

Cell Cycle, Technical Design Document

David Robertson, Thomas Hope, Davide Passaniti

Contents

Overview	2
Concept	2
Technical Goals	2
Technical Risks	2
System Requirements	2
Technology	3
Programming Team	3
Feature List	3
Technical	3
Art	3
Audio	4
Research	4
Feasibility	4
Challenges identified	4
Implementation	5
Documentation	5
Task Tracking and Assignment	6
Source Control	7
Bug Tracking	7
Testing Tool	7
Infinite grid	8
Type 3 engine on windows desktop with openGL ES	8
Compiling SDL for Android	9
Engine Overview	9
Purpose	9
Scope	9
References	9
Overview	10
Design	10

Building, testing, saving and sharing changes with Git and Android Studio	13
Prerequisites	13
First time setup (cloning the repository)	13
Opening the project and running it	14
Creating a new virtual device in Android Studio	19
Emulator trouble shooting and tips	20
Changing an asset (shader, texture, audio)	20
Changing the source code	21
Saving your changes	21
Pulling changes	23
Using Vim	23
Resources for Git and Vim	24
Schedule	24
About This Document	25
Compiling	25
TDD Version Control	25

Overview

Concept

The game is about cells growing and changing state. The player has to manage the growth of cells on a hex grid, cells will move autonomously if the player doesn't take action. Each cell has internal state that changes based on adjacent cells.

Technical Goals

The code team will be implementing the game in C++ using cross platform libraries. The code should be tailored specifically to our game, object orientated and minimal. Given the limited time the focus will be on simplicity and functionality over 'advanced' code.

Technical Risks

- Not having core gameplay features implemented
- Other members of the team being held back by slow engine progress
- Programmers having to be assist in testing game assets
- Could be harder to debug
- Might not build for target platform
- spending lots of time on engine programming, not enough time to polish
- 'infinite' grid required memory allocation = crashes (mitigate with modern c++ features)
- Bugs

System Requirements

Target platform is Android Tablets. The application will be compiled and tested with the AndroidSDK, NDK and android studio.

Device Specifics:

- 2gb of ram
- android 4.3 or higher
- gpu supporting opengl es 2.0 or higher

Technology

- C++
- OpenGL ES
- SDL2
- SDL2_mixer
- SDL2_image

Programming Team

- David Robertson
- Davide Passaniti
- Thomas Hope

Feature List

Technical

- splash screen
- main menu
- options menu
- Touch input, single touches
- Drag navigation
- Pinch to zoom
- hex grid
- “infinite” grid (memory allocation)
- scene culling (don’t render the entire world)
- nodes perform checks on adjacent nodes
- nodes change their state + spawn based on user input
- minimal path finding / line drawing on the grid
- interactive tutorial
- reading data from text files

Art

- loading png images
- static images
- rotated / scaled images
- animation from sprite sheets
- blending between animations
- transparency
- colour tinting
- programmatic drawing of the hex grid
- user Interface
- text rendering

Audio

- loading wav files
- loading mp3s
- play audio once
- volume control
- loop audio
- randomly select sound
- pitch shifting
- time shifting

Research

Feasibility

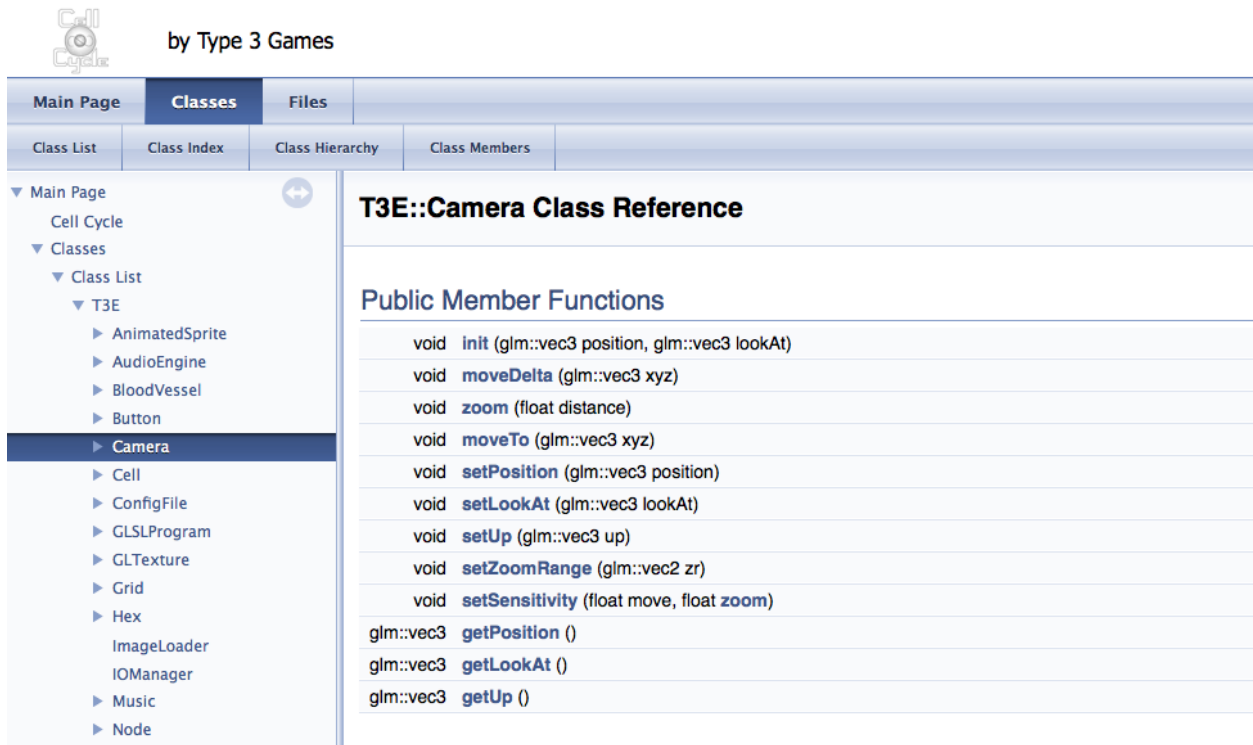
- Spoken to programmers from a team last year who wrote their own framework from scratch
- The game concept is fairly simple and there are 3 motivated programmers
- Online resources to aid in the creation of the basic framework

