

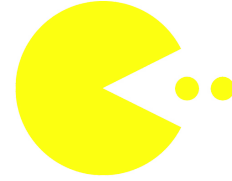


COMS1017: Intro. to Data Structures & Algorithms

Project 1: Pacman (Part 1)

Richard Klein

Due Date: 18:00 on July 30, 2015



1 Introduction

Pacman is an arcade game developed by Namanco in 1980. Scan the QR code to the right or visit the link below it to play the game. This next link will give you some interesting facts about the game: <http://goo.gl/hpjtjl>.

Throughout this course we will cover many different data structures and algorithms. As we cover these algorithms, a series of 3 projects will guide you in implementing the algorithms to construct a working game of Pacman. The first project will focus on C++ revision and cover the development of functions to help render images on the screen and animate the characters.

As we cover different structures, you'll be required to implement and integrate them into the game. By the time the course is complete, you should have a fully working version of Pacman that includes intelligence to control the ghosts.

You are encouraged to add to the game and make it your own. If you ever find that you have too much free time, I recommend going through SDL Tutorials by Lazy Foo (<http://lazyfoo.net/tutorials/SDL/index.php>). They will give you better insight to the structure of 2D graphics programming in SDL and will give you an architecture in which you can start creating your own game.

While the primary outcome of the course is a thorough knowledge of Data Structures and Algorithms, you will find that strong programming skills will assist you throughout your career in CS. Programming is as much of an art as it is a science. Remember that the only good way to become a good programmer is to practice. So think of interesting ways to extend your game and try add them in! Maybe try write your own game of Tetris!

Play Pacman Online



<https://goo.gl/jrNRk0>

2 Background & Tools

2.1 Git & BitBucket

When working in teams or on large projects you are usually required to use a version control system so that everyone knows what version of the code you are editing. These systems then help you keep track of why a change was made to the code (*Commit Message*), as well as who made it (*Author*). They help you keep track of different changes to the code and if you're working on multiple features at the same time, it can be helpful to work on each feature in its own independent *branch*.

Git is a version control system that allows you to track different versions of your code while programming. Git was developed by Linus Torvalds (the same person who started Linux) and is used by millions of programmers around the world and depending on who you listen to between 30%¹ and 70%² of professional programmers use it. We will be using basic the basic functionality of Git in this course.

BitBucket and GitHub are online hosting environments for Git. BitBucket will give you a free, unlimited subscription if you register your Wits email with them. This gives you access to private repositories for your school work. I strongly suggest making use of this.

Git is a command line tool, but there are are plenty of very good and free graphical front ends. I recommend SmartGIT, which is free for non-commercial use, QGit or Git-Gui.

2.1.1 Exercise

1. Go to <https://bitbucket.org> and sign up for a free account. Either signup using your Wits email address, or signup with your own address and then add your Wits to your profile under *Manage Account — Email Addresses*.
 - (a) Select a “Personal Account”
 - (b) Follow the process to prove that you're not a robot.
 - (c) Confirm your email address by clicking on the link in the email that was sent to you.
2. Once you've added and verified your Wits email address visit the BitBucket Academic License Page³ to convert your account to an unlimited academic license.

¹<http://programmers.stackexchange.com/questions/128851/empirical-evidence-of-popularity-of-git-and-mercurial>

²<http://stackoverflow.com/research/developer-survey-2015#tech-sourcecontrol>

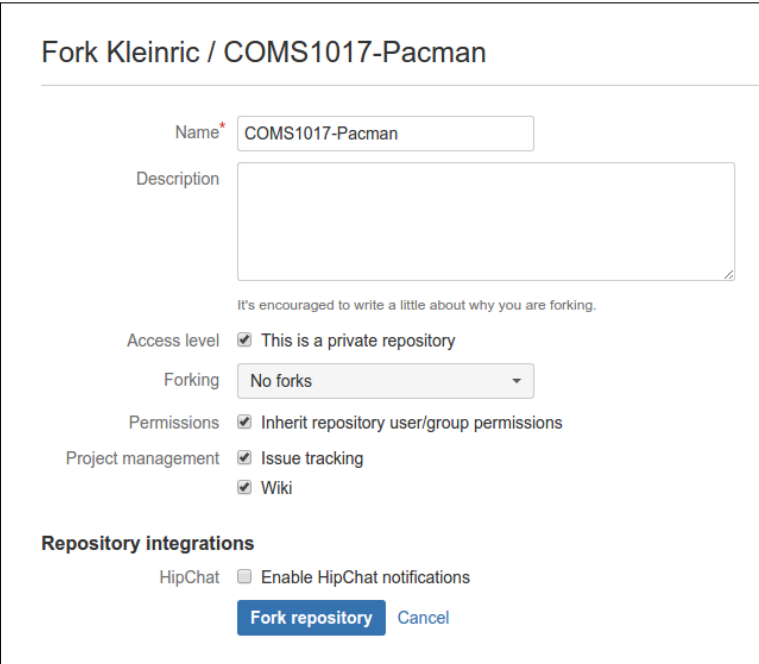
³<https://www.atlassian.com/software/views/bitbucket-academic-license.jsp>

3. Using Google, read about Git and teach yourself how to use the following Git commands: `init`, `add`, `commit`, `log`, `checkout`, `clone` and `push`.⁴
4. <https://goo.gl/CGgFjt> gives a really good Cheat Sheet, with a list of useful Git commands and their syntax.

