loader library will do more than just parse in a text file, it will also render the level with SFML. To do this we will build a list of sf::shapes for each tile in our array. The colour of this shape will depend on the colour association stored in our map. We only need to build this list of shapes once, so this function is called at the end of loadLevelFile().

```
1
2 void LevelSystem::buildSprites() {
3    _sprites.clear();
4    for (size_t y = 0; y < LevelSystem::getHeight(); ++y) {
5      for (size_t x = 0; x < LevelSystem::getWidth(); ++x) {
6        auto s = make_unique<sf::RectangleShape>();
7        s->setPosition(getTilePosition({x, y}));
8        s->setSize(Vector2f(_tileSize, _tileSize));
9        s->setFillColor(getColor(getTile({x, y})));
10       _sprites.push_back(move(s));
11     }
12   }
13 }
```

Listing 3.10: levelsystem.cpp

Notive we need yet another function, the getTilePosition().

```
1 sf::Vector2f LevelSystem::getTilePosition(sf::Vector2ul p) {
2    return (Vector2f(p.x, p.y) * _tileSize);
3 }
```

Listing 3.11: levelsystem.cpp

As we are just doing maths here we don't need to read into the tile array. We could add some validity checks to make sure the requested tile falls within our bounds, but I like my one-liner too much to bother with that.

There will be times when we need to retrieve the actual tile at a position, both screen-space and grid-space. This is where we must convert 2D coordinates to a single index in our tile array.

```cpp
LevelSystem::TILE LevelSystem::getTile(sf::Vector2ul p) {
  if (p.x > _width || p.y > _height) {
    throw string("Tile out of range: ") + to_string(p.x) + "," +
        to_string(p.y) + ")";
  }
  return _tiles[(p.y * _width) + p.x];
}
```

Listing 3.12: levelsystem.cpp

Most of this function is taken up by a range check (Where we throw an exception, new thing!). The real calculation is in that last line. The secret is to multiply the Y coordinate by the length and add the X. Don't continue on unless you understand why and how this works, it gets more difficult from here on out.

Doing the same, but with a screen-sapce coordinate is not any different. However as we are dealing with floats now, we must check it's a positive number first, then we can convert to grid-space, and call our above function. Again, don't continue on unless you understand why and how this works.

```cpp
LevelSystem::TILE LevelSystem::getTileAt(Vector2f v) {
  auto a = v - _offset;
  if (a.x < 0 || a.y < 0) {
    throw string("Tile out of range ");
  }
  return getTile(Vector2ul((v - _offset) / (_tileSize)));
}
```

Listing 3.13: levelsystem.cpp

And finally - here lies our Render Function. Nice and Simple.

```cpp
void LevelSystem::render(RenderWindow &window) {
  for (size_t i = 0; i < _width * _height; ++i) {
    window.draw(*_sprites[i]);
  }
}
```

Listing 3.14: levelsystem.cpp

3.4.1 Linking our Library

While the library build by itself, that's rather useless to us. We need to link it into our lab code. Back to CMake:

```
target_link_libraries(... lib_tile_level_loader sfml-graphics)
```

Listing 3.15: CMakeLists.txt

You probably could have guessed this addition. Just add the library target name to your link_libraries.

Using the library   Give it a test, Call some library functions from your lab code.

```cpp
#include "LevelSystem.h"

...

void load() {
  ...
  ls::loadLevelFile("res/maze_2.txt");

  // Print the level to the console
  for (size_t y = 0; y < ls::getHeight(); ++y) {
    for (size_t x = 0; x < ls::getWidth(); ++x) {
      cout << ls::getTile({x, y});
    }
    cout << endl;
  }
}
...
void render(RenderWindow &window) {
  ls::render(window);
  ...
}
```

Listing 3.16: main.cpp

3.5   Maths Library

Remember that Vector2ul type that doesn't exist? The vector maths functionality of SFML is quite lacking when compared to larger libraries like GLM. We could bring in GLM and write converter functions to allow it to interface with SFML. This would be a good idea if we needed to advanced thing like quaternions, but we don't. Instead we will build a small add-on helper library to add in the functions that SFML misses out.

CMake   Firstly, like we did with our first library, add this to CMake.

```
## Maths lib
add_library(lib_maths INTERFACE)
target_sources(lib_maths INTERFACE
    "${CMAKE_SOURCE_DIR}/lib_maths/maths.h")
target_include_directories(lib_maths INTERFACE
    "${CMAKE_SOURCE_DIR}/lib_maths" SYSTEM INTERFACE ${SFML_INCS})
```

Listing 3.17: Addition to CMakeLists.txt

This is slightly different to the level system library. This time we declare the library as IN-TERFACE. This changes some complex library and linker options that are beyond the scope of explanation here. The simplest explanation is an INTERFACE library target does not directly create build output, though it may have properties set on it and it may be installed, exported and imported. Meaning that in visual studio the library will look like it is part of our main lab code (except it isn't. Magic.)

There are many different way to create and link libraries, CMake allows us to change these options from a central point and not worry about digging through IDE options.

We need to link the maths library, against the levelsystem library. edit the level systems cmake code to include lib_maths

```
target_link_libraries(lib_tile_level_loader lib_maths sfml-graphics)
```

Listing 3.18: Addition to CMakeLists.txt

As this is a static library - and doesn't produce a compiled output, everything will be in a header. To extend the functionality of sf::vectors we must first be within the same namespace. From here we can define code as if we were inside the sfml library code itself. Thing get a little strange if we want to change or override functions that already exist, but we don't here as we are only creating new functionality.

We start by creating a new vector type the 'Vector2ul' which will use size_t (i.e the largest unsigned integer type supported on the system) as the internal components. We will use this for the tile array coordinates.

From this we implement the standard vector maths functions that any self respecting game engine would have. Length, Normalization, and Rotation. I've left these incomplete so you wil have to dredge up your vector maths skills to complete them.

Lastly, we override the ¡¡ stream operator to us do things like "cout << vector". Useful for debugging.

```cpp
#pragma once

#include <SFML/System.hpp>
#include <cmath>
#include <iostream>
#include <vector>

namespace sf {
  //Create a definition for a sf::vector using size_t types
  typedef Vector2<size_t> Vector2ul;
  // Returns the length of a sf::vector
  template <typename T> double length(const Vector2<T> &v) {
    return sqrt(...);
  }
  // return normalized sf::vector
  template <typename T> Vector2<T> normalize(const Vector2<T> &v) {
    Vector2<T> vector;
    double l = length(v);
    if (l != 0) {
      vector.x = ...
      vector.y = ...
    }
    return vector;
  }
  //Allow casting from one sf::vetor internal type to another
  template <typename T, typename U>
  Vector2<T> Vcast(const Vector2<U> &v) {
    return Vector2<T>(static_cast<T>(v.x), static_cast<T>(v.y));
  };
  // Degreess to radians conversion
  static double deg2rad(double degrees) {
    return ...
  }
  //Rotate a sf::vector by an angle(degrees)
  template <typename T>
  Vector2<T> rotate(const Vector2<T> &v, const double degrees) {
    const double theta = deg2rad(degrees);
    const double cs = cos(theta);
    const double sn = sin(theta);
    return {(T)(v.x * cs - v.y * sn), (T)(v.x * sn + v.y * cs)};
  }
  //Allow sf::vectors to be cout'ed
  template <typename T>
  std::ostream &operator<<(std::ostream &os, const Vector2<T> &v) {
    os << '(' << v.x << ',' << v.y << ')';
    return os;
  }
}
```

Listing 3.19: maths.h

That should be all we need to successfully build both our libraries and our game. Give it a go. Build. Run. See if your hard work typing all this has paid off.

3.6 Making the Game a Game

Player Class Disallow the player from moving into a tile:

Hint:

```cpp
bool validmove(Vector2f pos) {
  return (ls::getTileAt(pos) != ls::WALL);
}
```

Listing 3.20: main.cpp

Start the Player from the "Start Tile"

End the game When the player hit's the end tile

Time how long it takes to complete the maze

Show current, previous and best times

Lesson 4

Pacman

or: Engine Abstraction and the Entity Component Model
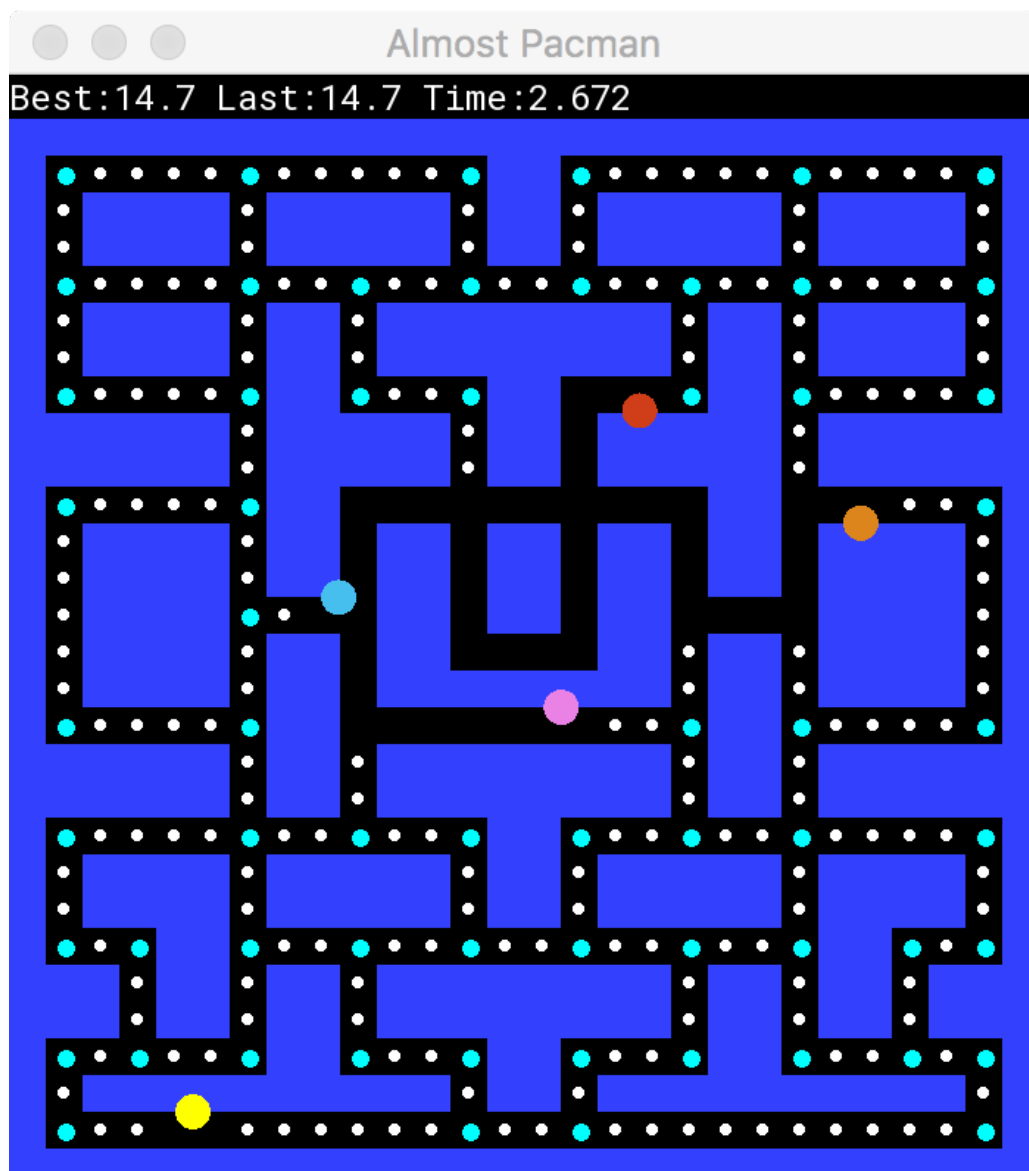
Figure 4.1: **Completed Pacman Game**

4.1 First Steps

1. Create a new project folder and add to CMake

2. Start with the usual blank gameloop example.

3. Create an abstract Entity Class.

4. Create a player class that inherits from Entity
   – moves around screen with keyboard keys
   – should be drawn as a yellow sf::shape

5. Create a "ghost" class, inherits from Entity.
   – should move around the screen randomly
   – should be a different colour to the player.

6. Main.cpp should create one player, and four ghosts, and store all in a *vector < entity∗ >*
   – should only call Update() and Render() on vector
   – all entity logic should be inside the entity classes.

The player and ghosts don't need any functional game logic behind them right now. Just make sure they are rendering and moving about before continuing on.

4.2 Engine Abstraction

This lab is where our game code starts to look more like a game *engine*. This is inevitable once a game reaches a certain level of complexity. Pacman just tips over that threshold where we can justify using some major-league game software architecture.

4.2.1 Entity Management

Games will have thousands of Entity's in flight, keeping them all in one global vector is not a good idea. Having multiple lists - relevant to their use is a better idea. This is a topic that strays instantly into optimizations, which really depend on the game you are making.

What we are going to go for in this lab is collating all the entities by the scene/level they are in. for pacman, we will just have two scenes: "Game" and Menu.

To move pacman to this paradigm is not going to take much of a change in code. Here's it is:

```cpp
struct EntityManager {
  std::vector<std::shared_ptr<Entity>> list;
  void update(double dt);
  void render(sf::RenderWindow &window);
};
```

Listing 4.1: Entity.h

The implementation of the two functions do exactly what you would expect, loop through the vector and update/render all Entities. Replace your Global entity vector in main.cpp with an

EntityManager called 'em', and insert both the player and ghosts into via em.list.push_back(). Swap out the calls to update and render to the pass through the manager instead.

You may be wondering why we even bothered doing this. We took simple code and made it more complex.

The point is to move logic out of main.cpp to a more appropriate place. This allows us to split our code into separate systems that we can expand upon and re-use later. This is the first small step on a big journey.

4.2.2 Render system

An annoying feature of our code right now is how we render Entities. Passing a reference around to the windows seems a little wrong seeing as the variable never changes. Let's fix this and pave the way for a new cool system.

The Renderer In most game engines, the system that handles rendering things is usually the largest and arguably the most important. For us, we only need to pass what we want to draw to SFML and it handles it all. You can bet the internals window.draw() function is pretty damn impressive (it is, go and look). We don't *need* a complex rendering system on-top of, but we'll build something anyway, if only to point out how a more complex system would do things.. If we weren't using SFML this is where things would get real complicated quickfast.

Our Render system will have a simplified Render() function that will take in a sf::Drawable object (e.g sprite,shape,text), and add this to a list of things to render.

The big difference here is that things won't be rendered immediately. The list will be built up of objects as each Render() function is called on all the Entities.