Challenges identified

An easy to use pipeline for the visual and audio artists will be essential to ensuring fast iterations and all members of the team are able to fully bring their skills to bear on the project.

Implementation

Documentation



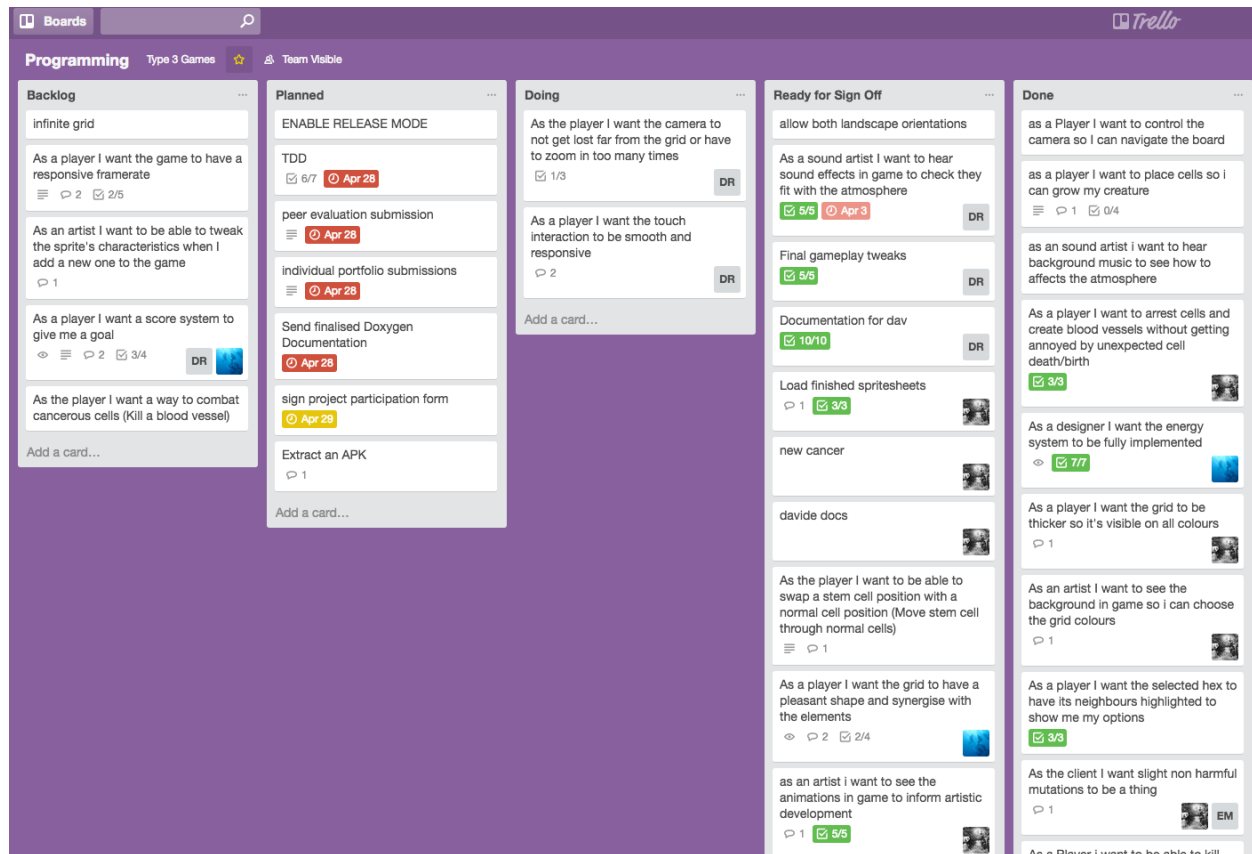
The screenshot displays the documentation for the **Cell Cycle** project, created by **Type 3 Games**. The interface includes a navigation sidebar on the left with a tree view showing the project structure: Main Page, Cell Cycle, Classes, Class List, T3E, and various sub-classes like AnimatedSprite, AudioEngine, BloodVessel, Button, Camera, Cell, ConfigFile, GLSLProgram, GLTexture, Grid, Hex, ImageLoader, IOManager, Music, and Node. The **Camera** class is selected. The main content area is titled **T3E::Camera Class Reference** and lists the **Public Member Functions** of the class:

- `void init (glm::vec3 position, glm::vec3 lookAt)`
- `void moveDelta (glm::vec3 xyz)`
- `void zoom (float distance)`
- `void moveTo (glm::vec3 xyz)`
- `void setPosition (glm::vec3 position)`
- `void setLookAt (glm::vec3 lookAt)`
- `void setUp (glm::vec3 up)`
- `void setZoomRange (glm::vec2 zr)`
- `void setSensitivity (float move, float zoom)`
- `glm::vec3 getPosition ()`
- `glm::vec3 getLookAt ()`
- `glm::vec3 getUp ()`

In addition to the TDD extensive documentation of the source code of *Cell Cycle* was created. The source documentation was written using specially formatted comment blocks allowing for the markup of parameters, return values, implementation and intent. These specially formatted comments were then processed using the tool [Doxygen](#) which converted them to a searchable, well structured website.

The generated documentation website was then hosted using [gh-pages](#), a feature provided by GitHub, and is viewable by clicking [here](#).

Task Tracking and Assignment



The programming team's tasks were tracked using a group Trello board. The board was organised around five lists, each with a specific function

1. Backlog: When anyone comes up with an idea it goes here, no matter how crazy or ambitious it is or who thought of it
2. Planned: These are the features that have been agreed on *as a group*, things only go in planned when the group is committed to completing them
3. Doing: Each member of the team should have one, and no more, cards in doing at a time. They update the card with notes and checklists as they make progress
4. Ready for Signoff: When a assignee *thinks* they are finished, they put it here
5. Done: Tasks only move from Ready for Signoff to Done when *the group* agrees they are done. Often what will happen is the group will review the result and ask for some changes to be made, the task will be given a modified or more specific definition and moved back to planned.

Ensuing this method is effective requires enforcing a few rules about how people interact with the board. Importantly tasks only move from Backlog to Planned, or Ready for Signoff to Done with the approval of *at least* two people. Preferably the entire group would be present.

The title of each card should describe a user story, i.e. something that represent value to the *end user*. User stories are often written using the format: **As a [user] I want [feature] so I can [reason]**

- “background audio” becomes “as a sound artist i want to test background music so I can see how it affects the atmosphere”
- “have a grid structure to put cells in” becomes “As a player I want to place cells so I can grow my creature”

The description of each card should include a *definiton of done*. A definiton of done usually takes the form

Done when [description]. Each card should have a sentence or set of criteria that is true when the user story has been fulfilled. A user story for animation could have a definition of done as:

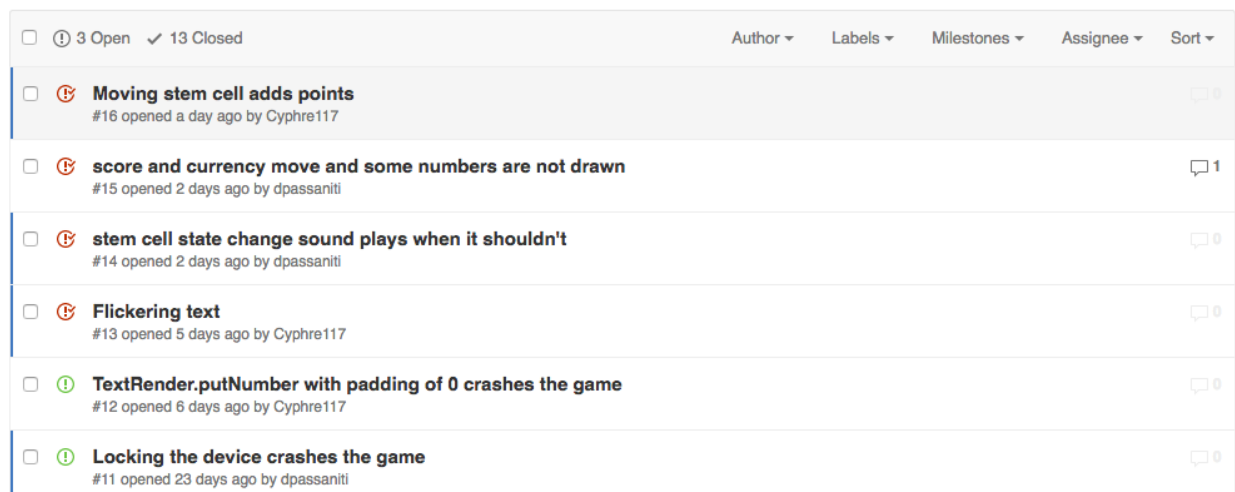
- “Done when cells have animations for idle, dividing and mutating, and these animations are triggered automatically according to the cells internal state”

Source Control

We will be using Git source control. The engine is currently stored in a private repository on Github.