2.2 Get the project files & Setup your repository

2.2.1 Fork the Project on BitBucket

Make sure you are logged in to BitBucket in your browser. Browse to <https://bitbucket.org/Kleinric/coms1017-pacman>, on the left hand side there is an action called *Fork*, click on it. This will allow you to create a repository based on the code that I have already provided. Make sure that you check the box next to “Access Level” saying that “This is a private repository.” If you don’t, then anyone will have access to your code and you’ll probably be flagged for plagiarism. Select “No Forks” and the repository should “Inherit repository user/group permissions.” The settings should match those shown below in Figure 1.



The screenshot shows the 'Fork Kleinric / COMS1017-Pacman' form in BitBucket. It includes fields for 'Name' (COMS1017-Pacman) and 'Description'. Below these are settings for 'Access level' (checked: 'This is a private repository'), 'Forking' (dropdown: 'No forks'), 'Permissions' (checked: 'Inherit repository user/group permissions'), 'Project management' (checked: 'Issue tracking' and 'Wiki'), and 'Repository integrations' (HipChat: 'Enable HipChat notifications' is unchecked). At the bottom are 'Fork repository' and 'Cancel' buttons.

Figure 1: Settings for your Forked Repo

This will create a private repository on under your account that is a copy of the one on my account. Whenever you have completed some work on your project you should push your changes back to this repo so that you always have an up to date backup on the BitBucket servers.

⁴Remember you can always look up the syntax of the command when you need it. Just make sure you know what each command does.

2.2.2 Clone the Repository

Now that you have your own private version of the code, you need to download the code so that you have a local copy on your machine. The idea is that you then edit this local copy, committing your changes as you go along, and then push the changes back to BitBucket.

Use Git to checkout the project files from your fork. If you browse to your new repository in the browser you'll see on the right hand side there is a link to your project as seen in Figure 2.

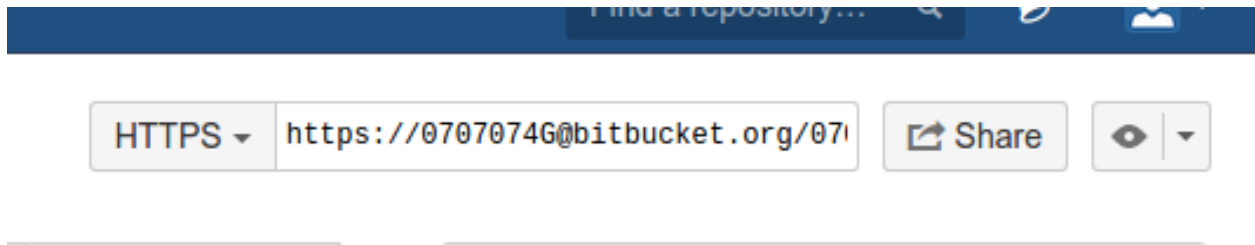


Figure 2: Copy the address of your repository.

To get a local copy of your project, open a terminal and run:

```
git clone https://<address to your repo>
```

Where the address is the one that the site shows you similar to the image above, but with your username. You should now have a copy of the code on your computer.

2.2.3 Add, Commit, Push & Get New Code

Now that you have a copy of the repository on your machine, open the file called `readme.txt` and add your name and student number to the top of it. In the terminal, `cd` to the folder with your code. Run

```
git status
```

to see which files have changed. Then run

```
git add readme.txt
```

to *stage* the file, this means that in your next commit, this file's changes will be recorded. Run

```
git commit -m "Added Name and Student Number to Readme file."
```

To push your changes back to BitBucket, run:

```
git push
```

This tells Git to send your changes to BitBucket. Go back to the website to see that your commits have been updated.

Finally, we want to fetch the code for Section 3. To add the code to your current branch, run:

```
git merge origin/HANDOUT1
```

This will open a text editor in the terminal called VIM. We want to accept the default commit message, so press ESC then `:wq` and hit Enter.

If you ever find that you've changed/deleted a file by mistake, or would like to get back to an older version of your code, you can access any committed version using `git checkout`. Git is an extremely powerful tool and I recommend getting comfortable with at least the basics. There are plenty of tutorials online, just search for them on Google if you ever get stuck!

2.3 SDL 2

In this course we will be using the Simple DirectMedia Layer (SDL) which will talk to the graphics card on our behalf. SDL is a cross platform library and runs on Windows, Mac OS X, Linux, iOS and Android. This means that if written carefully you can potentially port your games across to other platforms. For more information see <https://www.libsdl.org/> and for in depth tutorials see <http://lazyfoo.net/tutorials/SDL/>.

2.4 Sprites

In computer graphics a sprite is an image that we can manipulate and render to the screen as a single unit. These sprites are often distributed as a *sprite sheet* where the individual graphics are placed in a grid. For example, in our sprite sheet (`sprites.png`), you'll see that the images for the red ghost are found on row 5 and each image can be cut out into 20 pixel by 20 pixel blocks. We will need to write a number of helper functions that let us access the different items on the sprite sheet.

2.5 Unit Testing

Unit testing provides programmers with a convenient framework to test that their code is working correctly. We create a number of test cases and the *unit test framework* will run them all, so that we can be sure that our program is doing what we think it should be doing. We will be using a framework called `Catch`. You'll see that there is a makefile distributed in the code you downloaded earlier. If you type

```
make
```

in the terminal, it will compile your code along with the tests. For the first few sections, you don't need to worry about a main function - the tests will run everything for you. Later on you'll add in a make function. The compilation will create an executable called `a.out` which you should be able to run by typing `./a.out` in the terminal.