Once this process completed, we sent it all to SFML all at once.

The benefits to this is that we can keep track of how many things we are rendering per frame easily. More importantly it allows us to do optimisations. If z-order were important we could re-order the list to draw background objects first. Or do some form of debug-culling to stop certain object types of rendering. All useful stuff.

Again, if we were working with OpenGL or a more complex render system, this is were we would do some serious work. The reality is that SFML does almost everything for us so we don' actually have much to do here.

```cpp
#pragma once
#include <SFML/Graphics.hpp>

namespace Renderer {
  void initialise(sf::RenderWindow &);
  sf::RenderWindow &getWindow();

  void shutdown();
  void update(const double &);
  void queue(const sf::Drawable *s);
  void render();
};
```

Listing 4.2: system_renderer.h

```cpp
#include "system_renderer.h"
#include <queue>

using namespace std;
using namespace sf;

static queue<const Drawable *> sprites;
static RenderWindow *rw;

void Renderer::initialise(sf::RenderWindow &r) { rw = &r; }

sf::RenderWindow &Renderer::getWindow() { return *rw; }

void Renderer::shutdown() {
  while (!sprites.empty())
    sprites.pop();
}

void Renderer::update(const double &) {}

void Renderer::render() {
  if (rw == nullptr) {
    throw("No render window set! ");
  }
  while (!sprites.empty()) {
    rw->draw(*sprites.front());
    sprites.pop();
  }
}

void Renderer::queue(const sf::Drawable *s) { sprites.push(s); }
```

Listing 4.3: system_renderer.cpp

All that's left to do is Initialise the system from main.cpp, and call Renderer::render(); a the end of the main render() call.

Now whenever we need to render anything we can call something like.

```cpp
Renderer::queue(&text);
```

4.2.3 Scene Management

As mentioned above, we will have two 'scenes'. What is a Scene? Mainly it's a collection of Entities. All of the game logic will mainly be inside the Entities, but there will be some global game logic that needs to run, and the scene class is where it shall be.

The scene should also be responsible for loading and unloading content that it needs. This is where we could implement a loading screen. This would run while a scene loads, and then display the scene once finished. (We shouldn't be working with anything that will need a loading screen, simply freezing for a few frames while we transition scenes is acceptable).

Here we have our scene class. It has our usual two Update and Render functions, and an internal EntityManager. Additionally we have a Load() function and a public getter to the entity list.

```cpp
class Scene {
public:
  Scene() = default;

  virtual ~Scene() = default;
  virtual void update(double dt);
  virtual void render();
  virtual void load() = 0;
  std::vector<std::shared_ptr<Entity>> &getEnts();

protected:
  EntityManager _ents;
};
```

Listing 4.4: scene.h

We should instantiate tour two scenes. We should do this away from main.cpp – in a separate header that contains only Pacman related things. pacman.h. As these are extern'd, define them in pacman.cpp.

```cpp
extern std::shared_ptr<Scene> gameScene;
extern std::shared_ptr<Scene> menuScene;
extern std::shared_ptr<Scene> activeScene;
```

Listing 4.5: pacman.h

The Menu scene We've declared the two scenes we need as pointers to the base Scene class, when it comes to implementation, they will actually be two different classes. We should declare them in pacman.h and define them in pacman.cpp.

We will bring along anygameplay variables that would normally be in the global scope of main.cpp, and have them as private properties in each scene. For the Menu, this is just a single sf::Text.

```cpp
class MenuScene : public Scene {
private:
  sf::Text text;

public:
  MenuScene();
  void update(double dt) override;
  void render() override;
  void load()override;
};
```

Listing 4.6: pacman.h

The Game scene For the main game-play scene, we will have an extra method: Respawn() and a scoreClock. This is all we need for global game logic in the scene, the entities handle everything else.

```cpp
class GameScene : public Scene {
private:
  sf::Text text;
  sf::Clock scoreClock;
  void respawn();

public:
  GameScene() = default;
  void update(double dt) override;
  void render()override;
  void load()override;
};
```

Listing 4.7: pacman.h

The ghosts and layer that are still stored in a global EntityManager should now be moved into the GameScene. Do the Entity creation in the Load() function.

Instantiating the scenes   All that's left to do is actually Instantiate the two scenes, we do this in the main Load function. We an really start to see how we are separating out the logic to the different systems, with main.cpp becoming a small part that glues it all together.

```cpp
void Load() {
  // Load Scene-Local Assets
  gameScene.reset(new GameScene());
  menuScene.reset(new MenuScene());
  gameScene->load();
  menuScene->load();
  // Start at main menu
  activeScene = menuScene;
}
...
void Update() {
  static Clock clock;
  float dt = clock.restart().asSeconds();
  activeScene->update(dt);
}
...
void Render(RenderWindow &window) {
  activeScene->render();
  // flush to screen
  Renderer::render();
}
```

Listing 4.8: main.cpp

Changing scenes   Switching between the scenes is done with the variable 'activeScene'.

```cpp
void MenuScene::update(double dt) {
  if (Keyboard::isKeyPressed(Keyboard::Space)) {
    activeScene = gameScene;
  }
  Scene::update(dt);
  text.setString("Almost Pacman");
}
...
void GameScene::update(double dt) {
  if (Keyboard::isKeyPressed(Keyboard::Tab)) {
    activeScene = menuScene;
  }
  Scene::update(dt);
  ...
}
```

Listing 4.9: pacman.cpp

4.2.4 Checkpoint

After this long round trip, implementing a new render system, An entity manager class, a Scene class, and creating two scenes, we should be back to where we begun. The game will start to the menu scene, where you should see the text "Almost Pacman" Drawn. Pressing Space will take you into the game scene, where our player and 4 ghosts will be on screen and moving around. Pressing Tab will take us back to the menu. Pressing Escape will close the game down.

Make sure you have got here, and everything is working so far without any errors. Things are going to get a bit wild next. You should commit your code now.

4.3 The Entity Component Model

It's time to bring in the big-guns, standard inheritance and OO can only get us so far. Having a huge inheritance tree for all of our entities would become infeasible to write and maintain, and such we will now adopt the ECM pattern. This will be covered a lecture, here we will focus only on the implementation.

4.3.1 The ECM Library

The code we are about to write will be generic in nature, and we will want to sue it again, so we will spin it out to it's own library.

In case you've forgotten, here's the CMake: Remember to add it to the linked libraries of our lab executable too.

```
1 file(GLOB_RECURSE SOURCE_FILES lib_ecm/*.cpp engine/lib_ecm/*.h)
2 add_library(lib_ecm STATIC ${SOURCE_FILES})
3 target_include_directories(lib_ecm INTERFACE "${CMAKE_SOURCE_DIR}/lib_ecm" )
4 target_link_libraries(lib_ecm PRIVATE lib_maths)
```

Listing 4.10: CMakeLists.txt

Inside the lib_ecm folder we will have ecm.h and ecm.cpp. Move your Entity class that you've already written into the ecm library, and include it in the relevant places in the pacman code. **Check that everything still compiles and works**

Here's what your entity Class should look like:

```
1  #pragma once
2  #include "maths.h"
3  #include <algorithm>
4  #include <memory>
5  #include <typeindex>
6  #include <vector>
7
8  class Component; //forward declare
9
10 class Entity {
11
12 protected:
13   std::vector<std::shared_ptr<Component>> _components;
14   sf::Vector2f _position;
15   float _rotation;
16   bool _alive;       // should be updated
17   bool _visible;     // should be rendered
18   bool _fordeletion; // should be deleted
19 public:
20   Entity();
21   virtual ~Entity();
22   virtual void update(double dt);
23   virtual void render();
24
25   const sf::Vector2f &getPosition() const;
```

```
26    void setPosition(const sf::Vector2f &_position);
27    bool is_fordeletion() const;
28    float getRotation() const;
29    void setRotation(float _rotation);
30    bool isAlive() const;
31    void setAlive(bool _alive);
32    void setForDelete();
33    bool isVisible() const;
34    void setVisible(bool _visible);
```

Listing 4.11: ecm.h

You should have already had most of this, there are a couple of additions you may not of had. Note there is no property for a sprite or a shape. What's new is the declaration of a Component, and a vector of components stored privately.

4.3.2 Component

The component code is remarkably simple.

```
1    class Component {
2    protected:
3      Entity *const _parent;
4      bool _fordeletion; // should be removed
5      explicit Component(Entity *const p);
6
7    public:
8      Component() = delete;
9      bool is_fordeletion() const;
10     virtual void update(double dt) = 0;
11     virtual void render() = 0;
12     virtual ~Component();
13   };
```

Listing 4.12: ecm.cpp

When a component is constructed, an Entity must be passed to the constructor. This is so each component knows who it's parent is. Other than that, it's just our usual two friendly update and render() functions again. There is also a _fordeletion flag, we'll come back to this later.

4.3.3 Sprite component

Before we add the rest of the functionality it would be useful to work with an example of a component. Here is a ShapeComponent. Add it to the Pacman code for now. While it may be a good idea to have some generic components in the ecm library, we're not sure what we are going to need in the future. So we will keep components in the paceman code for now and have the library just be the definitions for the base Entity and Component.

We'll talk about that template in a bit.

```cpp
class ShapeComponent : public Component {
protected:
  std::shared_ptr<sf::Shape> _shape;
public:
  ShapeComponent() = delete;
  explicit ShapeComponent(Entity *p);

  void update(double dt) override;
  void render() override;

  sf::Shape &getShape() const;

  template <typename T, typename... Targs>
  void setShape(Targs... params) {
    _shape.reset(new T(params...));
  }
};
```

Listing 4.13: cmp_sprite.h

There's nothing funky in the definition .cpp. Components are remarkably simple when built correctly.

```cpp
void SpriteComponent::update(double dt) {
  _sprite->setPosition(_parent->getPosition());
}

void ShapeComponent::update(double dt) {
  _shape->setPosition(_parent->getPosition());
}

void ShapeComponent::render() { Renderer::queue(_shape.get()); }

sf::Shape& ShapeComponent::getShape() const { return *_shape; }

ShapeComponent::ShapeComponent(Entity* p)
    : Component(p), _shape(make_shared<sf::CircleShape>()) {}
```

Listing 4.14: cmp_sprite.cpp

4.3.4 Adding a component

So how do we add a shape component to an entity? There are many different approaches to this, the key is to to remember this happens at runtime. Components can be dynamically added and removed to entities. Therefore some of the usual methods you may think won't work.

The approach we will take is to go down (but not too far) the templated code route. Take a gander at this crazy thing:

```cpp
template <typename T, typename... Targs>
std::shared_ptr<T> addComponent(Targs... params) {
    static_assert(std::is_base_of<Component,T>::value,"must be a component");
    std::shared_ptr<T> sp(std::make_shared<T>(this, params...));
    _components.push_back(sp);
    return sp;
}
```

Listing 4.15: ecm.h

This is added to the Entity class in ecm.h.

This is called like so:

```cpp
auto s = ghost->addComponent<ShapeComponent>();
s->setShape<sf::CircleShape>(12.f);
```

We use templates here to do four major things.

1. Create a component of any <specified> type (line 4)

2. Check that the specified type is actually a component (line 3)

3. Pass any parameters to the component constructor. (line 4)

4. Add the built component into the entity component list (line 5)

The extra template we had in the ShapeComponent is unrelated to this process. That one just allows us to set the shape type with templates, rather than having a different shape component for each type of sf::shape.

Putting this to use It's time to kill off our original Entity classes for Ghosts and the player that were in the pacman code. We may need some of the code in there, so instead of deleting the files, just change any *#includes* pointing to them to point to ecm.h instead.

Creating Entities now follows this process:

```cpp
#define GHOSTS_COUNT 4
...
void GameScene::load() {
...
  {
    auto pl = make_shared<Entity>();

    auto s = pl->addComponent<ShapeComponent>();
    s->setShape<sf::CircleShape>(12.f);
    s->getShape().setFillColor(Color::Yellow);
    s->getShape().setOrigin(Vector2f(12.f, 12.f));

    _ents.list.push_back(pl);
  }

  const sf::Color ghost_cols[]{{208, 62, 25},    // red Blinky
                               {219, 133, 28},   // orange Clyde
                               {70, 191, 238},   // cyan Inky
                               {234, 130, 229}}; // pink Pinky

  for (int i = 0; i < GHOSTS_COUNT; ++i) {
    auto ghost = make_shared<Entity>();
    auto s = ghost->addComponent<ShapeComponent>();
    s->setShape<sf::CircleShape>(12.f);
    s->getShape().setFillColor(ghost_cols[i % 4]);
    s->getShape().setOrigin(Vector2f(12.f, 12.f));

    _ents.list.push_back(ghost);
  }
...
```

Listing 4.16: pacman.cpp

This should be all we need to get the game running again, but with one problem - things aren't moving any more.

4.3.5 Building More components

We've got a shape component that let's things be drawn. We need game logic and movement next.

Actor Movement Component For moving things around we will define 3 components. A base "Actor Movement" Component that has the generic methods and properties such as Move() and _speed. From there we will inherit to two seperate components "PlayerMovement-Component" and "EnemyAIComponent". The first will contain the keyboard controls to move the play, the second will contain the AI for the ghosts.

I'll give you the complete listing for the base Component:

```cpp
#pragma once
#include <ecm.h>

class ActorMovementComponent : public Component {
protected:
  bool validMove(const sf::Vector2f&);
  float _speed;

public:
  explicit ActorMovementComponent(Entity* p);
  ActorMovementComponent() = delete;

  float getSpeed() const;
  void setSpeed(float _speed);

  void move(const sf::Vector2f&);
  void move(float x, float y);

  void render() override {}
  void update(double dt) override;
};
```

Listing 4.17: cmp_actor_movement.h

Some lines are missing from the definition, for you to fill in.

```cpp
#include "cmp_actor_movement.h"
#include <LevelSystem.h>

using namespace sf;

void ActorMovementComponent::update(double dt) {}

ActorMovementComponent::ActorMovementComponent(Entity* p)
    : _speed(100.0f), Component(p) {}

bool ActorMovementComponent::validMove(const sf::Vector2f& pos) {
  return (LevelSystem::getTileAt(pos) != LevelSystem::WALL);
}

void ActorMovementComponent::move(const sf::Vector2f& p) {
  auto pp = _parent->getPosition() + p;
  if (validMove(pp)) {
    ...
  }
}

void ActorMovementComponent::move(float x, float y) {
  move(Vector2f(x, y));
}
float ActorMovementComponent::getSpeed() const { ... }
void ActorMovementComponent::setSpeed(float speed) { ... }
```

Listing 4.18: cmp_actor_movement.cpp

Player Movement Component This is super simple, inherit from ActorMovementComponent and add the usual keyboard controls to the Update();

Enemy AI Component This will get much more complex later on, as we dip into AI topics. For now, place your random movement code into the Update().