A log of commits can be generated by navigating to the repository and executing `git log --pretty=short --graph > log.txt` in a terminal session.

Bug Tracking



The screenshot shows a GitHub Issues page for a repository. At the top, there are filters for '3 Open' and '13 Closed' issues. Below the filters, there are tabs for 'Author', 'Labels', 'Milestones', 'Assignee', and 'Sort'. The main content area displays a list of six open issues, each with a checkbox, a status icon (bug or info), a title, a number, and a comment icon. The issues are:

Issue Number	Status	Title	Author	Opened
#16	Bug	Moving stem cell adds points	Cyphre117	a day ago
#15	Bug	score and currency move and some numbers are not drawn	dpassaniti	2 days ago
#14	Bug	stem cell state change sound plays when it shouldn't	dpassaniti	2 days ago
#13	Bug	Flickering text	Cyphre117	5 days ago
#12	Info	TextRender.putNumber with padding of 0 crashes the game	Cyphre117	6 days ago
#11	Info	Locking the device crashes the game	dpassaniti	23 days ago

When bugs are found they are added to the issue tracker associated with the repository on Github. As much information as possible about the bug should be included e.g. which platforms the bug exists on, conditions to reproduce the bug, already investigated solutions, and potential lines of enquiry.

Generally when a member of the team comes across a bug they would first spend some time investigating it themselves. Then if at some point they decide there are more pressing matters to move onto or progress slows they would post an issue on github describing what they had learned up to that point.

Testing Tool

Update 2016-04: The testing tool does not currently exist as a standalone tool, but some of the motives for creating such a tool were satisfied following the implementation of the `ConfigFile()` class.

While the engine is under development it would be very valuable to have some kind of instant feedback tool for the other members of the team. Rather than having to wait to have a programmer to integrate some new asset into the game they could test and iterate quickly on designs in their own time. The testing tool could show previews of animations under certain game conditions or how the manually created art would look alongside some procedural art or shader. It could also allow for the testing of audio assets, randomised events, or asset pools.

Infinite grid

Update 2016-04: This feature was considered not necessary for the prototype and shelved to be revisited at a later date.

The game will consist of a 2d hex grid which the player will be able to grow their cells across, hopefully infinitely. While it is impossible to have a truly infinite grid without infinite memory it is possible to create a game world so large that no user will conceivably reach the boundary. Game with such worlds already exist, the most popular of which is Minecraft. Minecraft's world is three dimensional, and infinite in two dimensions. The game is divided into 'chunks', each chunk being dynamically created as required and unloaded when no longer needed.

Will likely use STL containers since they are robust and well documented, the reference below makes use of a `std::map` to store chunks of the grid. Each chunk can be sorted within it's container for easy lookup. When checking the contents of some grid coordinate, first look in the container to see if the chunk with that coord exists. If it does search within the chunk to find the coord, if not then allocate the chunk.

The allocating and linking of chunks of memory should be handled within a game board class. Callers can then use simple `set(x, y)` and `get(x, y)` functions without having to worry about the implementation. When it comes to rendering the board a check will have to be done for each chunk to see if it's on screen so only chunks that are actually visible are drawn.

sources:

- https://github.com/gummikana/infinite_grid.cpp
- <http://www.redblobgames.com/grids/hexagons/>

Type 3 engine on windows desktop with OpenGL ES

The following is an example implementation of how to get Type 3 Engine to run on windows using OpenGL ES:

1. suppose our code requires the use of the function `glCreateProgram()`
2. make sure `SDL_opengles2.h` is being included in the file where we want to use the function
3. looking up the definition in `SDL_opengles2.h`, identify the types for the function's return value and input parameters (in this case `Gluint` and `void` respectively)
4. create a typedef similar to this:

```
typedef Gluint(APIENTRY * GL_CreateProgram_Func)(void)
```

(APIENTRY resolves to `__stdcall`)

5. Create a variable with the function's name using the new typedef, making sure the scope is local to where the function has to be called

```
GL_CreateProgram_Func glCreateProgram = NULL
```

6. Finally, assign the function's address to the variable using

```
glCreateProgram = (GL_CreateProgram_Func)SDL_GL_GetProcAddress("glCreateProgram")
```

7. Now `glCreateProgram()` can be used as normal in that scope.

This process has to be repeated for every required function. Also specify a GLES context when creating the window:

```
SDL_GL_SetAttribute(SDL_GL_CONTEXT_PROFILE_MASK,SDL_GL_CONTEXT_PROFILE_ES);
SDL_GL_SetAttribute(SDL_GL_CONTEXT_MAJOR_VERSION, 2);
SDL_GL_SetAttribute(SDL_GL_CONTEXT_MINOR_VERSION, 0);
```


Compiling SDL for Android

The following list describes the process with which we managed to get the SDL android project to run on Android Studio's emulator (Nexus 5):

1. Download Android Studio, the android NDK and the SDL2 source code.
2. Install the sdk platform and tools for the target platform you want to develop for(NDK supports up to 21, we used 18 in this test). This is done via the SDK-manager tool that comes with the IDE.
3. Set the environment path variable to where the ndk folder is to be able to call build commands later on (on windows 10: control panel > system and security > system > advanced system settings > environment variables).
4. Start android studio and choose "import project (eclipse, gradle,...)".
5. Select the "android-project" from the SDL source folder.
6. Copy the SDL source folder in `YourNewProject\app\src\main\jni`.
7. Create a `main.cpp` file in `YourNewProject\app\src\main\jni\src` and add your code(in our test: initialise SDL, create a GLES window, draw a triangle with a shader, wait for input to quit).