2.6 Framework

In the repository you cloned earlier, you'll find that I have provided the general structure for your program, and have coded a number of the more specialised SDL functions. Other functions are left unimplemented and you will need to complete them and make them work. You may add functions to the framework but you need to make sure that every required function is implemented correctly to receive full marks for the project.

3 Getting Started with Graphics and Games

There are 3 major issues that we will consider separately when building our game. The *Game State*, the *State Updates* and the *Display/Graphics*. The state describes the current condition of the game. This includes the position of all the characters, the design of the map, and various other bits of information that we need to remember to know what's happening in the game. The state updates apply the rules of the game, for example, if Pacman is facing left, then in the next time step his position along the x axis must change by -5 pixels unless he has hit a wall. This also includes the artificial intelligence and rules needed to update the positions of the ghosts. Finally, we need a way to display the current state of the game to the user. We want to do this in a way that is visually appealing and intuitive for the user to understand.

In the rest of this project, we'll work on getting drawing sprites to the screen, animating them and make the basic game logic work. In the next projects we'll focus on adding the ghosts.

3.1 The Window Class

I've written the window class for you. This class is a wrapper for the SDL Window object. The window gives us space on the screen to which we can draw. You'll see that there is a default constructor (that doesn't take any arguments) as well as a constructor that takes the width and height of the window. These constructors call the relevant low level functions to initialise SDL and create the necessary `sdlWindow` and `sdlRenderer` objects. The window gives us space to draw and the renderer does the actual drawing - we'll cover this in more detail soon.

Other than the constructors you'll see that there is a destructor and a `free()` function. These release the SDL window and renderer and quit the SDL system. This will happen at the end of the program so that we don't try to hold on to resources that we are no longer using.

The other functions return the width and height of the window in pixels.

SDL will allow us to make multiple windows, but we don't need that in this game and it'll make things more complicated than they need to be. In our program we will only ever have 1 instance of our window class. If you are interested in learning out more about these SDL functions, look at the Lazy Foo tutorials and they are all explained in detail. That's all we need in the window class for now.

3.2 The Texture Class

The texture class is the first important class that you'll need to write to build the game. This class will store all the images that we need to draw to the screen and will provide a render function that we'll use to draw all the different spites that we want to access.

3.2.1 Constructor

First you need to write the default constructor. If you look in `texture.h` you'll see that there are 5 different variables that need to be initialised. As we haven't loaded any images yet, we want to give all the integer variables a value of -1 and all the pointer variables a value of the `nullptr`. Implement this code in `texture.cpp`

If you open a terminal and run `make test` your code should compile. If you run your program with `./a.out` you should find that it now passes the first few *tests*. If you run your program with `./a.out -s` it will list ever test that passes.

3.2.2 LoadFile

This function loads the sprite grid from an image file and creates the relevant SDL hardware textures on the graphics card. I've given you this function for free, but if you're interested in how it works, use Google or have a look at the SDL reference to see what each function does.

What's important for us, is that the sprite sheet is loaded into memory, the `myTexture` pointer points to the correct location and that all the dimension variables are filled.

Because I've written this function for you, you should find that the second test case passes as well!

3.2.3 Free

Because the SDL memory is not automatically managed for us, we need to make sure that we free the texture memory when we're done with it. Implement the `free()` function so that it checks whether `myTexture` is the `nullptr`. If it is, then we don't need to do anything, if it is not, then

we need to call `SDL_DestroyTexture` and set all the relevant variables to their default values as we did in the constructor.

It is very important that you call `SDL_DestroyTexture` in this function, otherwise the memory used on the graphics card will not be freed. The C++ unit tests will not be able to check whether you have made this call, so if you're unsure, ask a tutor to check if its right.

Your code should now pass the next test.

3.2.4 SheetWidth, SheetHeight, TileWidth, TileHeight

You'll notice that we have four functions that should just return the values of 4 variables!

These functions are public, but the variables are private/protected (until you learn more about object oriented programming, assume that these mean the same thing). Why didn't we just make the variables public?

These are called *accessor methods*. We follow this *design pattern* so that we can stop programmers from accidentally changing the values of these variables. By making the actual variable private and only allowing access to it through a function, it doesn't allow someone using our code to change it by mistake. In this case, we're the only ones using our code, but this helps stop us introducing silly mistakes.

Implement these four functions so that they return the values of the relevant variables. Your code should now pass the next test case as well.

3.2.5 Render (Let's draw something to the screen!!!)

We are now ready to draw something to the screen! Finally! You'll see that there are two functions called `render(...)`. This is called *Function Overloading* and it allows us to have two functions with the same name, that accept different arguments. The compiler is smart enough, that is know which version of the function to call based on the type of arguments it receives.

We are going to implement the first version:

```
void Texture::render(int x, int y, SDL_Rect src);
```

An `SDL_Rect` is just a structure that has an `x`, `y`, `w` and `h` that represents a rectangle on the screen. `x` and `y` represent the co-ordinates of the top left of the rectangle, and `w` and `h` give us the width and height respectively.

This function should take in `x` and `y` which are the co-ordinates on the window to which the image should be rendered. The four values of `src` then describe where in the texture we should find the image to draw.

Use Google to look up how to use `SDL_RenderCopy`. Remember that the SDL Renderer is in `myWin.sdlRenderer` and the SDL Texture pointer is `myTexture`. We already have a source rectangle (from the `src` argument). The destination rectangle can be constructed using the `x` and `y` arguments and the width and height should be the same as those of the source rectangle.