Add the new Components Adding the new components we just made to our player and ghosts follows the same principle we use to add the shape component:

```cpp
void GameScene::load() {
...
  {
    auto pl = make_shared<Entity>();
    ...

    pl->addComponent<PlayerMovementComponent>();

    ...
  }

  ...

  for (int i = 0; i < GHOSTS_COUNT; ++i) {
    ...

    ghost->addComponent<EnemyAIComponent>();

    ...
  }
  ...
```

Listing 4.19: pacman.cpp

4.3.6 Checkpoint

That was a big change-up in code design. We are no longer using a single classes for Player and Ghost. Instead we are constructing them from components. You can see how this approach lends itself to modular and "Game-like" design. We can easily add or remove components at runtime to any Entity. A logical extension of this would be to design a "Factory" that constructs pre-set Entities with specific components, rather than doing it in pacman.cpp. That's a job for another day, our way is fine for now.

Make sure you have got here, and everything is working, compiling and running so far without any errors. You should commit your code now. Have a well-earned break.

4.4 Pacman AI

AI will be covered in detail in later labs, for this we will be using a very basic state-machine and super simple path-finding. You may have noticed that we have brought in the level system library within the ActorMovementComponent. You should have already altered the CMake to link against this. We will be using the Level system to feed the Ghost AI with information about the level.

Pacman Level First up, let's get the level loaded and rendered.

```
void GameScene::load() {
...
 ls::loadLevelFile("res/pacman.txt", 25.0f);
...
}

void GameScene::render() {
  ls::render(Renderer::getWindow());
  ...
}
```

Listing 4.20: pacman.cpp

Easily done, thanks to our well built level system. What we can do now is use this to change the spawn positions of our Entities

```
void GameScene::respawn() {
 player->setPosition(ls::getTilePosition(ls::findTiles(ls::START)[0]));
 player->GetCompatibleComponent<ActorMovementComponent>()[0]
           ->setSpeed(150.f);

 auto ghost_spawns = ls::findTiles(ls::ENEMY);
 for (auto& g : ghosts) {
   g->setPosition(
        ls::getTilePosition(ghost_spawns[rand() % ghost_spawns.size()]));
   g->GetCompatibleComponent<ActorMovementComponent>()[0]->setSpeed(100.0f);
 }
 ...
}
```

Listing 4.21: pacman.cpp

This makes use of a new function 'findTiles()' which we haven't written yet, go an implement it into the level system library. Here's the declaration, you must figure out the implementation.

```
static std::vector<sf::Vector2ul> findTiles(TILE);
```

Listing 4.22: LevelSystem.h

With this done, the player should be spawning at the bottom, and the ghosts randomly in the middle. The ActorMovementComponent has a "validMove()" function which should stop both Entity types from moving into a wall. We are getting pretty close to a game now.

4.4.1 Ghost Movement

We could have the Ghosts chasing the player, for now we will do something easier. The ghosts will move along the level continuously, until they reach a corner or junction tile. They will then pick a random direction to turn and move in, they will never turn 180 deg and move back the way they came.

```cpp
class EnemyAIComponent : public ActorMovementComponent {
protected:
  sf::Vector2f _direction;
  enum state { DEADEND, ROAMING, ROTATING, ROTATED };
  state _state;
...
}
```

<div align="center">Listing 4.23: cmp_enemy_ai.h</div>

For this to work we only need to store two additional properties in the ghost component: The current state, and current direction.

```cpp
class EnemyAIComponent : public ActorMovementComponent {
protected:
  sf::Vector2f _direction;
  enum state { DEADEND, ROAMING, ROTATING, ROTATED };
  state _state;
...
}
```

<div align="center">Listing 4.24: cmp_enemy_ai.h</div>

The four states a ghost can be are as follows:

- **ROAMING** - Happily moving along - not at waypoint

- **ROTATING** - Currently Choosing a different direction to move

- **ROTATED** - Keep moving until out of waypoint.

You may think that this is a little over-complicated, why do we need two different rotating states? The reason is that as we are moving as float coordinates – not teleporting from tile to tile, a ghost will be inside a way-point for more than one frame. For this reason we need to store a flag to know that we have already rotated, and should not rotate again until the ghost exits the tile. There are many approaches to this, this is just my implementation, feel free to do something alternative.

Picking a Direction With our states figured out, we now move onto the code that transitions between them, the ghost 'AI'. Each update() the ghost will need ot evaluate if it's time to change state, we will need a few variables to accomplish this (Lines 4 to 13). A switch statement forms the logic flow, starting with the current state. I've left two of the states to complete.

```cpp
static const Vector2i directions[] = {{1, 0}, {0, 1}, {0, -1}, {-1, 0}};

void EnemyAIComponent::update(double dt) {
  //amount to move
  const auto mva = (float)(dt * _speed);
  //Curent position
  const Vector2f pos = _parent->getPosition();
  //Next position
  const Vector2f newpos = pos + _direction * mva;
  //Inverse of our current direction
  const Vector2i baddir = -1 * Vector2i(_direction);
  //Random new direction
  Vector2i newdir = directions[(rand() % 4)];

 switch (_state) {
   case ROAMING:
     if (... )// Wall in front or at waypoint
     {
       .. // start rotate
     } else {
      ... //keep moving
     }
     break;

   case ROTATING:
     while (
       // Don't reverse
        ...
       // and Don't pick a direction that will lead to a wall
        ...
       ) {
        ... // pick new direction
       }
     _direction = Vector2f(newdir);
     _state = ROTATED;
     break;

   case ROTATED:
     //have we left the waypoint?
     if (LevelSystem::getTileAt(pos) != LevelSystem::WAYPOINT) {
        _state = ROAMING; //yes
     }
     move(_direction * mva); //No
     break;
 }
 ActorMovementComponent::update(dt);
}
```

Listing 4.25: cmp_enemy_ai.cpp

4.4.2 Collision

Pacman just isn't Pacman without dangerous ghosts. Detecting when a ghost has collided with the player could be done in a number of places. A well-engineered solution would be to have a "collidable" interface on the player, with each ghost checking itself against the player. Another approach would be to ship this out to a standalone physics and collision system.

The approach we are going to take is the most simple, doing the check in the pacman.cpp Update().

For this to work we need to keep a reference to both the player and the ghosts. the game scene does have an EntityList which contains both, and so we could iterate through that. But wouldn't it just be easier if we did this?

```cpp
vector<shared_ptr<Entity>> ghosts;
shared_ptr<Entity> player;

void GameScene::load() {
...
  {
  ...
    _ents.list.push_back(pl);
    player = pl;
  }

  for (int i = 0; i < GHOSTS_COUNT; ++i) {
  ...
    ghosts.push_back(ghost);
    _ents.list.push_back(ghost);
  }
```

Listing 4.26: pacman.cpp

Sometimes it's best to fall back on the simplest solutions. And here is our collision check.

```cpp
for (auto& g : ghosts) {
  if (length(g->getPosition() - player->getPosition()) < 30.0f) {
    respawn();
  }
}
```

Listing 4.27: pacman.cpp

You'll have to build up the respawn() code to reset everything.

4.4.3 Nibbles

One last thing to add, food for pacman.

We're deviating quite far from the proper game rules and going to have the pickup 'nibbles' speed Pacman(and the ghosts) up. We'll place a small white nibble on every EMPTY tile, and larger blue nibble on all the WAYPOINTs. the larger blue nibbles will speed the eater up more than the white nibbles.
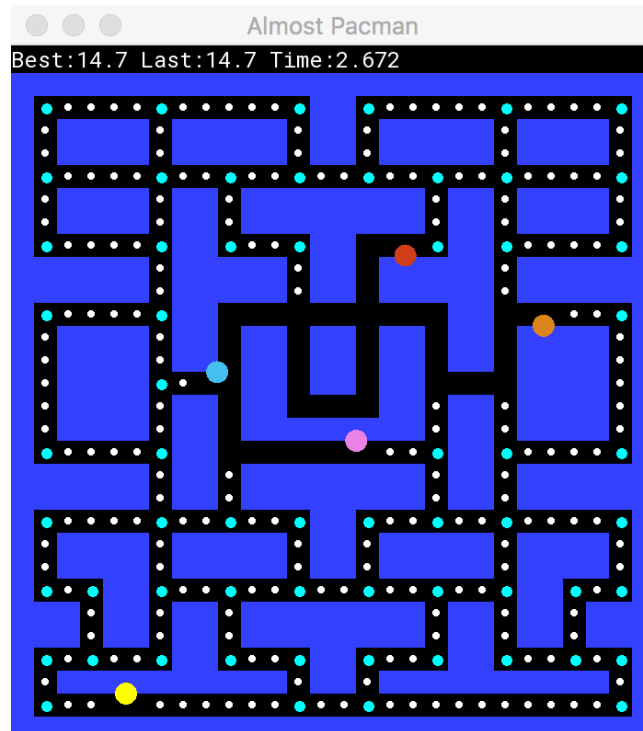
Figure 4.2: **nibble locations**

For this we will need a new component, a PickupComponent.

PickupComponent The class declaration is very simple so it's committed, the only non standard property is a float 'points' which is how much to speed the eater up.

```cpp
void PickupComponent::update(double) {
  for (...) {        //every entity in the scene
    if (...) {       //within 30.f unit of me
      ...            //get the entity ActorMovementComponent, if it has one
      if (...) {     //if it has one
        // nom nom
        ...               //speed the entity up
        _parent->setForDelete(); //delete myself
        break;            //stop looking
      }
    }
  }
}
```

Listing 4.28: cmp_pickup.cpp

.. and here is how we create the nibbles..

```cpp
vector<shared_ptr<Entity>> nibbles;

shared_ptr<Entity> makeNibble(const Vector2ul& nl, bool big) {
    auto cherry = make_shared<Entity>();
    auto s = cherry->addComponent<ShapeComponent>();
    //set colour
    ...

    cherry->addComponent<PickupComponent>(big);
    cherry->setPosition(ls::getTilePosition(nl) + Vector2f(10.f, 10.f));
    return cherry;
}
```

Listing 4.29: pacman.cpp

... and here's where we call that function.

```cpp
void GameScene::respawn() {
    ...
    //clear any remaining nibbles
    for (auto n : nibbles) {
        n->setForDelete();
        n.reset();
    }
    nibbles.clear();

    //white nibbles
    auto nibbleLoc = LevelSystem::findTiles(LevelSystem::EMPTY);
    for (const auto& nl : nibbleLoc) {
        auto cherry = makeNibble(nl, false);
        //add to _wnts and nibbles list
    ...
    }
    //blue nibbles
    nibbleLoc = LevelSystem::findTiles(LevelSystem::WAYPOINT);
    for (const auto& nl : nibbleLoc) {
        ...
    }
    ...
}
```

Listing 4.30: pacman.cpp

4.5 Last steps

All that reamins now is some form of high score system, and we are done. I'll leave this one up to you.

Lesson 5

Physics

5.1 Getting Box2D

We will be using the Box2D physics engine from here on. B2D is reactively robust and well used. Building a 2d physics engine yourself isn't an impossible task, but we don't have time to cover it in this module, so we will be using something that already exists.

When with picking software off the web, chances are it needs some tweaks. B2D doesn't have a well-built CMake Script, but thanks to the process of open-source software, I've fixed it. The fix is still pending in a pull request to the main repo so for now we will use my fork.

Add the Submodule We haven't done this since the very beginning when we added SFML, time to do it again. Open gitbash in the root of your repo

```
1 git submodule add https://github.com/dooglz/Box2D.git lib/b2d
2 git submodule init
3 git submodule update
```

Amend the CMakeLists Adding B2D to our build process is pretty easy:

```
1 # B2D - Box2D phyics library
2 add_subdirectory("lib/b2d/Box2D")
3 #include_directories("lib/b2d/Box2D/")
4 set(B2D_INCS "lib/b2d/Box2D/")
5 link_directories("${CMAKE_BINARY_DIR}/lib/sfml/lib")
```

Listing 5.1: CMakeLists.txt

Then we can just link with "Box2D" and include ${B2D_INCS}.

```
1 ## physics
2 file(GLOB_RECURSE SOURCES 5_physics/*.cpp 5_physics/*.h)
3 add_executable(5_PHYSICS ${SOURCES})
4 target_include_directories(5_PHYSICS SYSTEM PRIVATE ${SFML_INCS} ${B2D_INCS})
5 target_link_libraries(5_PHYSICS Box2D)
6 set(EXECUTABLES ${EXECUTABLES} PRACTICAL_5_PHYSICS)
```

Listing 5.2: CMakeLists.txt

5.2 A standard physics Engine

A physics System/Engine usually has the following components

- A World
  A data-structure that contains all the physics objects in the "world". Usually this also has some global parameters such as "Gravity". Some physics engines allow you to have multiple "worlds". Think of this like a "physics Scene".

- An Integrator
  This is the algorithm that runs each physics 'Tick' or 'Step', to calculate the acceleration, velocity, and position of all bodies in the world. The More 'ticks', the more accurate the simulation is. We usually don't have any control of the inner workings of this.

- Physics Bodies
  Usually called Rigid-bodies (unless dealing with deformable or fluid things). These are things that have mass, inertia, position, and velocity. The physics Integrator moves theses things around based on the rules of physics.

- Colliders
  These are the physical 'shapes' of bodies. e.g cubes, circles, polygons. They determine how two bodies collide with each other. A body is just an abstract 'thing' that has mass. Colliders give them shapes and behaviour.

- Constraints
  Connect Bodies together, either permanently, or based on some form of logic (elastic, ropes, springs, hinges, axles).

The typical process of dealing with a physics engine is as follows

1. Create the world

2. Create Bodies and attach colliders to them

3. Each Update(), step the physics simulation.

4. Update Entity positions to that of the physics bodies.

As you can see, we keep the physics world separate from the Game world. We leave this all to box2D to manage. After a simulation step, we look at the new positions of all the objects in the physics world and copy the new positions to the 'real' world render objects.

Interactivity   So far this works well for an initial scene, but we want interactivity, we want a game. For this we need to feed some gamelogic *into* the physics world.

We are allowed to manually set the position and velocity of any physics body as a cheap "teleport". Doing this isn't great as it breaks the rules of physics that B2D is trying ot stick to. Things don't just teleport in real life. Instead we should use "impulses".

Impulses   These are momentary forces that are applied to a body for one frame. Think of it as giving a thing a little or nudge, or in some cases, strapping rockets to a box for one frame. This is how we will mostly move things around in the physics world, as it obeys the rules of physics. Heavier objects will need a larger impulse force.