Then make the following changes to the following files (using api 18 for examples) 1. `YourNewProject\app\src\main\jni\Android.mk` add `APP_PLATFORM := android-18`

2. `YourNewProject\app\src\main\jni\src\Android.mk`: change `yourSourceHere.c` to `main.cpp` and make sure `SDL_PATH` is the correct relative (or absolute) path to your SDL source folder
3. `C:\Users\<USER>\.gradle`: create a file named "gradle.properties" and add `android.useDeprecatedNdk=true` in it
4. `YourNewProject\local.properties`: add `ndk.dir=<path to your ndk folder>`
5. `YourNewProject\app\build.gradle`: change `compileSdkVersion`, `minSdkVersion` and `targetSdkVersion` to 18 inside `defaultConfig` scope add: `~ ndk { moduleName 'main' }` ~ inside android scope add: `~ sourceSets.main { jni.srcDirs = [] jniLibs.srcDir 'src/main/libs' }` ~ This is where we tell gradle we don't have native code to compile by clearing the native source directories paths
6. In android studio's terminal (or in the command prompt) move to `YourNewProject\app\src\main\jni` and run the "ndk-build" command. This will build SDL and the cpp files previously added.
7. Finally, run the gradle build and start the correct emulator (in our test a nexus5 with api 23).

Engine Overview

Purpose

The purpose of this piece of software is to enable the development of the game "The Enemy Within" via providing adequate tools for developing this title.

Scope

To create a framework that will provide the required tools for making a 2D game.

References

Making Games With Ben - <https://www.youtube.com/user/makinggameswithben>

Overview

This framework will provide the following tools for game creation:

- A camera class
- Error reporting
- An input manager
- A resource manager and the loading of resources
- A animated and regular sprite class
- Sprite Batching
- Timing commands
- An audio engine
- A way of reading a config file
- A way to render text
- Buttons for UI interaction

Design

Camera Class

The camera class consists of seven functions, however five of these are either “getter” or “setter” functions so they simply update or return a variable value. The remaining two functions consist of an initialisation function which simply sets up the camera class by initialising all of the variables and giving them base values.

The final function is the update function which does the heavy lifting of the class, when called it will check if the camera needs to be updated or not. If it does it will translate the camera’s matrix to the correct position and then scale it accordingly.

Error Reporting

This will simply output an error string and check for the users’s input before exiting the program.

Input Manager

The input manager consists of 3 functions and an unordered map. Two of the functions are very simple; they either set the value of the key to true or false in the unordered map. In doing this however, if they value of the key has not been added to the map yet, it is now added with the value it has been set to. The “is key pressed function”, checks to see if the value of the key that is being checked is in the unordered map. If it is not then false is returned. If the key value is in the unordered map, the Boolean value of that key is checked and if it is true, true is returned, otherwise false is returned.

Resource Manager

The resource manager consists of one function and a texture cache. The function “Get Texture” simply hands the file path onto the texture cache, which in turn will check to see if the file is contained in the texture cache, if it is then it will return the texture. If it is not then the texture will be loaded into the cache from the file path and returned.

Sprite Class

The sprite class simply contains two functions, the initialisation function which sets up all the base variables for the new sprite and loads the desired texture from the texture cache.

Animated Sprite Class

The Animated sprite class contains an initialisation function, four other public functions and a private function. The initialisation function takes in eleven variables, this will define the position of the sprite, its width and height, and how the animation is played. The draw function simply sends the data to the OpenGL rendering system to be rendered to the screen. The update function takes in the current delta time from the application and updates what frame the sprite should currently be displaying. The set speed function takes in a float and updates how fast the animation will play. The refresh function simply sets the needs refreshed boolean to true, this will make the update function reset the current frame being displayed to the first in the animation. The one private function updates the current UV coordinates depending on what frame needs to be displayed and rebinds the geometry.