If you've coded the render function correctly, then you should see a blue window appear with the sprite sheet displayed on top as shown in Figure 3. Following that it should change to show just the red ghost. This window will automatically close after about 10 seconds.

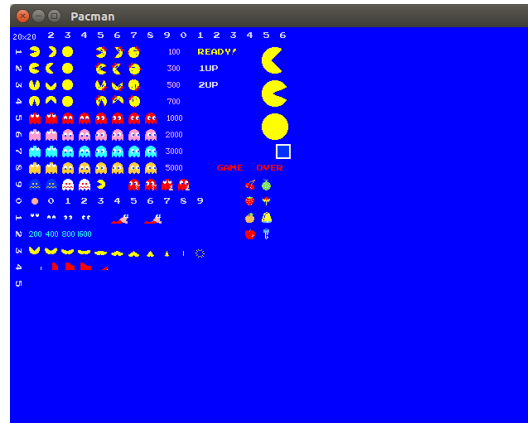


Figure 3: Correct Output After First Rendering Function

Congratulations! You can now draw images to the screen!

3.2.6 GetSpritePosition

We are now going to complete the last two functions necessary for the Texture class. At the moment we can draw an image to the screen using the first `render` function. The problem is that we have to know the `x` and `y` co-ordinates of the sprite in the sheet. It would be much easier if we could just say “Draw the sprite on row 4 column 6.”

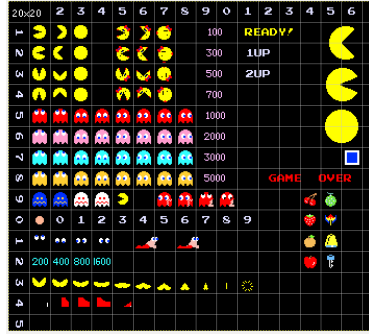


Figure 4: Sprite Guide

Figure 4 shows how the sprites are positioned. `getSpritePosition(...)` should take in the row and column of the sprite and as well as its width (in squares, not pixels) and return the `SDL_Rect` that is required by our first version of `render`.

For example, to get the red ghost we would call `getSpritePosition(5, 1, 1, 1)` and would expect the function to return an `SDL_Rect` with values: `{20, 100, 20, 20}`.

If we wanted to get the text that says Ready! we would call `getSpritePosition(1, 11, 3, 1)` and it should return: `{220, 20, 60, 20}`.

Your code should now pass the next test case.

Finally, complete the second render function. This should pass the relevant information to `getSpritePosition` to get the corresponding `SDL_Rect`. After this, it should call the other version of the render function using the `x`, `y` and `SDL_Rect` information.

If your code passes all the test cases and is rendering correctly to the screen then you are almost ready to move on to the next section where we talk about the event loop and animation!

3.3 Commit, Push & Fetch New Code

Remember to commit and push your changes to your repository!

To view files that have changed, run:

```
git status
```

We only want to keep changes to `texture.cpp` so we need to stage the file:

```
git add texture.cpp
```

Then, to commit your changes, run:

```
git commit -m "Texture class renders to screen."
```

Now push your changes back to BitBucket:

```
git push
```

Now we want to tell git to add on the new code for the next section, so do this, we are going to merge in changes from my code. Run the following command to update your project:

```
git merge origin/HANDOUT2
```

Remember :wq to save the default commit message.

4 Animation

Now that we can draw individual sprites to the screen we can look at how to animate them. We want Pacman's mouth to open and close and we want the ghosts to wriggle around! We do this by showing each frame of the animation in sequence. When we do this fast enough, then it appears as though there is a smooth movement of the character. As you can see in the sprite sheet, all the different frames of the characters are presented in a grid, so to make it look like Pacman's mouth is opening and closing, we first render the image with his mouth half open (1,1), after that we render the image with his mouth completely open (1,2), then back to a half open mouth (1,1) and finally the image with his mouth completely closed (1,3). Once we have completed this sequence we repeat it indefinitely until the game is finished.

To represent an animated sprite we are going to construct the `Tile` class. The tile class's constructor will take the `x` and `y` co-ordinates where it should be rendered on the window, it will remember which sprite images form the different frames of the animation and finally it will remember its type and dimensions.

```
1 class Tile
2 {
3     public:
4     Tile(int windowX, int windowY,
5         std::vector<std::pair<int,int>> frames,
6         TileType t,
7         int spriteWidth = 1, int spriteHeight = 1);
8
9     void render(Texture *t, int frame);
10    int x, y, w_inTiles, h_inTiles;
11    std::vector<std::pair<int,int>> myFrames;
12
13    TileType myType;
14
15    // Static variables are shared between all instances of the class
16    static int tileWidth, tileHeight;
17 };
```

The first two parameters give the pixel co-ordinates of the top left corner of the tile on the window. The second parameter is a vector of `pair<int,int>` objects. Each pair object consists of two integers, we can access the items in a pair object using the `first()` and `second()` functions respectively. Using this structure to represent the Pacman animation above we would look like this:

```
1 vector<pair<int, int>> myFrames = { {1,1}, {1,2}, {1,1}, {1,3} };
```

Each tile also has a type. We use something called an *enumeration* to specify the different types of tiles.

```
1  enum TileType{Pacman, MrsPacman, GhostR, GhostP, GhostB, GhostY,  
2      Wall, Blank, Food};
```

This is just like passing an integer where Pacman corresponds to 0, MrsPacman corresponds to 1 etc. The benefit here is that it makes our code much easier to use and the compiler treats the different `TileType` values as though they were normal types.

Finally the last two parameters represent the width and height of the images in the animation measured in sprites. We use a default value of 1 here as most of our tiles will be the normal single tile width.