Cheating Physics If our game was solely physical bodies moving around realistically, impusles would be all we needed. Unforntauantly in the world of games, we tend to need things that don't *quite* follow *all* the rules of physics.

Think of Super Mario, he can jump and fall and collide with things, which obey the rules of physics. However, Mario never rotates. He also jumps very quickly, to a set height, and then falls down rather slowly. He can move at set speed left and right, he never "accelerates" up to that speed.

Figuring out the correct amount of newtons to impulse Mario by when he jumps seems like a complicated step backwards. We just want him to "jump" like a video game character. This is the folly of Physics engines, they work so hard to give us a near-perfect physical world, only for us to introduce strange limitations and additions to make it feel fun. It can feel at times like the physics system is working against you, a beast to be tamed that really *really* wants to make things go flying off at light-speed (Cite: any Bethesda game).

5.3 Working with Box2D

There are three major factors that we must consider when working with B2D specifically.

1. B2D has it's own Vector maths classes that we must convert to/from

2. B2D's world goes upwards. Positive Y is towards the top of the screen.

3. B2D's world has a 'scale'. We render things in 'pixels'. A sf::box would be 100 'pixels' wide. How much is this in real world units? 100cm, 100m?
   Usually I use 1 unit = 1 meter, when working on 3D games.
   Box2D has a recommended 30'units' per 1'pixel' that feels realistic

Converting between sfml 'screenspace' and b2d 'physics world space' requires taking the above 3 factors into account.

5.3.1 Creating the world

For this practical we will using a single main.cpp approach to get the basics of B2D shown.

```cpp
b2World* world;

void init() {
  const b2Vec2 gravity(0.0f, -10.0f);

  // Construct a world, which holds and simulates the physics bodies.
  world = new b2World(gravity);
  ...
}
```

Listing 5.3: main.cpp

Done, we've just created a world, in 3 lines.

5.3.2 Creating physics Bodies

I'll give you five functions. The first 3 are conversion helper functions to deal with translating between the two worlds. The CreatePhysicsBox() is the biggie, inside is all the B2D logic required to add a body to the scene. The last function in an overload of the fourth, which takes in a sf::RectangleShape rather than a position and size.

```cpp
//Convert from b2Vec2 to a Vector2f
inline const Vector2f bv2_to_sv2(const b2Vec2& in) {
  return Vector2f(in.x * physics_scale, (in.y * physics_scale));
}
//Convert from Vector2f to a b2Vec2
inline const b2Vec2 sv2_to_bv2(const Vector2f& in) {
  return b2Vec2(in.x * physics_scale_inv, (in.y * physics_scale_inv));
}
//Convert from Screenspce.y to physics.y
inline const Vector2f invert_height(const Vector2f& in) {
  return Vector2f(in.x, gameHeight - in.y);
}

//Create a Box3d body with a box fixture
b2Body* CreatePhysicsBox(b2World& World, const bool dynamic,
                         const Vector2f& position, const Vector2f& size) {
  b2BodyDef BodyDef;
  //Is Dynamic(moving), or static(Stationary)
  BodyDef.type = dynamic ? b2_dynamicBody : b2_staticBody;
  BodyDef.position = sv2_to_bv2(position);
  //Create the body
  b2Body* body = World.CreateBody(&BodyDef);

  //Create the fixture shape
  b2PolygonShape Shape;
  Shape.SetAsBox(sv2_to_bv2(size).x * 0.5f, sv2_to_bv2(size).y * 0.5f);
  b2FixtureDef FixtureDef;
  //Fixture properties
  FixtureDef.density = dynamic ? 10.f : 0.f;
  FixtureDef.friction = dynamic ? 0.8f : 1.f;
  FixtureDef.restitution = 1.0;
  FixtureDef.shape = &Shape;
  //Add to body
  body->CreateFixture(&FixtureDef);
  return body;
}

// Create a Box2d body with a box fixture, from a sfml::RectangleShape
b2Body* CreatePhysicsBox(b2World& world, const bool dynamic,
                         const RectangleShape& rs) {
  return CreatePhysicsBox(world, dynamic, rs.getPosition(), rs.getSize());
}
```

Listing 5.4: main.cpp

Let's put it to use, back to that Init() function.

```cpp
std::vector<b2Body*> bodies;
std::vector<RectangleShape*> sprites;
...

void init() {
...
  // Create Boxes
  for (int i = 1; i < 11; ++i) {
    // Create SFML shapes for each box
    auto s = new RectangleShape();
    s->setPosition(Vector2f(i * (gameWidth / 12.f), gameHeight * .7f));
    s->setSize(Vector2f(50.0f, 50.0f));
    s->setOrigin(Vector2f(25.0f, 25.0f));
    s->setFillColor(Color::White);
    sprites.push_back(s);

    // Create a dynamic physics body for the box
    auto b = CreatePhysicsBox(*world, true, *s);
    // Give the box a spin
    b->ApplyAngularImpulse(5.0f, true);
    bodies.push_back(b);
  }
}
```

Listing 5.5: main.cpp

So we are creating 10 boxes - both as sfml::RectangleShapes and b2d::bodies, and storing them both in global vectors. Now we just need to keep them in sync. Can you guess what's coming next?

5.3.3 Updating physics Bodies

This is a two step process, 1: Stepping the physics world, and then copying the data from the bodies to the sf::shapes.

```cpp

void Update() {
  static sf::Clock clock;
  float dt = clock.restart().asSeconds();
  // Step Physics world by Dt (non-fixed timestep)
  world->Step(dt, velocityIterations, positionIterations);

  for (int i = 0; i < bodies.size(); ++i) {
    // Sync Sprites to physics position
    sprites[i]->setPosition(
        invert_height(bv2_to_sv2(bodies[i]->GetPosition())));
    // Sync Sprites to physics Rotation
    sprites[i]->setRotation((180 / b2_pi) * bodies[i]->GetAngle());
  }
}
```

Listing 5.6: main.cpp

5.3.4 Walls

At the moment our boxes just fall into the abyss. Let's put some walls in. Back to Init() for one last time. We will create 4 walls. The position and size of each will be store continuously in a vector that we will loop through. I'll let you figure out the full details. In the end it should look like this:

```cpp
void init() {
  ...
  // Wall Dimensions
  Vector2f walls[] = {
   // Top
   Vector2f(gameWidth * .5f, 5.f), Vector2f(gameWidth, 10.f),
   // Bottom
   Vector2f(gameWidth * .5f, gameHeight - 5.f), Vector2f(gameWidth, 10.f),
   // left
   Vector2f(5.f, gameHeight * .5f), Vector2f(10.f, gameHeight),
   // right
   Vector2f(gameWidth - 5.f, gameHeight * .5f), Vector2f(10.f, gameHeight)
  };

  // Build Walls
  for (int i = 0; i < 7; i += 2) {
    // Create SFML shapes for each wall
    ...
    sprites.push_back(s);
    // Create a static physics body for the wall
    ...
  }
  // Create Boxes
  ...
}
```

Listing 5.7: main.cpp

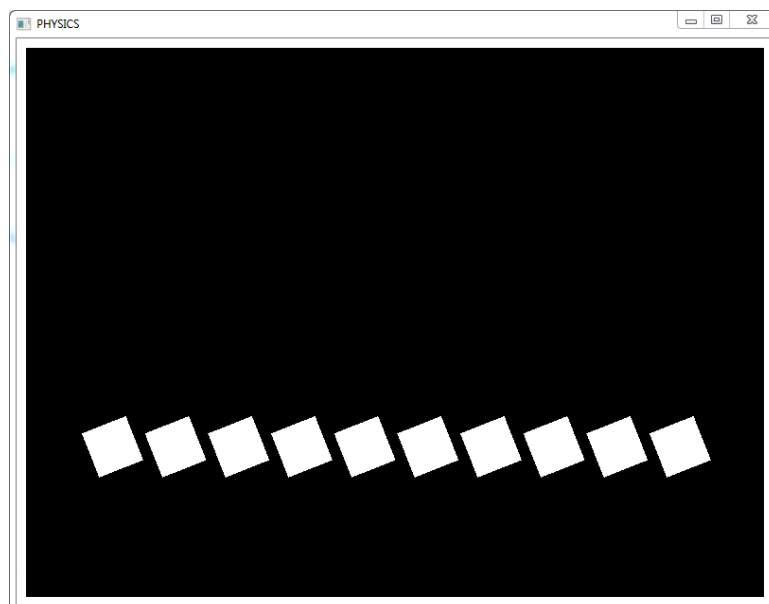

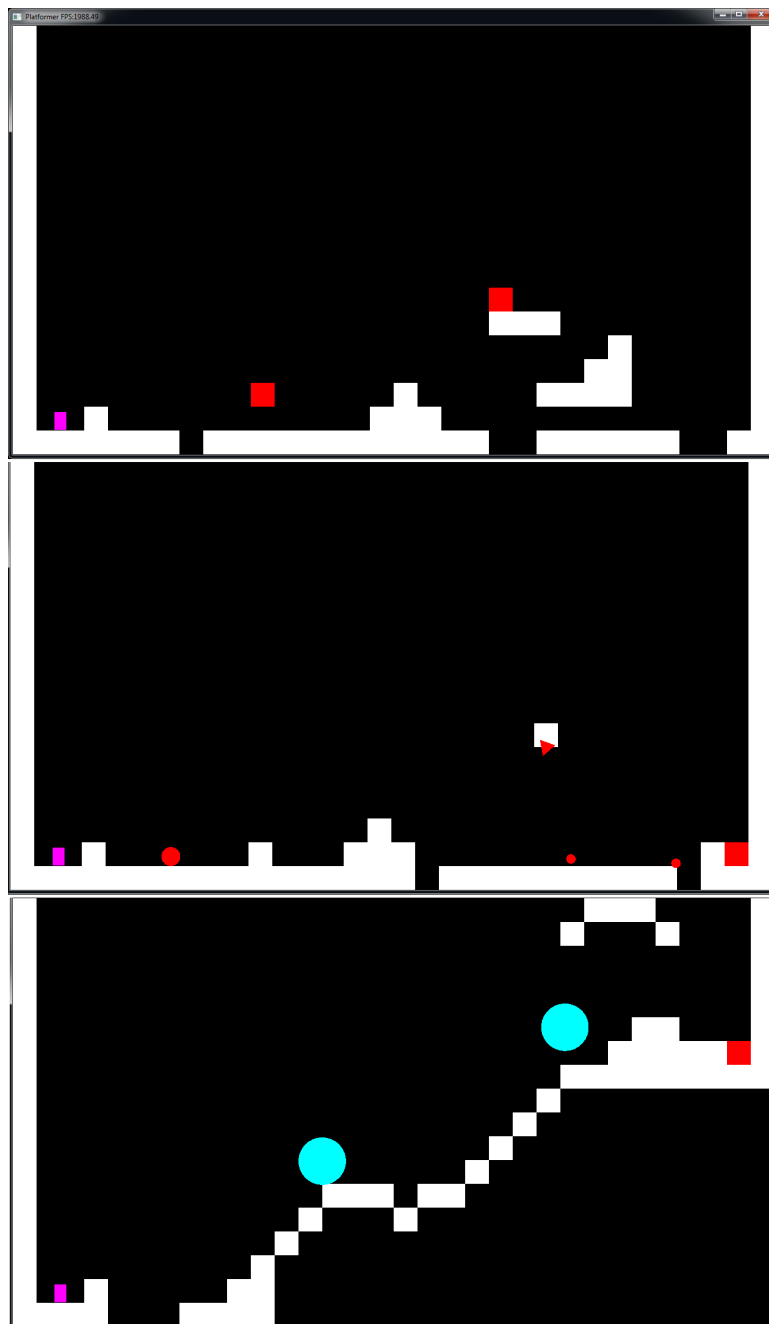Figure 5.1: **Completed Physics Demo**

Lesson 6

Platformer

Figure 6.1: **Completed platformer Game**

6.1 Getting Started

Get the Code There is a lot of code behind this practical, it would not be a good use of your time to get you to copy it all manually, so for this practical. You will be provided with most of the code, with some sections left for you to complete.

Get it here : **https://github.com/edinburgh-napier/set09121**

You don't need to fork the repo or anything special, just download it in some way and then copy the code over to your usual lab repo.

The code you have been given contains:

- The skeleton code for lab6

- The three libraries we have already written. Ecm, LevelLoader, Maths.

- A new Library, the "Engine". (Fully complete, no code to edit)

- My completed CMakeLists.txt - for you to check against

The Engine The Engine library is built upon the knowledge that we gained Working on Pacman. Where we started to separate out the engine logic from the game logic and build an abstracted API to interface with SFML. Most of the code within the engine should be familiar and understandable to you, it's mostly code we have written before. There are some scary new things, which we will cover in due time, like asynchronous level loading.

Don't see the engine code as something you should shy away from just because I've given it to you completed. Have a read, get stuck in, poke about and change some things.

Changes There have been some changes to the other libraries, some bugfixes, and some small additions that wouldn't have made sense when we first built them. However it was necessary to do some tweaks to help build the engine. This is why I have also given you the code to the three helper libraries. Feel free to copy them wholesale to keep in sync.

Summary of big changes

- ECM: Entities can have String Tags associated with them, scenes can be searched for entities with tags.

- LevelSystem: A compression stage happens before generating sprites. This groups similar tiles into one large sprite. This help massively with rendering and physics performance.

- LevelSystem: Can now read, parse, and lookup any character from a level file. The ENUM is still there and working to give helpful names to certain values

6.2 The Game Loop

Let's step through our new execution environment, starting with platformer_main.cpp

```cpp
#include "engine.h"
#include "game.h"

using namespace std;

MenuScene menu;
Level1Scene level1;
Level2Scene level2;
Level3Scene level3;

int main() {
    Engine::Start(1280, 720, "Platformer",&menu);
}
```

Listing 6.1: platformer_main.cpp

Wow, now that's simple. Obviously we have 'magic'd' away all the work we used to have to do. It's all down in the Engine.cpp. The point here is that we are in **Game** code, which shouldn't need to care about the implementation underneath. Game code shouldn't care if we are using SFML or some other rendering framework, game code should only care about Game logic.

In practise there isn't this perfect abstraction, we still include plenty of SFML headers throughout the game code. We could replace all this with an engine abstraction layer, but that would be over-engineering and possibly a performance hit. It's that balance of clean-software engineering versus optimal and fast code again, only your experience building these systems can guide you to making the right architecture decisions.