Sprite Batching

- The Sprite Batching class consists of a “Glyph” struct, a “GlyphSortType” enum class, a “RenderBatch” class, five public functions and six private functions.
- The “GlyphSortType” simply defines what sorting method will be used in the sorting functions.
- The “Glyph” is used to store the texture, the depth and a vertex for each corner of the sprite.
- The “RenderBatch” class contains only one public function and three variables, the function simply initialises the render batch class.
- The initialisation function of the sprite batch class simply calls the create vertex array function.
- The begin function sets the sort type to TEXTURE, and then clears out all of the render batches and glyphs.
- The end function sorts all of the glyphs and creates all of the render batches.
- The draw function takes in all of the information about a glyph and creates a new one using this information. Then it pushes the glyph into the glyph vector.
- The render batch function firstly binds the vertex array and draws all of the render batches. It then unbinds the vertex array.
- The create render batches function checks to see if the glyphs vector is empty, if it is then the function returns. If it is not then the function continues. Runs through all of the glyphs in the glyph vector and adds them to the corresponding render batch, depending on what texture they are using.
- The create vertex arrays function simply generates vertex arrays and makes sure that there is a value for the vertex array and the vertex buffer.
- The sort Glyphs function checks what the value of sort type is, and then uses std stable sort to sort the vector of glyphs accordingly.
- The three compare functions are simply used to compare the values of two inputs. These are used by the std stable sort function.

Timing Commands

The **FpsLimiter** Class consists of four public functions and one private function. The initialisation function simply sets up the variables with base values and sets the maxFPS to the desired value. The set max fps function simply sets the max fps value to the value passed into the function. The begin function sets the start ticks value to the value returned from the `SDL_GetTicks()` function. The end function firstly calculates the fps by calling the calculate fps function and then gets the frame ticks by taking the start ticks value away from the current value returned by `SDL_GetTicks()`. It then checks to see if the frame ticks is less than one thousand divided by the max fps. If it is then the `SDL_Delay()` function is called and the value of one thousand divided by the max fps minus the frame ticks is sent. Finally the fps is returned. The calculate fps function firstly sets up four static variables consisting of the number of samples, an array of frame times, the current frame and the previous ticks which is equal to the value of `SDL_GetTicks()`. The current tick is then calculated by calling `SDL_GetTicks()` again and the frame time is then calculated by taking away the previous ticks from the current ticks. The modulo value of the current frame time and the number of samples is calculated and that position in the frame times array is set to the calculated frame time. The previous ticks are then set to the current ticks and the current frame is increased. A count variable is then created and if

the current frame is less than the number of samples the count variable is set to the current frame, if not the count variable is set to the number of samples. The average frame time is then calculated by adding all of the used values in the frame time array together and then dividing them by the count variable. if the frame time average is above 0 the fps is set to one thousand divided by the frame time average. If not it is set to sixty.

Audio Engine

The Audio engine class controls all of the audio in the game. This contains an initialisation function, a destroy function and two loading functions. This also contains two “friend” classes to make the division of music and sound effects easier. Using two “friend” classes, the audio engine will be able to load and play sound effects and music. the Audio engine will also contain a map of all the sound effects loaded and a map of all the music that has been loaded, this will stop any need to load sound assets during play.

Config File

The `ConfigFile()` class reads a given text file and creates a table of colon delimited **key: value** pairs. This allows various gameplay parameters such as score values awarded and deducted from the player, procedural colour values, and random percentages to be altered without recompiling the application from source. This allows non-technical members of the team to test out ideas autonomously without the need to wait for a programmer.

The constructor of the config file takes in a string as the filepath of the config file to be loaded. When the config file is processed keys and values are stored as strings in a STL `std::map`. The value strings are only converted to their numerical values when the `getValue()` functions are called. This requires the caller to know the type of the value they should be receiving while it is the responsibility of the config file editor to ensure that the correct value types are paired with each key.

When calling one of the get value functions on the config file a default parameter can be supplied. Then if the requested key is not found in the map the passed default value can be used instead.

The config file is also able to handle inconsistent whitespace and comments starting with a `#` symbol.

Text Renderer

The `TextRender()` loads a 256 character font file arranged in a 16x16 grid and uses it to display text and other dynamic information to the player.

The basic feature of the TextRender is its ability to draw single characters. Each character in the font file is mapped to a number in the range [0:255] and can be displayed at a given location on the screen in OpenGL clip space coordinates. ranging from (-1,-1) in the bottom left to (1,1) in the top right. The ability to draw strings and numbers is built up by the repeated application of the `putChar()` function.

Additionally the text renderer can add padding to numbers, ensuring they stay in the same position on screen as the number of digits increases. To do this the digits in the given number are first counted. This is then compared against the requested padding and an appropriate number of spaces, “”, are prepended to the string before drawing.

When a call is made to `putChar()` or one of the other rendering functions of the TextRender the result is not immediately drawn to the screen. Instead the output is calculated and added to an internal buffer. Later, once all other calls to the TextRender have been made the `render()` function is called. This binds the font texture, reads the entire buffer and renders it all at once before clearing the buffer ready for next frame. By storing the values in a buffer we are able to avoid binding and rebinding texture and vertex objects multiple times throughout the frame and improve performance.

Button

The 'Button' Class works very similarly to a regular sprite, however it has three extra setters and two extra getters. The setters simply relate to the condition of the button, these are press, unpress and toggle, these all manipulate the "isPressed" boolean. The first of the getters is the isPressed function which simply returns the condition of the isPressed boolean. The other getter is the getTextUnit function which returns the GLunit containing the information about the texture unit of the pressed sprite.