This class will also have a render function that accepts a pointer to our texture object and takes a frame number. It will then tell the texture object to render frame number `frame % numFrames`.

For example, when working with `myFrames` above, we would have the following behaviour:

Function Call	Behaviour
<code>myTile.render(&myTexture, 0)</code>	Renders <code>myFrames[0]</code>
<code>myTile.render(&myTexture, 1)</code>	Renders <code>myFrames[1]</code>
<code>myTile.render(&myTexture, 2)</code>	Renders <code>myFrames[2]</code>
<code>myTile.render(&myTexture, 3)</code>	Renders <code>myFrames[3]</code>
<code>myTile.render(&myTexture, 4)</code>	Renders <code>myFrames[0]</code>
<code>myTile.render(&myTexture, 5)</code>	Renders <code>myFrames[1]</code>
<code>:</code>	<code>:</code>
<code>myTile.render(&myTexture, n)</code>	Renders <code>myFrames[n % myFrames.size()]</code>

Complete the constructor and render functions. After doing so correctly, your code should pass the `Tile` test case. Note that nothing will be drawn to the screen just yet. You'll be setting that up properly in the next section.

4.1 Commit, Push & Fetch New Code

Remember to commit and push your changes to your repository!

To view files that have changed, run:

```
git status
```

We only want to keep changes to `tile.cpp` so we need to stage the file:

```
git add tile.cpp
```

Then, to commit your changes, run:

```
git commit -m "Tile class constructor and render functions."
```

Now push your changes back to BitBucket:

```
git push
```

Now we want to tell git to add on the new code for the next section, so do this, we are going to merge in changes from my code. Run the following command to update your project:

```
git merge origin/HANDOUT3
```

5 The Main Loop

In applications like our game we need to have a loop that co-ordinates the different parts of the game. The loop will sit in the main function and will look something like this:

```
1 int main() {
2     // SpriteSheet Filename
3     string spriteFilename = SPRITEFILENAME; // Leave this line
4
5     // Setup and Load Texture object here
6     bool quit = false;
7
8     while(!quit){
9         // Handle any SDL Events
10        // Such as resize, clicking the close button,
11        // and process any key press events.
12
13        // Update the Game State Information
14
15        // Draw the current state to the screen.
16    }
17
18    return 0;
19 }
```

There are three main phases in the loop. The first checks whether there are any SDL events that our program should handle. The second should update the positions of pacman and the ghosts, and apply any other rules of the game. The third will call the relevant render functions to draw everything onto the screen.

In this section, we are going to handle the first and last phases as well as some of the setup before the loop runs. Note that there are no unit tests for this section, as the output is completely graphical.

5.1 Initial Setup

Before our program starts we need to make sure that we create and load the texture object. We will only need one texture object in our program. In the `main.cpp` file, declare a `Texture` object called `myTextures` and then call `loadFile(spriteFilename, 20, 20)`.

Also create a `bool` variable called `quit` and set its value to `false`. Now make a while loop that will continue looping until `quit` becomes `true` as shown in the code above.

5.2 Event Loop

We now need to add code that will ask SDL whether anything interesting has happened to our window. For now we are interested in knowing whether the user has clicked on the quit button.

```
1  while(!quit) {
2      // Handle SDL Events
3      SDL_Event e;
4      while(SDL_PollEvent(&e)) {
5          if(e.type == SDL_QUIT) {
6              quit = true;
7          }
8      }
9      // Update Game.
10     // Draw to screen.
11 }
```

SDL keeps a list of all the events that have happened to our window called the `Event Queue`. When we call `SDL_PollEvent` it removes one of the events from that list and gives us information about it in the `SDL_Event` object. It returns `true` if there was an event to fetch, and `false` otherwise. In the code above we loop over all the events in the queue and check whether the type of event is `SDL_QUIT`, if it is, then we set the `quit` variable to `true` which should stop our main loop from repeating. If the type is not `SDL_QUIT` then we ignore the event (for now).

If you run the code now, it should show a window. The window won't have anything useful drawn on it, but if you click the "x" button to close it, you should find that it now closes correctly.

5.3 Drawing to the Screen

5.3.1 Controlling the Renderer

You'll see that the window probably just has garbage in its borders – nothing is drawn when it first appears. So the first thing we want to do is clear the renderer so that we start with a clear screen. After the event loop, add the following line:

```
1  SDL_RenderClear(myTexture.myWin.sdlRenderer);
```

This clears the memory that the renderer uses to draw to the window. After this, we tell the renderer to actually draw this memory to the window using the following line:

```
1 SDL_RenderPresent(myTexture.myWin.sdlRenderer);
```

If you run your code now, you'll see that the window should be all blue. This is because I set the default colour for the renderer to blue. You can change it by adding the following line before the `SDL_RenderClear` call:

```
1 SDL_SetRenderDrawColor(myTexture.myWin.sdlRenderer, 0, 0, 0, 255);
```

This will set the background colour to black. You can read more about these functions on the SDL website.

5.3.2 Draw Sprites

If all your code up to this point is working, you are ready to animate some sprites!!! Create a tile called `pm` before the main loop starts.

```
1 Tile pm(0,0,{ {1,1}, {1,2}, {1,1}, {1,3}},Pacman,1,1);
```

Then, inside the main loop, between the calls that Clear and Present the renderer, call the render function of our tile, giving it the address of our texture and telling it to render frame 0.

```
1 // Set background to black
2 SDL_SetRenderDrawColor(myTexture.myWin.sdlRenderer, 0, 0, 0, 0xFF);
3 // Clear the renderer
4 SDL_RenderClear(myTexture.myWin.sdlRenderer);
5 // Render the tile
6 pm.render(&myTexture, 0);
7 // Copy the memory of the renderer to the window
8 SDL_RenderPresent(myTexture.myWin.sdlRenderer);
```

If you run your code now, you should find that the window displays frame 0 in the top left corner of the window.