Let's take a peak at what that Engine::Start call did down in the Engine:

```cpp
void Engine::Start(unsigned int width, unsigned int height,
                   const std::string& gameName, Scene* scn) {
  RenderWindow window(VideoMode(width, height), gameName);
  _gameName = gameName;
  _window = &window;
  Renderer::initialise(window);
  Physics::initialise();
  ChangeScene(scn);
  while (window.isOpen()) {
   // Usual Game loop stuff
  }
  if (_activeScene != nullptr) {
    _activeScene->UnLoad();
    _activeScene = nullptr;
  }
  window.close();
  Physics::shutdown();
}
```

Listing 6.2: engine.cpp

There's our usual game loop! It's never that far away.

Some new init and shut-down code has been added, and we have _activeScene to keep track of the current level that should be updated and rendered.

That ChangeScene call is new, let's take a look at what that does:

```cpp
void Engine::ChangeScene(Scene* s) {
  cout << "Eng: changing scene: " << s << endl;
  auto old = _activeScene;
  _activeScene = s;

  if (old != nullptr) {
    old->UnLoad();
  }

  if (!s->isLoaded()) {
    cout << "Eng: Entering Loading Screen\n";
    loadingTime =0;
    _activeScene->LoadAsync();
    loading = true;
  }
}
```

Listing 6.3: engine.cpp

What we have here is a system to switch scenes, first it un-loads the current scene, then loads the new scene. Scenes have two loading methods, Load() and LoadAsync(). Let's talk about that.

6.2.1 Scene Loading

Synchronous loading Scene.Load() Does what you would expect. It's a normal function doing normal things for normal people. It's basically a constructor for a scene, that can be called repeatedly (note: scene's constructors are deleted, you can't call them). A scene shouldn't be Updated() or Rendered() until Load() has completed and Scene.isLoaded() == true;

Asynchronous loading Scene.LoadAsync() Shifts us into another dimension of software engineering: Concurrency. Simply put, this function calls the standard Load() function, *but in the background*. This means we can do something else while it is loading, like update and render a loading screen!

There is a "thread::sleep_for(3 seconds)" inside Scene1.load(), to simulate a big scene.

Games and Concurrency This is a big and complicated topic, and we have a separate module just for this

Concurrent and Parallel Systems

This is just a small toe-dip into the topic, and the code is already all written for you. What we are dealing with here is Multithreading. Using more than just one of those CPU cores that we have to gain us some more performance.

Multi-threading for games is a bit tricky, some things can be parallelised easy(background asset loading), some things are almost impossible(Updating a complex set of interconnected Entities). It's not a silver bullet to give us more FPS. But for developers on current-gen consoles, it's a must-have to squeeze out every last drop of performance out of the hardware.

Checking up on the background task While the level is loading in the background, we need to know when it is finished. For this I have used a std::future guarded by a std::mutex inside Scene::isLoaded(). Don't worry about what that means or how that works for now, just know why that code is there.

Debugging Important tip!

If your game crashes or throws an exeption while loading Asynchronously, it can be difficult to debug. In this case, switch out the line in Engine::ChangeScene from

_activeScene->LoadAsync();

to

_activeScene->Load();

To temporarily disable background loading (Loading screen will never show, and game will hang as the scene loads).

6.3   Scene 1

With *how* Levels are loaded covered, take a look at scene_level1.cpp. This is where our Game Kicks off (scene_menu is first, but that's boring).

All the work is in the Load(). The Update() does very little – only checking to see if the player has reached the end. This is the beautify of the Entity Component Model. Just build our level from your box of parts(components) and the game logic makes itself (almost Emergent behaviour).

Figure 6.2: **Platformer Level 1**

Components   We have some familiar components in use: ShapeComponent, which is unchanged. The two new components on the scene are PhysicsComponent and it's child: PlayerPhysicsComponent. This is how we are going to integrate Box2d into our engine, as components.

There is a PhysicsSystem located in the engine/system_physics.cpp. This doesn't do much other than house some maths functions and the world.Step(); Most of our Box2D code lies in the components.

6.3.1   Physics Component

The base PhysicsComponent in cmp_physics.cpp is perhaps our most complex component yet. Many of it's function are getters and setters to interface correctly with SFML and Box2d maths, and as an interface to change physics properties.

The constructor does the same logic that we had in the physics practical, creating wither a static or dynamic body.

Then there are some new functions such as isTouching() and getTouching(), which we can use to interface gamelogic with collisions. Frustratingly the B2D API flips interchangeably between C and C++.

92

```
//This is why some of the B2D code is the C++ way:

_fixture->getSomething()->doSomething();

// and some code is bit oldschool

Thing thing;
_fixture->getSomething(&thing);
thing.doSomething();
```

Keep a lookout fo this when dealing with B2D directly, and take care with raw pointers.

Movement Gotcha Take a look at the Update():

```
void PhysicsComponent::update(double dt) {
  _parent->setPosition(invert_height(bv2_to_sv2(_body->GetPosition())));
  _parent->setRotation((180 / b2_pi) * _body->GetAngle());
}
```

This is how the physics world is linked to the SFML/Entity world. This happens every frame. There is one huge problem with this:

- The PhysicsComponent is now in charge of the Entities position.

- If anything modifies the Entities position (i.e another component), the physics component will overwrite this change with it's own value copied form the physics world.

We Can still move Entities manually, by calling PhysicsComponent.teleport(), but we have to *know* that an Entity has a physics object beforehand. This get's complex if we wanted this logic inside a component. What we have done is made certain component *incompatible*. I haven't included a solution to this in the current Engine, as we don't need it. For your coursework you may run into this problem, there are a few solutions, but I'll leave that to you to figure out.

So that's the Physics Component, it does a lot, but nothing complicated. It's Child Player-PhysicsComponent is where things get weird.

6.3.2 Player Physics Component

Inheriting from PhysicsComponent, the PlayerPhysicsComponent is what drives the player.

Inside you will find:

- A constructor - Set's some relevant physics flags, nothing fancy

- A rather complicated Update()

- bool isGrounded() - detect if the player is standing on something.

The Player Update As described in section 5.2, players characters don't obey the rules of physics. They *look* like they do, but they cheat and bend physics to make the game *feel* responsive and fast.

See: Tommy Refenes' on the physics of Super Meat Boy

To make the player *feel* right, we do several things:

- Dampen X Movement if not moving left or right (essentially apply handbrake to player if no keys are pressed)

- Apply impulse to move left or right, only if current going slower than a max Velocity.

- Jump, if only on the ground.

- kill all Y velocity at the start of the Jump

- Jump teleports slightly upwards first, then apply an impulse

- If not on the ground - player has no friction

- After Everything - clamp velocity to a maximum value

None of these things are an industry standard, it a method that I've adopted after doing this a few times. There are better ways, but this way *works*, but may not work for your game.

6.3.3 Run The Scene

You shouldn't need to edit anything to get the Menu and first level running. Give it a go and bounce around.

6.4 Scene 2

There is some code missing form the scene_level2.cpp file for you to complete.

This scene incorporates a non-physics moving abject (using EnemyAIComponent), and a cool new Turret (EnemyTurretComponent) that fires physical bullets. There are two more new components in use here:

- The BulletComponent is just a countdown timer that deletes _parent when it reaches 0;

- The HurtComponent, check for (fake) collision with the player and kills the player.
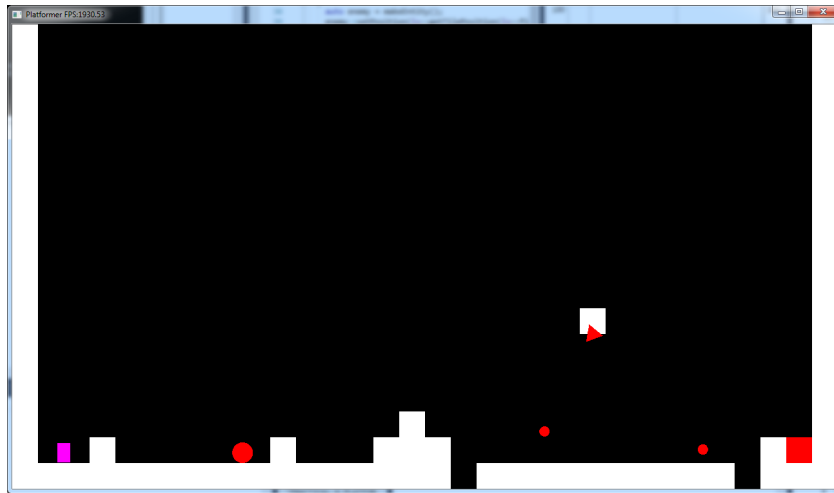
Once completed, it should look like this:

Figure 6.3: **Platformer Level 2**

6.5 Scene 3

The same goes for Scene 3. This scene is easier than Scene 2, just the player and physics components to add.
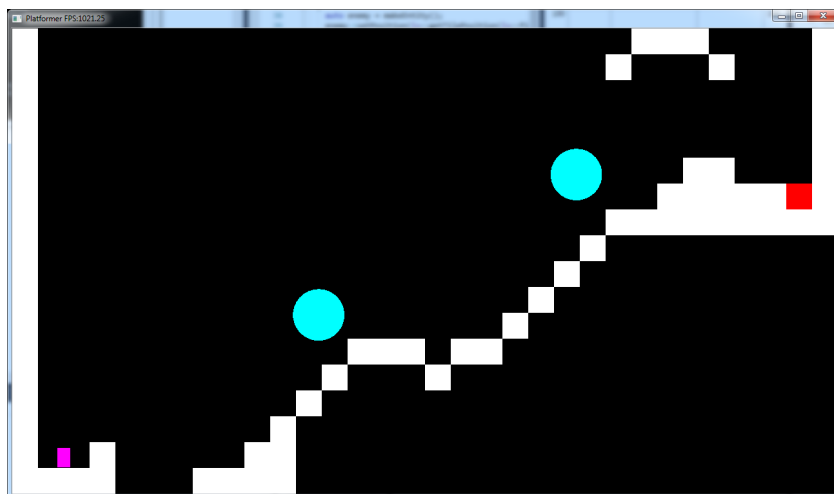
Figure 6.4: **Platformer Level 3**

Lesson 7

AI: Steering and Pathfinding

We are now going to start the AI labs. The first one will focus on movement and the decision making around movement. This will involve defining steering behaviours and pathfinding. This lab will use the engine as it stands at the end of practical 6. That is, we will be using *scenes* and *components* to build our solution. You will need the standard set up for a new lab. Our addition to the CMakeLists.txt is as follows.

```
1 file(GLOB_RECURSE SOURCES practical_7_steering/*.cpp practical_7_steering/*.↩
    h)
2 file(GLOB_RECURSE CMPNTS practical_7_steering/components/*.cpp ↩
    practical_7_steering/components/*.h)
3 file(GLOB_RECURSE SCENES practical_7_steering/scenes/*.cpp ↩
    practical_7_steering/scenes/*.h)
4 add_executable(PRACTICAL_7_STEERING ${SOURCES} ${CMPNTS} ${SCENES} ${↩
    RESOURCE_FILES})
5 target_link_libraries(PRACTICAL_7_STEERING lib_engine)
6 set(EXECUTABLES ${EXECUTABLES} PRACTICAL_7_STEERING)
```

Listing 7.1: CMakeLists.txt

You will also need the components from practical 6 copied across.

7.1   Steering Behaviours

We are going to split the lab into two parts. First we will focus on steering behaviours. We will need to do some setup first to get a menu system up and running and using that to drive our high-level game logic.

7.1.1   Setting Up

The first item we will add is a `MenuSystem` class. First we will add a `scene_menu.h` file to our `scenes` folder of our project. This is below. The definition is just a standard scene.

```
1 #pragma once
2
3 #include "engine.h"
4
```

```cpp
class MenuScene : public Scene
{
public:
    MenuScene() = default;
    ~MenuScene() = default;

    void Load() override;

    void Update(const double &dt) override;
};
```

Listing 7.2: scene_menu.h

Our implementation (`scene_menu.cpp` in the `scene` folder) is below.

```cpp
#include "scene_menu.h"
#include "../components/cmp_text.h"
#include "../game.h"
#include <SFML/Window/Keyboard.hpp>

using namespace std;
using namespace sf;

void MenuScene::Load()
{
    auto txt = makeEntity();
    auto t = txt->addComponent<TextComponent>("Movement Demos\nPress 1 for ←
        Steering\nPress 2 for Pathfinding");
    setLoaded(true);
}

void MenuScene::Update(const double &dt)
{
    if (sf::Keyboard::isKeyPressed(Keyboard::Num1))
    {

    }
    else if (sf::Keyboard::isKeyPressed(Keyboard::Num2))
    {

    }
    Scene::Update(dt);
}
```

Listing 7.3: Starting scene_menu.cpp

The `MenuScene` will display a `TextComponent` which will give two options:

1. Steering Behaviour Demo

2. Pathfinding Demo

The `Update` checks which button is pressed and has the outline for the code to handle this change. Once we have our other scenes in place we will add the new lines of code.