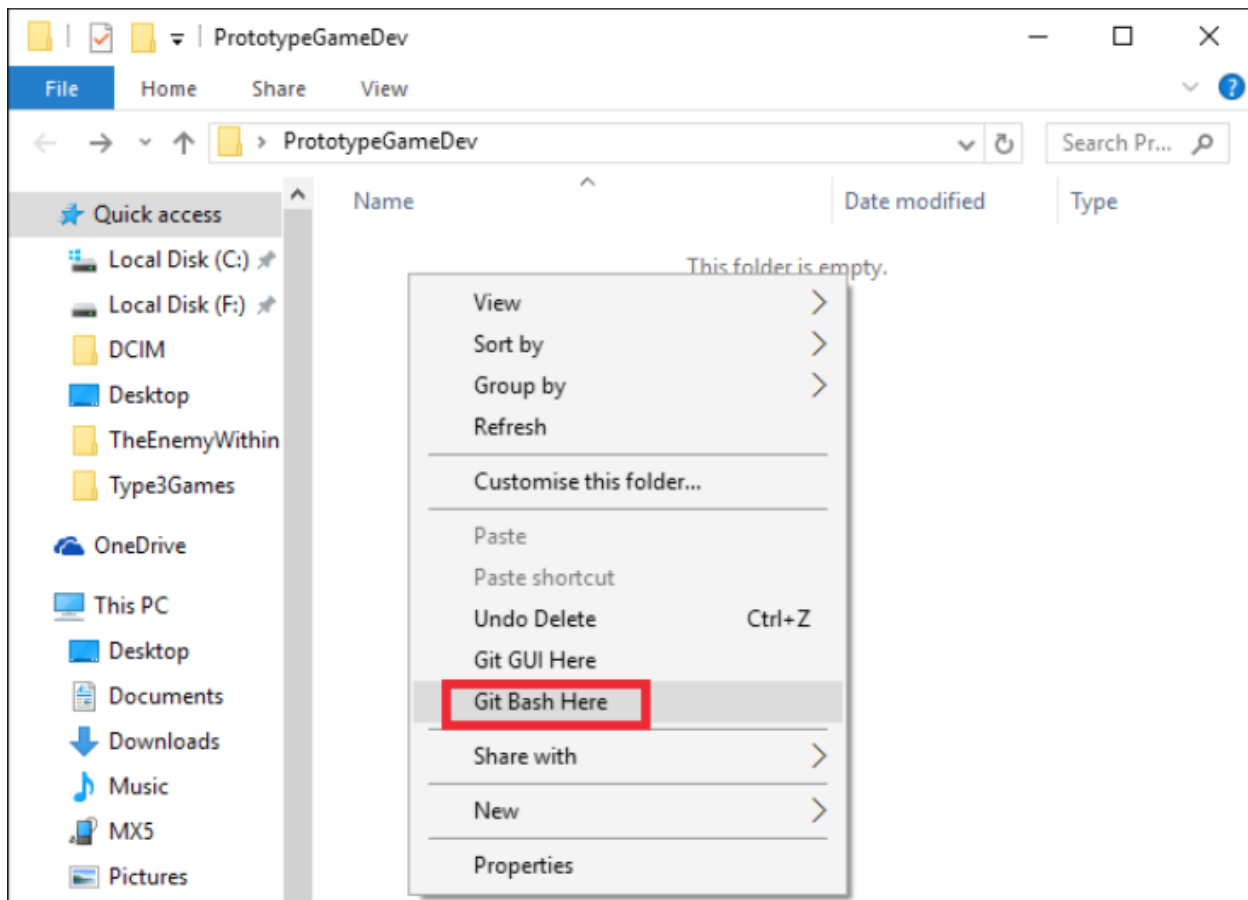
Building, testing, saving and sharing changes with Git and Android Studio

Prerequisites

- Download and install Android studio - <http://developer.android.com/sdk/index.html>
- Download and extract the NDK - <http://developer.android.com/ndk/downloads/index.html>
- Download and install Git - <https://git-scm.com/downloads>
- Create a GitHub account at github.com and post the email you used on slack so you can be invited to our repository

First time setup (cloning the repository)

- Create a folder somewhere on your computer e.g. "PrototypeGameDev" (will be using this name in this tutorial, you can call it whatever you like)
- Open the folder, right click in it and select 'Git Bash Here'