5.3.3 Animate Sprites

Now all we need to do is tell our program to render a different frame each time the loop executes. Before the main loop, create an integer variable called `frame` and set its value to 0. At the end of the loop, after the render present, increment the value of `frame` by 1. Instead of calling

```
pm.render(&myTexture, 0)
```

we want to use our new frame variable to change the frame that's drawn each time the loop runs. Change the line above to:

```
pm.render(&myTexture, frame)
```

If you run the code now, you'll see that pacman is sitting in the corner uncontrollably guzzling away! The problem now is that the loop is running too fast, so we want to slow it down. We will

tell the program to sleep for a 75 milliseconds at the end of each iteration of the loop by adding this line:

```
this_thread::sleep_for(chrono::milliseconds(75))
```

5.3.4 Commit, Push & Fetch New Code

Remember to commit and push your changes to your repository!

To view files that have changed, run:

```
git status
```

We only want to keep changes to `main.cpp` so we need to stage the file:

```
git add main.cpp
```

Then, to commit your changes, run:

```
git commit -m "Animate Sprite in Main Loop."
```

Now push your changes back to BitBucket:

```
git push
```

Now we want to tell git to add on the new code for the next section, so do this, we are going to merge in changes from my code. Run the following command to update your project:

```
git merge origin/HANDOUT4
```

5.4 Create Tiles based on Type and Direction

Note: You need to add `#include "helpers.h"` at the top of `main.cpp`

Throughout the game, we will need to create lots of tiles as these represent each sprite that we want to render on the screen. It would be helpful to have a function that would create a `Tile` based on the `TileType` and `Direction`.

In `helpers.cpp` implement the `makeTile` function, that must correctly construct and return a `Tile` based on the supplied parameters. I would recommend using a switch statement on the `TileType` and then, if relevant, check the `Direction`. For example, if the type is `Wall` then regardless of the direction, the function should return

```
1 Tile(x, y, {{7,16}},t,1,1);
```

If the type is `Pacman` then depending on the direction, it should do something similar to this:

```

1  switch(dir) {
2      case Up:
3          return Tile(x, y, {{3,1},{3,2},{3,1},{3,3}},t,1,1);
4      case Left:
5          return Tile(x, y, {{1,1},{1,2},{1,1},{1,3}},t,1,1);
6      case Down:
7          return Tile(x, y, {{4,1},{4,2},{4,1},{4,3}},t,1,1);
8      case Right:
9      default:
10         return Tile(x, y, {{2,1},{2,2},{2,1},{2,3}},t,1,1);
11 }

```

There is a function called `getTestTiles` that returns a `vector<Tile>`. Before your main loop, call this function by adding:

```

1  vector<Tile> testTiles = getTestTiles();

```

In your main loop, comment out the line where you were rendering `pm` and add a loop where you iterate over all the `Tiles` in `testTiles` and call the relevant render function. If everything is correct, you should see a window of animated sprites similar to that in Figure 5.

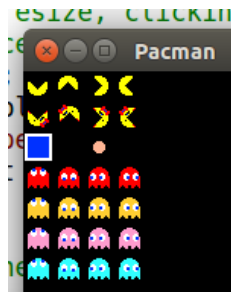


Figure 5: TestTile Output

5.4.1 Commit, Push & Fetch New Code

Remember to commit and push your changes to your repository!

To view files that have changed, run:

```
git status
```

We only want to keep changes to both `helpers.cpp` and `main.cpp` so we need to stage them:

```
git add helpers.cpp main.cpp
```

Then, to commit your changes, run:

```
git commit -m "Helper function to create tiles by Type and Direction"
```

Now push your changes back to BitBucket:

```
git push
```

Now we want to tell git to add on the new code for the next section, so do this, we are going to merge in changes from my code. Run the following command to update your project:

```
git merge origin/HANDOUT5
```

5.5 Characters

Now that we can animate and easily construct tiles we need a way of handling which `Tile` gets rendered to the screen based on the direction of a character. Complete the code for the character class, so that a character can be constructed from 4 tiles, or from a `TileType`.

Hint: makeTile function from the previous section.

The character class should store 4 `Tile` objects, a direction and its window co-ordinates. Each of the 4 `Tile` objects represents the `Tile` that should be rendered based on a particular direction. When we call the `render()` function, it should update the `x` and `y` values of the relevant tile and call its `render()` function. Implement the `render()` function.

We also need a `handle_event` function that will take in an `SDL_Event` object. This function checks if the type of event is an `SDL_KEYDOWN` event (i.e. the user pressed a button). If it is, then updates the direction of the character. This function has been coded for you and is just a normal switch statement. If the type of event is not an `SDL_KEYDOWN` event, then the function leaves the direction unchanged.

In your main loop, delete the `tileTest` variable from the previous section and create a number of character objects using the relevant types (`Pacman`, `MrsPacman` and the `Ghosts`) as well as the different directions and check that they render to the screen correctly.

Don't worry about `getNextPosition` for now, we'll cover that later in [Section 7](#).