We will use a `game.h` file to allow easy access to the scenes. This is given below.

```
1  #pragma once
2
3  #include "scenes/scene_menu.h"
4
5  extern MenuScene menu;
```

Listing 7.4: Starting game.h

And finally we need our `main.h` file. This is below.

```
1  #include "engine.h"
2  #include "game.h"
3
4  using namespace std;
5
6  MenuScene menu;
7
8  int main()
9  {
10     Engine::Start(1280, 720, "Steering", &menu);
11     return 0;
12 }
```

Listing 7.5: Starting main.h

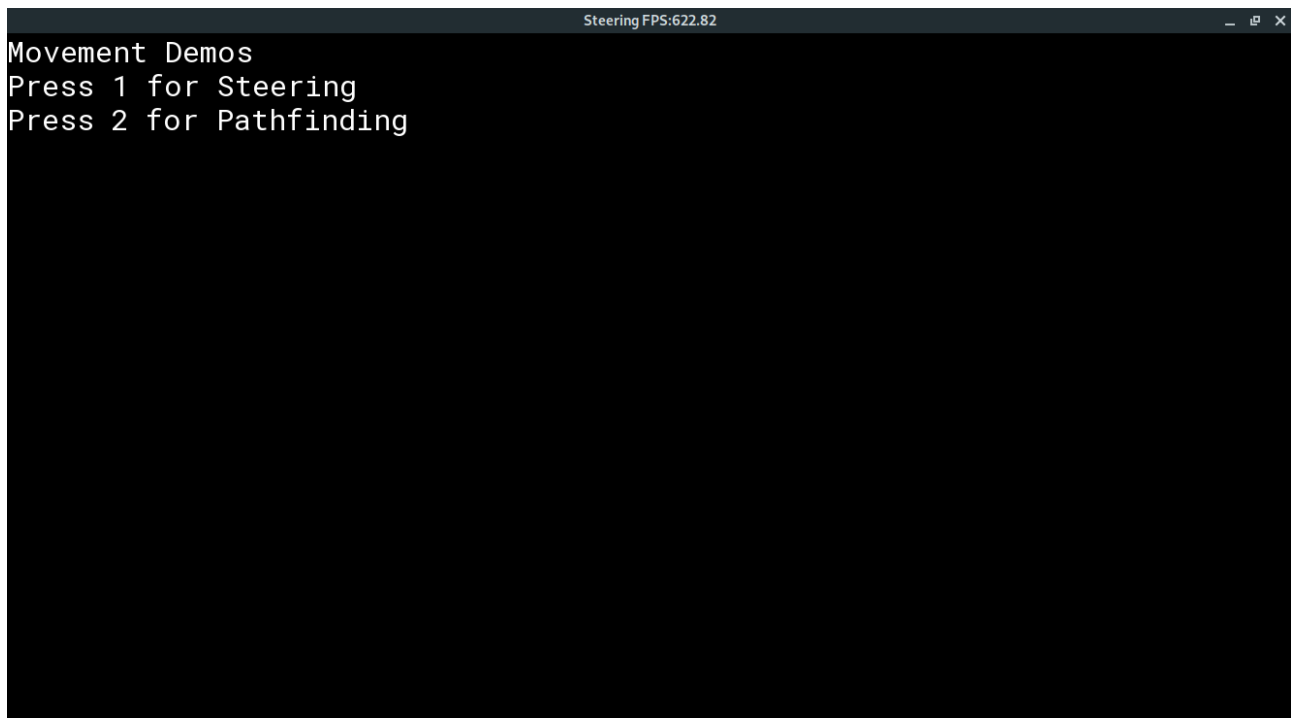You should be able to run this now and get the output shown in Figure 7.1.

Figure 7.1: Steering Lab Menu

SteeringScene

Before moving onto steering behaviours properly let us set up our `SteeringScene`. We are going to have an object on the screen (represented by a red dot) that the player will move

around using the arrow keys. To do this we will create a basic `BasicMovementComponent`. You can replace this with the physics movement component from last practical if you wish. The `cmp_basic_movement.h` and `cmp_basic_movement.cpp` files are provided below. You should be able to understand the basic premise by now.

```cpp
#pragma once

#include <ecm.h>

// A component to allow basic movement behaviour
class BasicMovementComponent : public Component
{
protected:
    // Speed we can travel
    float _speed;
    // Checks if the move is valid.
    bool validMove(const sf::Vector2f&);
public:
    // Will check the keyboard and move the component's parent.
    void update(double) override;
    // Moves the component's parent.
    void move(const sf::Vector2f&);
    // Moves the component's parent.
    void move(float x, float y);
    // Component does not need rendered.
    void render() override { }
    // Used to create the component for an entity
    explicit BasicMovementComponent(Entity *p);

    BasicMovementComponent() = delete;
};
```

Listing 7.6: cmp_basic_movement.h

```cpp
#include "cmp_basic_movement.h"
#include <SFML/Window/Keyboard.hpp>
#include <engine.h>

using namespace sf;
using namespace std;

// Checks keyboard and moves the player.
void BasicMovementComponent::update(double dt)
{
    Vector2f direction(0.0f, 0.0f);
    if (Keyboard::isKeyPressed(Keyboard::Left))
    {
        direction.x -= 1.0f;
    }
    if (Keyboard::isKeyPressed(Keyboard::Right))
    {
        direction.x += 1.0f;
    }
    if (Keyboard::isKeyPressed(Keyboard::Up))
    {
```

```
22          direction.y -= 1.0f;
23      }
24      if (Keyboard::isKeyPressed(Keyboard::Down))
25      {
26          direction.y += 1.0f;
27      }
28
29      move(normalize(direction) * _speed * (float)dt);
30  }
31
32  // Initialises the component.
33  BasicMovementComponent::BasicMovementComponent(Entity *p)
34  : _speed(100.0f), Component(p)
35  {
36  }
37
38  // Checks if the proposed move is valid.
39  bool BasicMovementComponent::validMove(const sf::Vector2f& pos)
40  {
41      if (pos.x < 0.0f
42          || pos.x > Engine::GetWindow().getSize().x
43          || pos.y < 0.0f
44          || pos.y > Engine::GetWindow().getSize().y)
45          return false;
46      else
47          return true;
48  }
49
50  // Moves the component's parent.
51  void BasicMovementComponent::move(const sf::Vector2f &p)
52  {
53      auto new_pos = _parent->getPosition() + p;
54      if (validMove(new_pos))
55      {
56          _parent->setPosition(new_pos);
57      }
58  }
59
60  // Moves the component's parent.
61  void BasicMovementComponent::move(float x, float y)
62  {
63      move(Vector2f(x, y));
64  }
```

Listing 7.7: cmp_basic_movement.cpp

Our `SteeringScene` will add an `Entity` with this component attached. We first need to define the `scene_steering.h` file which is given below.

```
1  #pragma once
2
3  #include "engine.h"
4
5  class SteeringScene : public Scene
```

```
 6 {
 7 public:
 8     void Load() override;
 9     void UnLoad() override;
10     void Update(const double &dt) override;
11     void Render() override;
12 };
```

Listing 7.8: scene_steering.h

The key behaviour we can add at the moment will be defined in the `Load` method. Here we will create an `Entity` and add a `BasicMovementComponent`. `scene_steering.cpp` is given below.

```
 1 #include "scene_steering.h"
 2 #include "../components/cmp_sprite.h"
 3 #include "../components/cmp_basic_movement.h"
 4 #include <LevelSystem.h>
 5 #include <random>
 6 #include <chrono>
 7
 8 using namespace std;
 9 using namespace std::chrono;
10 using namespace sf;
11
12 void SteeringScene::Load()
13 {
14     auto player = makeEntity();
15     player->setPosition(Vector2f(Engine::GetWindow().getSize().x / 2, Engine↩
          ::GetWindow().getSize().y / 2));
16     auto s = player->addComponent<ShapeComponent>();
17     s->setShape<CircleShape>(10.0f);
18     s->getShape().setFillColor(Color::Red);
19     player->addComponent<BasicMovementComponent>();
20 }
21
22 void SteeringScene::UnLoad()
23 {
24     Scene::UnLoad();
25 }
26
27 void SteeringScene::Update(const double &dt)
28 {
29     Scene::Update(dt);
30 }
31
32 void SteeringScene::Render()
33 {
34     Scene::Render();
35 }
```

Listing 7.9: Starting scene_steering.cpp

All we have to do now is update `game.h` as below.

```cpp
#pragma once

#include "scenes/scene_menu.h"
#include "scenes/scene_steering.h"

extern MenuScene menu;
// New line
extern SteeringScene steeringScene;
```

Listing 7.10: Updated game.h

Update `MenuScene::Update` as below.

```cpp
void MenuScene::Update(const double &dt)
{
    if (sf::Keyboard::isKeyPressed(Keyboard::Num1))
    {
        // New line
        Engine::ChangeScene(&steeringScene);
    }
    else if (sf::Keyboard::isKeyPressed(Keyboard::Num2))
    {
    }
    Scene::Update(dt);
}
```

Listing 7.11: Updated MenuScene::Update method

And then add a line `SteeringScene steeringScene;` to `main.cpp`. You should now be able to run the game, select option 1 and control the red dot as shown in Figure 7.2.
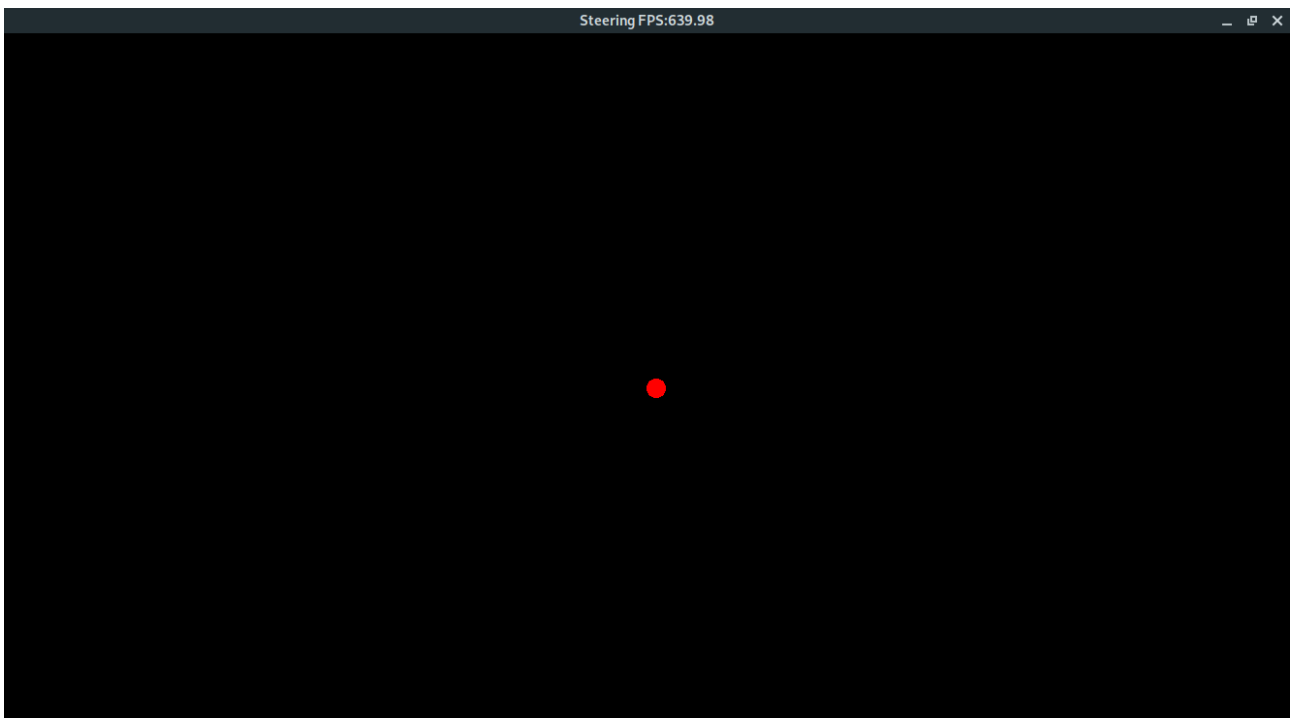
Figure 7.2: Player on Screen

7.1.2 SteeringOutput

As discussed in the lecture, the first part we need is the `SteeringOutput` type. We will define this in `steering.h`. This is given below.

```cpp
#pragma once

#include <engine.h>

// Output from a steering behaviour.
struct SteeringOutput
{
    // Direction of travel.
    sf::Vector2f direction;
    // Rotation of travel.
    float rotation;
};
```

Listing 7.12: Starting steering.h

We defined these values in the lecture. They allow us to move or rotate on the basis of a steering behaviour.

7.1.3 SteeringBehaviour

`SteeringBehaviour` we also defined in the lecture. This is a `virtual` class. The important part is the `getSteering` method. Add the below to the `steering.h` file.

```cpp
// Base class for steering behaviour
class SteeringBehaviour
{
public:
    virtual ~SteeringBehaviour() = default;

    // Gets the output from a steering behaviour.
    virtual SteeringOutput getSteering() const noexcept = 0;
};
```

Listing 7.13: SteeringBehaviour Definition – add to steering.h

We will build two steering behaviours – seek and flee. They are fairly easy as the following sections will show.

Seek

`Seek` is the behaviour to move an object to a target. We described the general idea in the lecture. The following is the definition to add to `steering.h`.

```cpp
// Seek steering behaviour
class Seek : public SteeringBehaviour
{
private:
    Entity* _character;
```

```
 6      Entity* _target;
 7      float _maxSpeed;
 8      public:
 9      Seek() = delete;
10      Seek(Entity *character, Entity *target, float maxSpeed)
11      : _character(character), _target(target), _maxSpeed(maxSpeed) { }
12      SteeringOutput getSteering() const noexcept;
13  };
```

Listing 7.14: Seek Behaviour – add to steering.h

The only part we need to define is the `Seek::getSteering` method. We will do this in a new file – `steering.cpp`. This is given below.

```
 1  #include "steering.h"
 2
 3  using namespace sf;
 4
 5  SteeringOutput Seek::getSteering() const noexcept
 6  {
 7      SteeringOutput steering;
 8      steering.direction = _target->getPosition() - _character->getPosition();
 9      steering.direction = normalize(steering.direction);
10      steering.direction *= _maxSpeed;
11      steering.rotation = 0.0f;
12      return steering;
13  }
```

Listing 7.15: Implementation of Seek::getSteering – add to steering.cpp

Flee

Flee is likewise simple to implement. We add the declaration to `steering.h` and then the definition to `steering.cpp`. These are given below.

```
 1  class Flee : public SteeringBehaviour
 2  {
 3  private:
 4      Entity* _character;
 5      Entity* _target;
 6      float _maxSpeed;
 7      public:
 8      Flee() = delete;
 9      Flee(Entity *character, Entity *target, float maxSpeed)
10      : _character(character), _target(target), _maxSpeed(maxSpeed) { }
11      SteeringOutput getSteering() const noexcept;
12  };
```

Listing 7.16: Flee Behaviour – add to steering.h

```
 1  SteeringOutput Flee::getSteering() const noexcept
 2  {
 3      SteeringOutput steering;
 4      steering.direction = _character->getPosition() - _target->getPosition();
```

```
5    steering.direction = normalize(steering.direction);
6    steering.direction *= _maxSpeed;
7    steering.rotation = 0.0f;
8    return steering;
9 }
```

Listing 7.17: Implementation of Flee::getSteering – add to steering.cpp

7.1.4 SteeringComponent

We will now add a new component that will control the AI movement. This we will call SteeringComponent and we will define it in cmp_ai_steering.h as below.

```
1  #pragma once
2
3  #include <ecm.h>
4  #include "../steering.h"
5
6  class SteeringComponent : public Component
7  {
8  protected:
9      Seek _seek;
10     Flee _flee;
11     Entity *_player;
12     bool validMove(const sf::Vector2f&) const;
13 public:
14     void update(double) override;
15     void move(const sf::Vector2f&);
16     void move(float x, float y);
17     void render() override { }
18     explicit SteeringComponent(Entity *p, Entity *player);
19     SteeringComponent() = delete;
20 };
```