5.6 Connecting a Character to the Event Loop

Before the main loop, create a character called `myPacman`.

```
Character myPacman(0,0,Pacman);
```

Inside the main loop, call this object's render function, using the `frame` counter from earlier. You should see an animated sprite of Pacman facing up. We now want to connect this to our event loop so that the direction it faces changes as we press the different arrows on the keyboard.

The top of our main loop has the event loop that processes each event that is waiting for us from SDL. In the event loop, call `myPacman.handle_event(e);`

```

SDL_Event e;
while(SDL_PollEvent(&e)) {
    if(e.type == SDL_QUIT){
        quit = true;
    }
    myPacman.handle_event(e);
}

```

This will get our pacman object to check whether it needs to change direction. If you now run your code, pacman's direction should change when you press the different arrows on the keyboard.

5.6.1 Commit, Push & Fetch New Code

Remember to commit and push your changes to your repository!

To view files that have changed, run:

```
git status
```

We only want to keep changes to both `character.cpp` and `main.cpp` so we need to stage them:

```
git add character.cpp main.cpp
```

Then, to commit your changes, run:

```
git commit -m "Character Class Wraps Tiles to handle direction."
```

Now push your changes back to BitBucket:

```
git push
```

Now we want to tell git to add on the new code for the next section, so do this, we are going to merge in changes from my code. Run the following command to update your project:

```
git merge origin/HANDOUT6
```

6 Representing the Game

We will now take a break from animating our characters for a while and we'll look at how to render the maze. Once we have done this, we will come back to the characters and make it so that they can move around the maze properly!

6.1 Game State

Implement the `World` class in `world.cpp`. The constructor should take a filename as a string. The input file will have the following format:

- The first line will have 2 numbers, (R C) separated by a space. These represent the number of rows and columns respectively.
- There are then R rows. Each row has C columns.
 - `x` Represents a Wall.
 - (space) Represents a blank tile.
 - `.` Represents a food tile.
 - `0` Represents the starting position of Pacman.
 - `1` Represents the starting position of the red ghost.
 - `2` Represents the starting position of the yellow ghost.
 - `3` Represents the starting position of the pink ghost.
 - `4` Represents the starting position of the blue ghost.
- For the maze, the tiles below Pacman and the Ghosts should be considered blank tiles.

For example the following map has 14 rows and 12 columns.

```
14 12
xxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxx
x...xx...x
x.xx...xx.x
x.x..xx..x.x
x.....x
x.x.xx x.x.x
x.x.x12x.x.x
x...xxxx...x
x.x..0...x.x
x.xx.xx.xx.x
x.....x
xxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxx
```

The constructor should read in the file, and create a `vector<vector<Tile>>` maze where each tile is constructed using the `makeTile` function. When you encounter Pacman and Ghost tiles in the file, the maze array should have a blank tile (i.e. the `TileType` is of type `Blank`). Calculate the correct `(x, y)` co-ordinates for Pacman and initialise the character object with the correct position on the screen.

Now implement the `render` function, that loops over all of the Tiles in the 2D vector and calls their relevant render functions.

Before the main loop, create a `World` object called `myWorld`. In your main loop, remove all the code relating to the pacman and character objects. Between the `SDL_Renderer` functions, call `myWorld.render(&myTexture, frame);` This should correctly render the maze to the screen.

The constructor should also count the number of food pellets in the game and initialise the relevant fields for the game.

6.2 Render Pacman On Top!

In the `World` class there is also a `Character` object called `pacman`. Update your `World::render` function so that after rendering all the tiles in the maze, it calls the render function on the pacman object.

Whenever we *blit* or render something to the `SDL_Renderer` it overwrites whatever we had drawn there before. This means that if we draw the map first, then draw the pacman object, pacman will simply be drawn over whatever map tiles were already in that space. This means that it's easy to draw pacman on top of things – we call render on the bottom layer first, then call render on the stuff that needs to go on top.

Running your code should now render the maze with pacman on top. If you modify your event loop so that it calls `myWorld.pacman.handle_event(e);` you should be able to control the direction that pacman faces while he is rendered on top of the maze.

Change your main function so that it has the code from Listing 1. Remember you need to `#include "world.h"` at the top of the file. If you run your code it should now show the map with pacman at the starting position.

6.3 Commit & Push

Stage, commit and push your changes to the repository. You do not need to merge anything this time.

```

1  int main()
2  {
3      // SpriteSheet Filename
4      string spriteFilename = SPRITEFILENAME; // Leave this line
5
6      // Setup and Load Texture object here
7      Texture myTexture;
8      myTexture.loadFile(spriteFilename, 20, 20);
9      int frame = 0;
10
11     bool quit = false;
12
13     World myWorld(MAZEFILENAME, myTexture.tileWidth(),
14                  myTexture.tileHeight());
15
16     while(!quit){
17         // Handle any SDL Events
18         // Such as resize, clicking the close button,
19         // and process and key press events.
20         SDL_Event e;
21         while(SDL_PollEvent(&e)){
22             if(e.type == SDL_QUIT){
23                 quit = true;
24             }
25             myWorld.pacman.handle_event(e);
26         }
27         // Update the Game State Information
28         // myWorld.UpdateWorld();
29
30         // Draw the current state to the screen.
31         SDL_SetRenderDrawColor(myTexture.myWin.sdlRenderer, 0, 0, 0, 255);
32         SDL_RenderClear(myTexture.myWin.sdlRenderer);
33
34         myWorld.render(&myTexture, frame);
35
36         SDL_RenderPresent(myTexture.myWin.sdlRenderer);
37         frame++;
38
39         this_thread::sleep_for(chrono::milliseconds(75));
40     }
41
42     return 0;
43 }

```

Listing 1: Main Function

7 Character Movement and Collision Detection

Now that we have initialised the maze and pacman, we want him to be able to move around the map. First we are going to update our character class so that on every iteration of the loop it calculates its new position in the world based on its current direction. In the second section, we will consider a method called *Collision Detection*, to make sure that characters can't pass through walls.