Listing 7.18: cmp_ai_steering.h

The component is similar to the other movement ones we have defined and you could reuse those as well if you wanted. The key behaviour is really in update. This is defined in cmp_ai_steering.cpp (as well as other methods). See this below.

```
1  #include "cmp_ai_steering.h"
2
3  using namespace sf;
4
5  void SteeringComponent::update(double dt)
6  {
7      // If target (player) is more than 100 pixels away seek
8      if (length(_parent->getPosition() - _player->getPosition()) > 100.0f)
9      {
10         auto output = _seek.getSteering();
11         move(output.direction * (float)dt);
12     }
13     // If target (player) is less than 50 pixels away flee
```

```cpp
14      else if (length(_parent->getPosition() - _player->getPosition()) < 50.0f
            )
15      {
16          auto output = _flee.getSteering();
17          move(output.direction * (float)dt);
18      }
19  }
20
21  SteeringComponent::SteeringComponent(Entity *p, Entity *player)
22  : _player(player), _seek(Seek(p, player, 100.0f)), _flee(Flee(p, player,
        100.0f)), Component(p)
23  {
24  }
25
26  bool SteeringComponent::validMove(const sf::Vector2f& pos) const
27  {
28      if (pos.x < 0.0f
29          || pos.x > Engine::GetWindow().getSize().x
30          || pos.y < 0.0f
31          || pos.y > Engine::GetWindow().getSize().y)
32          return false;
33      else
34          return true;
35  }
36
37  void SteeringComponent::move(const sf::Vector2f &p)
38  {
39      auto new_pos = _parent->getPosition() + p;
40      if (validMove(new_pos))
41      {
42          _parent->setPosition(new_pos);
43      }
44  }
45
46  void SteeringComponent::move(float x, float y)
47  {
48      move(Vector2f(x, y));
49  }
```

Listing 7.19: cmp_ai_steering.cpp

The key behaviour in `update` is a check on how far the `entity` (`_parent`) is from the player. If it is more than 100 pixels the component will use seek. If it is less than 50 it will flee. Now all we need to do is add some enemies with this behaviour. We do this back in the `SteeringScene::Load` method as below.

```cpp
1  void SteeringScene::Load()
2  {
3      auto player = makeEntity();
4      player->setPosition(Vector2f(Engine::GetWindow().getSize().x / 2, Engine
            ::GetWindow().getSize().y / 2));
5      auto s = player->addComponent<ShapeComponent>();
6      s->setShape<CircleShape>(10.0f);
7      s->getShape().setFillColor(Color::Red);
```

```
 8    player->addComponent<BasicMovementComponent>();
 9
10    // New lines for here.
11    // Setup C++ random number generation
12    random_device dev;
13    default_random_engine engine(dev());
14    uniform_real_distribution<float> x_dist(0.0f, Engine::GetWindow().↩
          getSize().x);
15    uniform_real_distribution<float> y_dist(0.0f, Engine::GetWindow().↩
          getSize().y);
16    for (size_t n = 0; n < 100; ++n)
17    {
18        auto enemy = makeEntity();
19        enemy->setPosition(Vector2f(x_dist(engine), y_dist(engine)));
20        auto s = enemy->addComponent<ShapeComponent>();
21        s->setShape<RectangleShape>(Vector2f(10.0f, 10.0f));
22        s->getShape().setFillColor(Color::Blue);
23        enemy->addComponent<SteeringComponent>(player.get());
24    }
25 }
```

Listing 7.20: Updated SteeringScene::Load

You can now run the application and you will have the enemies chase and run away from you accordingly. Figure 7.5 illustrates this.
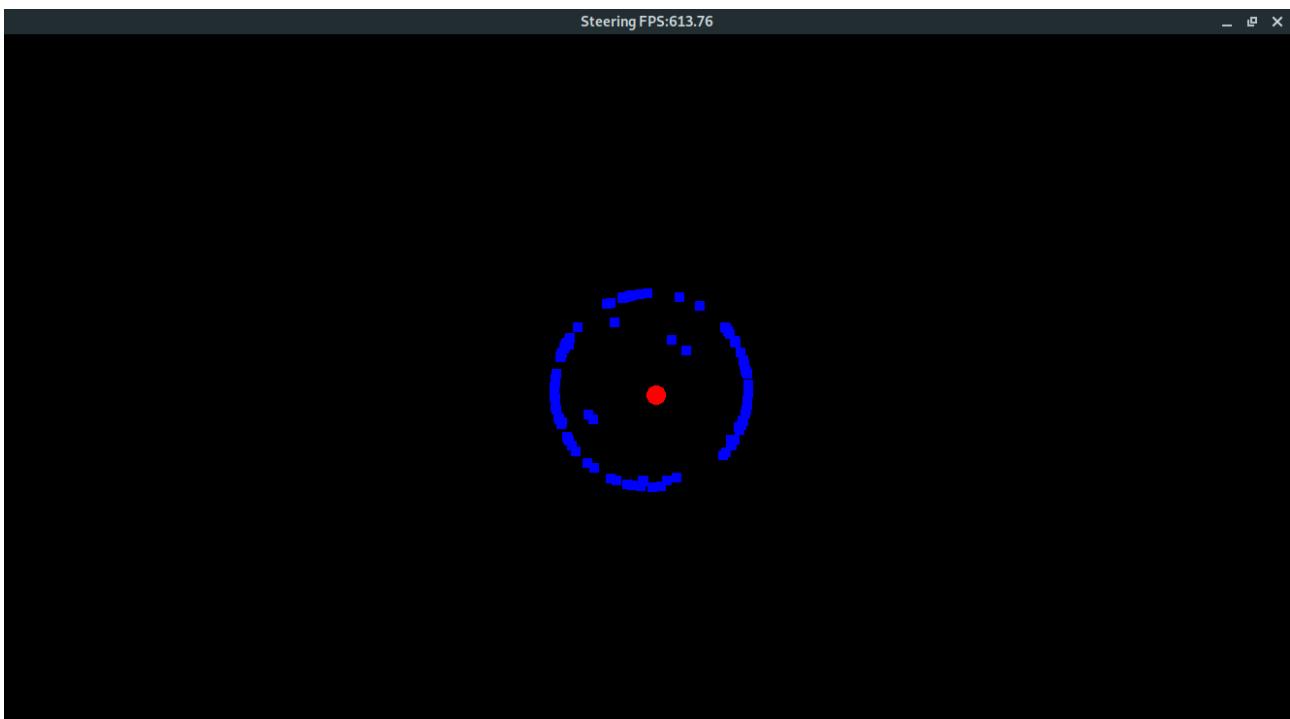

Figure 7.3: Final Steering Behaviour Output

7.2 Pathfinding

Now that the steering behaviour scene is working we can focus on pathfinding. Our aim here is to use the level system to provide information so we can tell a character how to move around a map. This requires implementation of the A* algorithm. This will provide information we can send to a component so it can decide how to move. We will end by adding controls so you can click on the map to change the target to move towards.

7.2.1 PathFindingScene

First we need to implement the `PathFindingScene`. The starting files are defined below and go into `scene_pathfinding.h` and `scene_pathfinding.cpp`.

```cpp
#pragma once

#include <engine.h>

class PathfindingScene : public Scene
{
public:
    void Load() override;
    void UnLoad() override;
    void Update(const double &dt) override;
    void Render() override;
};
```

<div align="center">Listing 7.21: scene_pathfinding.h</div>

```cpp
#include "scene_pathfinding.h"

#include <LevelSystem.h>
#include "../components/cmp_sprite.h"

using namespace std;
using namespace sf;

shared_ptr<Entity> character;

void PathfindingScene::Load()
{
    ls::loadLevelFile("res/pacman.txt", 20.0f);

    character = makeEntity();
    Vector2f startPos(ls::getOffset().x + 1 * ls::getTileSize(), ls::←
        getOffset().y + 1 * ls::getTileSize());
    character->setPosition(startPos);
    auto s = character->addComponent<ShapeComponent>();
    s->setShape<CircleShape>(10.0f);
    s->getShape().setFillColor(Color::Red);
}

void PathfindingScene::UnLoad()
```

```
24 {
25      Scene::UnLoad();
26 }
27
28 void PathfindingScene::Update(const double &dt)
29 {
30      Scene::Update(dt);
31 }
32
33 void PathfindingScene::Render()
34 {
35      ls::render(Engine::GetWindow());
36      Scene::Render();
37 }
```

Listing 7.22: Starting scene_pathfinding.cpp

The `Load` method just loads a tile map (I will use the one from PacMan) and add a red circle to the screen. We then just update `game.h` to below.

```
1 #pragma once
2
3 #include "scenes/scene_menu.h"
4 #include "scenes/scene_steering.h"
5 #include "scenes/scene_pathfinding.h"
6
7 extern MenuScene menu;
8 extern SteeringScene steeringScene;
9 extern PathfindingScene pathfindingScene;
```

Listing 7.23: Updated game.h

Update `MenuScene::update` so it will select the new `PathFindingScene`.

```
1 void MenuScene::Update(const double &dt)
2 {
3      if (sf::Keyboard::isKeyPressed(Keyboard::Num1))
4      {
5          Engine::ChangeScene(&steeringScene);
6      }
7      else if (sf::Keyboard::isKeyPressed(Keyboard::Num2))
8      {
9          Engine::ChangeScene(&pathfindingScene);
10     }
11     Scene::Update(dt);
12 }
```

Listing 7.24: Updated MenuScene::update

And add the line `PathfindingScene pathfindingScene;` to our declarations in `main.cpp`. Then you should run the application and get the output shown in Figure 7.4.
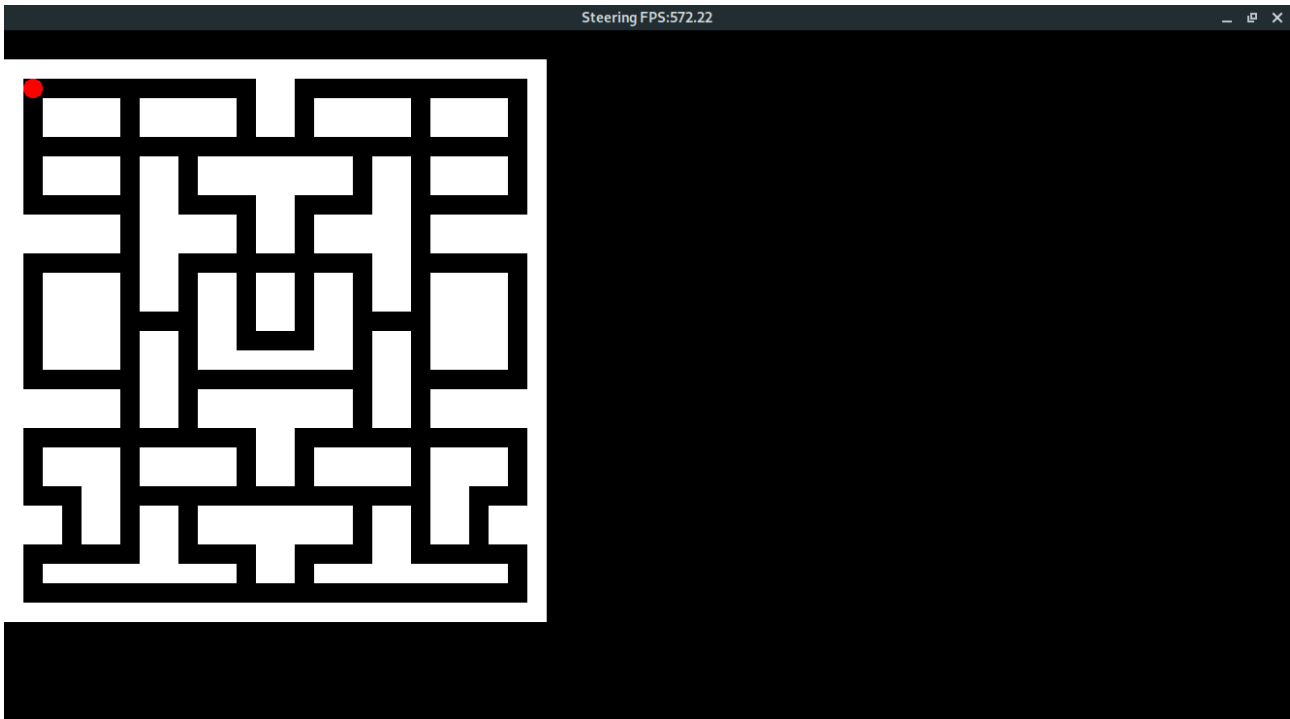
Figure 7.4: Start of Pathfinding Demo

7.2.2 Implementing A*

We discussed the basics of A* in the lecture. We will define these in the files `astar.h` and `astar.cpp`. The latter is a long definition but the comments and the lecture slides will help you understand what is happening.

```
#pragma once

#include <engine.h>
#include <vector>

std::vector<sf::Vector2i> pathFind(sf::Vector2i start, sf::Vector2i finish);
```

Listing 7.25: astar.h

```
#include "astar.h"
#include <queue>
#include <LevelSystem.h>

using namespace std;
using namespace sf;

// Node in the search graph.
class Node
{
private:
    Vector2i _pos;
    int _level;
    int _priority;
public:
    Node() = default;
```

```cpp
17    Node(const Vector2i &pos, int level, int priority)
18    : _pos(pos), _level(level), _priority(priority)
19    {
20    }
21
22    const Vector2i& getPos() const { return _pos; }
23
24    int getLevel() const { return _level; }
25
26    int getPriority() const { return _priority; }
27
28    unsigned int estimate(const Vector2i &dest) const
29    {
30        // Heuristic is just Euclidian distance.
31        // Can be modified.
32        Vector2i delta = dest - _pos;
33        return static_cast<unsigned int>(length(delta));
34    }
35
36    void updatePriority(const Vector2i &dest)
37    {
38        _priority = _level + estimate(dest) * 10;
39    }
40
41    void nextLevel()
42    {
43        _level += 10;
44    }
45
46    // Used for priority ordering
47    bool operator<(const Node &other) const
48    {
49        return _priority > other._priority;
50    }
51 };
52
53 vector<Vector2i> pathFind(Vector2i start, Vector2i finish)
54 {
55    static array<Vector2i, 4> directions =
56    {
57        Vector2i(1, 0),
58        Vector2i(0, 1),
59        Vector2i(-1, 0),
60        Vector2i(0, -1)
61    };
62
63    // This may not be the most efficient mechanism, but should not be a ↵
           problem
64    // unless your maps get very big.
65    vector<vector<bool>> closed_nodes_map(ls::getWidth());
66    vector<vector<int>> open_nodes_map(ls::getWidth());
67    vector<vector<Vector2i>> direction_map(ls::getWidth());
68    // Queue of nodes to test.  Priority ordered.
```

```
69      // We need two for when we redirect and copy the path.
70      priority_queue<Node> queue[2];
71      // Index of current queue
72      size_t queue_index = 0;
73
74      // Initialise the lookup maps.  Initially everything looks traversable ↩
            and
75      // no route has been planned.
76      for (size_t y = 0; y < ls::getHeight(); ++y)
77      {
78          for (size_t x = 0; x < ls::getWidth(); ++x)
79          {
80              closed_nodes_map[x].push_back(false);
81              open_nodes_map[x].push_back(0);
82              direction_map[x].emplace_back(Vector2i(0, 0));
83          }
84      }
85
86      // Add start node to the queue - we will check it first
87      Node n0(start, 0, 0);
88      // Update the priority of the node.
89      n0.updatePriority(finish);
90      // Push onto the current queue of nodes to check.
91      queue[queue_index].push(n0);
92      // And update the value in the open_nodes_map.
93      open_nodes_map[start.x][start.y] = n0.getPriority();
94
95      // Loop until queue is empty
96      while (!queue[queue_index].empty())
97      {
98          auto tmp = queue[queue_index].top();
99          n0 = Node(tmp.getPos(), tmp.getLevel(), tmp.getPriority());
100         auto pos = n0.getPos();
101
102         // Remove node from open list
103         queue[queue_index].pop();
104         open_nodes_map[pos.x][pos.y] = 0;
105         // Mark on closed node list
106         closed_nodes_map[pos.x][pos.y] = true;
107
108         // Check if reached goal
109         if (pos == finish)
110         {
111             vector<Vector2i> path;
112             while (!(pos == start))
113             {
114                 // Get the direction
115                 auto dir = direction_map[pos.x][pos.y];
116                 // Add the current position to the path
117                 path.push_back(pos);
118                 pos += dir;
119             }
120             // Path is inverted.  Reverse it.
```

```
121          reverse(begin(path), end(path));
122          // Return path.
123          return path;
124       }
125       // Not reached goal. Check four directions.
126       for (auto &dir : directions)
127       {
128          auto next = pos + dir;
129
130          // Check if next is valid.
131          if (!(next.x < 0 || next.x > ls::getWidth() || next.y < 0 || ↩
                next.y > ls::getHeight()
132             || ls::getTile(Vector2ul(next.x, next.y)) == LevelSystem::↩
                   WALL
133             || closed_nodes_map[next.x][next.y]))
134          {
135             // Generate new node
136             Node m0(next, n0.getLevel(), n0.getPriority());
137             m0.nextLevel();
138             m0.updatePriority(finish);
139
140             // Check if new node has no priority
141             if (open_nodes_map[next.x][next.y] == 0)
142             {
143                // Update priority and add to the queue.
144                open_nodes_map[next.x][next.y] = m0.getPriority();
145                queue[queue_index].push(m0);
146                // Put the opposite direction into the direction map.
147                // We work backwards.
148                direction_map[next.x][next.y] = dir * -1;
149             }
150             // If it has a priority, check if the priority is better on ↩
                   the new route.
151             else if (open_nodes_map[next.x][next.y] > m0.getPriority())
152             {
153                // Update the priority information
154                open_nodes_map[next.x][next.y] = m0.getPriority();
155                // Update the direction map with the inverse direction.
156                direction_map[next.x][next.y] = dir * -1;
157                // Now replace the node via swapping across the lists ↩
                      and ignoring the old one.
158                while (queue[queue_index].top().getPos() != next)
159                {
160                   queue[1 - queue_index].push(queue[queue_index].top()↩
                         );
161                   queue[queue_index].pop();
162                }
163                queue[queue_index].pop();
164                if (queue[queue_index].size() > queue[1 - queue_index].↩
                      size())
165                   queue_index = 1 - queue_index;
166                while (!queue[queue_index].empty())
167                {
```

```
168                         queue[1 - queue_index].push(queue[queue_index].top()↩
                                );
169                         queue[queue_index].pop();
170                     }
171                     queue_index = 1 - queue_index;
172                     queue[queue_index].push(m0);
173                 }
174             }
175         }
176     }
177     return vector<Vector2i>();
178 }
```

Listing 7.26: astar.cpp

Now we just need to add a component to use this behaviour.

7.2.3 PathfindingComponent

PathfindingCompoonent will use a list of nodes to traverse the level. Every second it will move to the next position in the level until it reaches its destination. To do this it needs to keep track of a path, its index in the path, and the amount of time passed since the last move. We will also require a method to set the path. cmp_path_follow.h and cmp_path_follow.cpp are provided below.

```cpp
1 #pragma once
2
3 #include <ecm.h>
4
5 class PathfindingComponent : public Component
6 {
7 protected:
8     std::vector<sf::Vector2i> _path;
9     size_t _index = 0;
10     double _elapsed = 0.0;
11 public:
12     void update(double) override;
13     void render() override { }
14     void setPath(std::vector<sf::Vector2i> &path);
15     explicit PathfindingComponent(Entity *p);
16     PathfindingComponent() = delete;
17 };
```

Listing 7.27: cmp_path_follow.h

```cpp
1 #include "cmp_path_follow.h"
2 #include "../astar.h"
3 #include <LevelSystem.h>
4
5 using namespace sf;
6 using namespace std;
7
8 void PathfindingComponent::update(double dt)
```

```
 9 {
10     _elapsed += dt;
11     if (_elapsed >= 0.1)
12     {
13         _elapsed = 0.0;
14         if (_index < _path.size())
15         {
16             // Could do this in a single line - expanded for clarity
17             float new_x = ls::getOffset().x + _path[_index].x * ls::←
                   getTileSize();
18             float new_y = ls::getOffset().y + _path[_index].y * ls::←
                   getTileSize();
19             _parent->setPosition(Vector2f(new_x, new_y));
20             ++_index;
21         }
22     }
23 }
24
25 PathfindingComponent::PathfindingComponent(Entity *p)
26 : Component(p)
27 {
28 }
29
30 void PathfindingComponent::setPath(std::vector<sf::Vector2i> &path)
31 {
32     _index = 0;
33     _path = path;
34 }
```

Listing 7.28: cmp_path_follow.cpp

We now just update the PathfindingComponent definition so we create a PathfindingComponent on our entity and find a path. We will find a path from the character's start position (1,1) to the bottom left corner. The updates are below.

```
 1 shared_ptr<PathfindingComponent> ai;
 2
 3 void PathfindingScene::Load()
 4 {
 5     ls::loadLevelFile("res/pacman.txt", 20.0f);
 6
 7     character = makeEntity();
 8     Vector2f startPos(ls::getOffset().x + 1 * ls::getTileSize(), ls::←
           getOffset().y + 1 * ls::getTileSize());
 9     character->setPosition(startPos);
10     auto s = character->addComponent<ShapeComponent>();
11     s->setShape<CircleShape>(10.0f);
12     s->getShape().setFillColor(Color::Red);
13
14     // New code from here
15     auto path = pathFind(Vector2i(1, 1), Vector2i(ls::getWidth() - 2, ls::←
           getHeight() - 2));
16     ai = character->addComponent<PathfindingComponent>();
17     ai->setPath(path);
```

```
18 }
```

Listing 7.29: Updated PathFindingScene

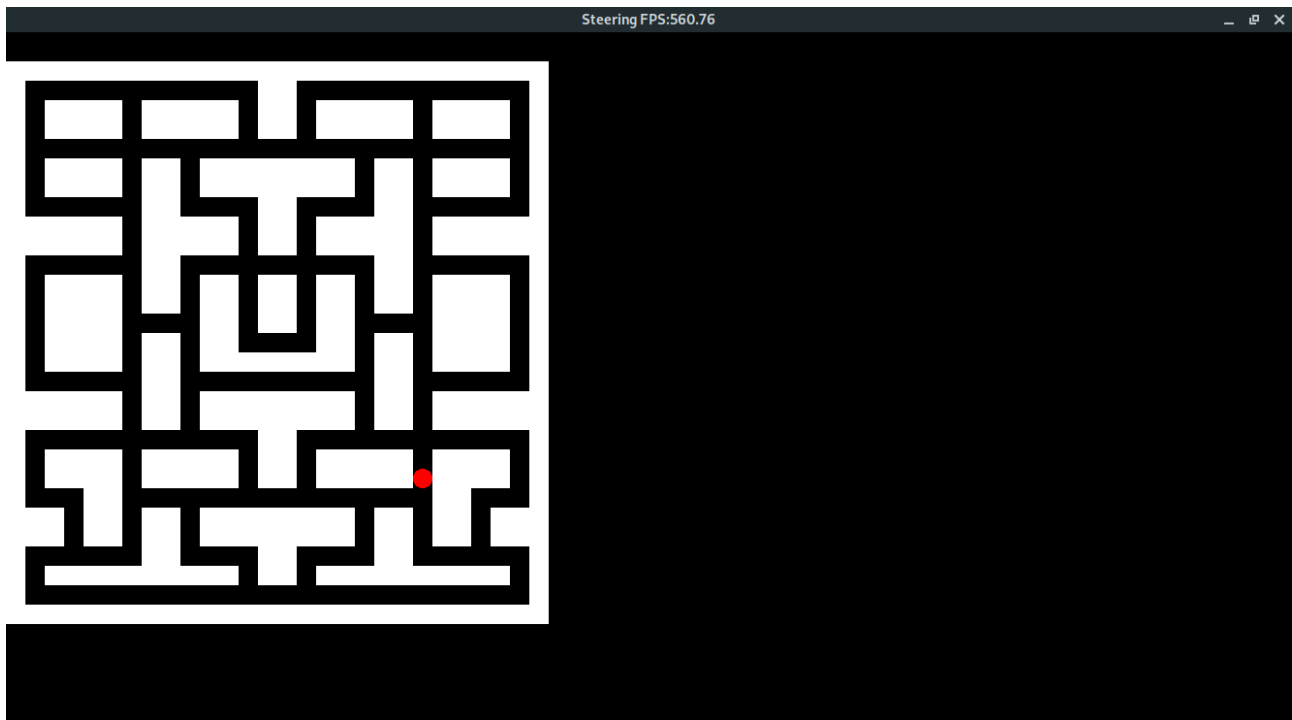If you run the application now the character will pathfind to the destination.

Figure 7.5: Pathfinding in Action

7.2.4 Adding Mouse Input

Let us now modify the update for the `PathfindingScene` to allow the user to click on a tile to change the location to move to. This requires the following stages:

1. Check if mouse button has been pressed since last frame.

2. If it has, check if the mouse click is on the map.

3. If it is, calculate which tile has been clicked.

4. If tile clicked is not a wall, calculate current character tile.

5. Use the current character tile and the clicked tile to perform a pathfind.

The code for this is below.

```
1 void PathfindingScene::Update(const double &dt)
2 {
3     static bool mouse_down = false;
4     if (Mouse::isButtonPressed(Mouse::Left) && !mouse_down)
5     {
6         auto mouse_pos = Mouse::getPosition(Engine::GetWindow());
7         mouse_down = true;
8         if (ls::isOnGrid(Vector2f(mouse_pos)))
9         {
```

```
10          auto relative_pos = mouse_pos - Vector2i(ls::getOffset());
11          auto tile_coord = relative_pos / (int)ls::getTileSize();
12          if (ls::getTile(Vector2ul(tile_coord)) != ls::WALL)
13          {
14              auto char_relative = character->getPosition() - ls::↩
                    getOffset();
15              auto char_tile = Vector2i(char_relative / ls::getTileSize())↩
                    ;
16              auto path = pathFind(char_tile, tile_coord);
17              ai->setPath(path);
18          }
19      }
20  }
21  if (mouse_down && !Mouse::isButtonPressed(Mouse::Left))
22  {
23      mouse_down = false;
24  }
25  Scene::Update(dt);
26 }
```

Listing 7.30: Using Mouse Clicks to Update Path

You should now be able to click on a new tile to change the path being used.

7.3 Exercise

The key thing to try now is to integrate the physics into our movement. Then, try and implement the arrive steering behaviour with pathfinding to create a smoother movement during 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

Compiling c++11 on updated OSX   osx 10.9+ (High Sierra) has different c++ libraries, CMake needs to know to use them.

Place this before the project() call.

```
1   set(CMAKE_OSX_DEPLOYMENT_TARGET 10.9)
```

Listing A.1: cmake

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

```
1  #### Target for copying resources to build dir####
2  if(APPLE)
3    add_custom_target(copy_resources ALL
4      COMMAND ${CMAKE_COMMAND} -E copy_directory
5      "${PROJECT_SOURCE_DIR}/res"
6      ${CMAKE_RUNTIME_OUTPUT_DIRECTORY}/res
7
8      COMMAND ${CMAKE_COMMAND} -E copy_directory
9      "${CMAKE_SOURCE_DIR}/lib/sfml/extlibs/libs-osx/Frameworks"
10     ${CMAKE_BINARY_DIR}/lib/sfml/Frameworks
11   )
12 else()
13   add_custom_target(copy_resources ALL COMMAND ${CMAKE_COMMAND} -E
14   copy_directory
15     "${PROJECT_SOURCE_DIR}/res"
16     ${CMAKE_RUNTIME_OUTPUT_DIRECTORY}/$<CONFIGURATION>/res
17   )
18 endif()
```

Listing A.2: custom target dependancy (including mac)

A.2 C++ Header file tips

Don't forget   Headers aren't magic or smart. The preprocessor just replaces every instance of "#include file.h" with everything that it's in file.h. Headers are nothing more than a copy and paste macro. If you are struggling to figure something out, consider what would be the effects of writing whatever is giving you an error in the .h, directly in all the .cpp files where you include the .h.

- Only Include what you need.

- Only declare items that other files will need.

- Never use "using namespace"'in a header.

- Avoid defining functions in a header.

- Don't forget include guards "#pragma once"

- Dont forget to define declared static members

- Templates must be defined in a header.

A.2.1 Forward declaring - to avoid circular dependencies

In the case where class A in A.h needs to use and include class B in B.h, but class B also needs to use class A, we get a "circular dependency". We can't include both Headers within each other. We can avoid this by doing a forward declaration.

```cpp
//----- A.h -----
class B; // forward declaration

class A
{
public:
    B* myB;
};

//----- B.h -----
class A; // forward declaration

class B
{
public:
A* myA;
};
```

Listing A.3: Foward declare

Note we don't include the Headers of each other within each other. This is similar to the "extern keyword". This works only because we don't need to know anything about the class we are forward declaring, and we are storing it as a pointer. This is because we don't know how big a class B is, but we know how big a class B pointer is, it's just the standard 32/64 bits.

```cpp
class B; // forward declaration

class A
{
public:
B* myBpointer; //works
B myB; //will not work.
};
```

Listing A.4: Foward declare