7.1 Character Movement

In the `Character` class, implement the function called `getNextPosition` that calculates the `SDL_Rect` representing the position of the character in the next time step of the game. For example, if Pacman is currently at `{40, 60, 20, 20}` (`x, y, pixelW, pixelH`)⁵ and his direction is left, then this function should return `{35, 60, 20, 20}` (for a velocity of 5 pixels per timestep - which we will use).

In the `Maze::UpdateWorld` function, call `pacman.getNextPosition`. Then, also in `Maze::UpdateWorld`, set Pacman's co-ordinates equal to the new values. Finally, in the main loop, after the event loop, but before the render functions, call `myMaze.UpdateWorld()` as shown on line 28 in Listing 1.

If your code is working correctly, you should see the maze with pacman drawn over it and he should be moving around the screen. By pressing the arrow buttons, you should be able to control pacman. Note that at this stage, pacman should move through walls and doesn't actually eat the food because we haven't implemented that code yet.

Commit and push your changes.

7.2 Collision Detection

Now that we have pacman moving around the screen, we need to set it up so that the interactions between the map and the characters are correct. We don't want characters moving through the walls!

We do this by implementing *collision detection*. There are many different methods to do this, but we'll implement a simple (and fast) method based on rectangles.

Implement the `collision` function in `helpers.cpp` based on the algorithm described in class. There are comments to guide you through the implementation. The function should return

⁵The `Tile` class stores pixel width of a block in `Tile::tileWidth/tileHeight` variables. Remember to set them in your main function.

true if the two `SDL_Rect` arguments overlap, and false otherwise. For the case where the rectangles are touching, consider the rectangles to be non-overlapping (i.e. return false). For example, `{0, 0, 1, 1}` and `{0, 1, 1, 1}` are not colliding.

In `myWorld.UpdateWorld` we now need to check that Pacman's new position doesn't collide with a wall before actually updating his position. We need to use the `getNextPosition` to get the position where Pacman would like to be based on his current direction. We then iterate over all the tiles in the maze and, if that tile is of `Wall` type, then we check if there is a collision. If there is a collision with a `Wall` tile, then it means that Pacman cannot move into his new position and we should leave him where he is. If there is no collision, then he is free to move to the new position on the map.

After iterating through all the different tiles in the maze (for the walls), we then need to see if Pacman has found a food pellet. If Pacman's move succeeds, we should then check if he collides with a tile of type `Food`. In that case we need to increment the points variable and replace that tile with a new tile of type `Blank`. Each time Pacman collides with `Food`, then you should print out the current score to the terminal.

Once this is implemented correctly, you'll find that you have a playable version of the game (without Ghosts for now). At the end of `UpdateWorld` write some code that checks whether Pacman's points equal the total number of available food items from the start. If so, then Pacman has eaten all the food and the game is complete. When the game is over, render the text "Game Over" over the map in the centre of the maze.

For the pro's: See if you can make this "Game Over" text flash once every 10 frames!

7.3 Final Extra Tweaks

7.3.1 Fine Tune Collision

You'll find that sometimes its hard to get Pacman to turn into an opening in the wall as he has to be in the exactly correct position to avoid a collision. We have the same issue that the system registers that Pacman has eaten food when the graphics show that he's still a few pixels away from it. We can avoid this by shrinking Pacman/the food tile/wall tile's `SDL_Rect` before calling the collision function. Update your collision function so that it subtracts the offset from the relevant rectangles before checking for a collision.

Change your implementation of `UpdateWorld` so that it when checking for a collision between Pacman and a `Wall`, the rectangles shrink by 3 and 2 pixels respectively. Comparing Pacman and a `Food` tile, shrink both by 5. Play around with different values to see what works best!

7.3.2 Handle Window Resize

To make the graphics scale to the size of the window, you need to edit your event loop to handle a window resize event. The event should call `myTexture.handle_event` with the number of rows and columns in the world as shown below. This will tell the SDL renderer to stretch to fit the window.

```
1     SDL_Event e;
2     while(SDL_PollEvent(&e)) {
3         if(e.type == SDL_QUIT) {
4             quit = true;
5         }
6         myTexture.handle_event(e, myWorld.rows, myWorld.cols);
7         myWorld.pacman.handle_event(e);
8     }
```

A call to `scaleGraphics` before the main loop will make sure that the graphics are setup correctly from the start.

8 To be continued...

You should now have a working version of Pacman without any ghosts. This project walked you through the process of creating your own game in lots of detail because its probably the first game you've ever written! Future projects will use the data structures presented in class to extend Pacman by introducing ghosts and some intelligence so that they will chase Pacman around the map!

Make sure you commit and push your code!

9 Grades

This part of the project will be graded in the labs on 23 and 30 July. Don't leave the labs without having your code marked. Grades will be awarded based on the following criteria:

- 20%: Consistent use of git
- 25%: Code correctly passing the relevant test cases.
- 25%: Correct screen rendering.
- 30%: Game plays correctly.

Part 2 of this project will involve Linked Lists and will focus on adding ghosts to the game. Part 1 and Part 2 will each count 50% towards your Project 1 mark.