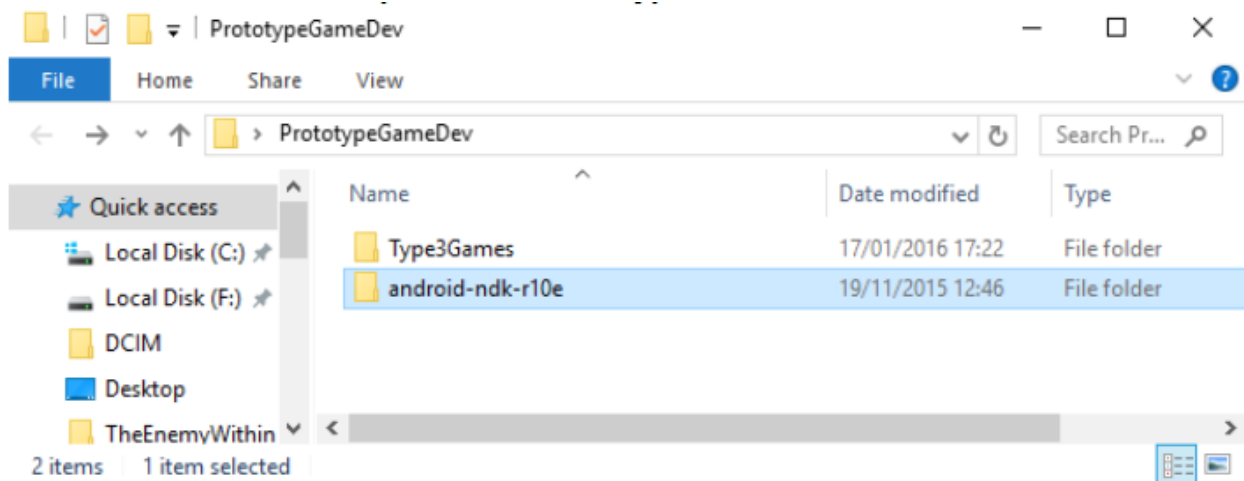
If there is no 'Git Bash Here' option in the context menu, go to where you installed git and open git-bash.exe, then type `cd path/to/where/you/put/PrototypeGameDev`

- Now that you have the git console open in the right folder, to clone the android branch from the repository write: `git clone -b android --single-branch https://github.com/d4v33d123/Type3Games.git` (you can paste this into the console with right click > paste, or right click on the title bar > edit > paste for older versions)



```
MINGW64: c:/Users/Da/Desktop/PrototypeGameDev
Da@DESKTOP-V3MMGHR MINGW64 ~/Desktop/PrototypeGameDev
$ git clone -b android --single-branch https://github.com/d4v33d123/Type3Games.git
```

- Press enter and you will be prompted to insert the username and password you used to register on Github
- When the download has complete, you should have a folder named Type3Games inside the PrototypeGameDev folder.
- You can add your name and email to git so that when you save changes in the future everyone will be able to see who did what. To do this type in the gitbash:
 - `'git config --global user.name "your user name here"'`
 - `'git config --global user.email "your email here"'`
 - You can omit `global` if there are other repositories on your computer that you don't want to affect.
- Now, as per prerequisites, you should have downloaded and extracted the NDK, and therefore have a folder named 'android-ndk-r20e' stored somewhere
- Take that folder and put it inside PrototypeGameDev

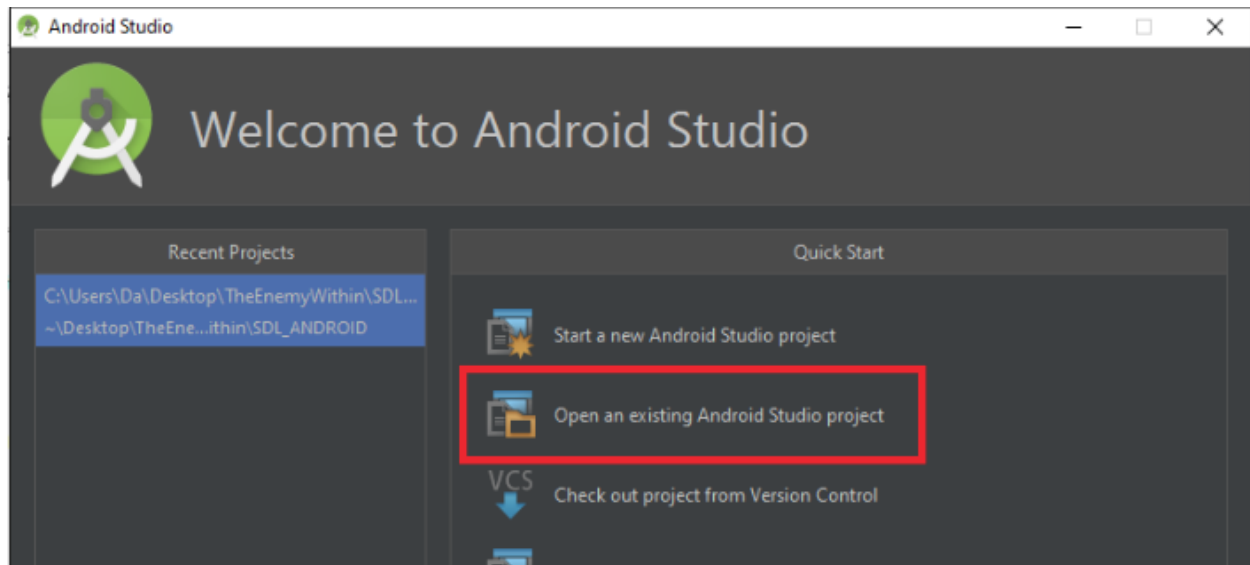


- Now you have a copy of the repository and the tools to build the code, you should not need to repeat this process ever again.

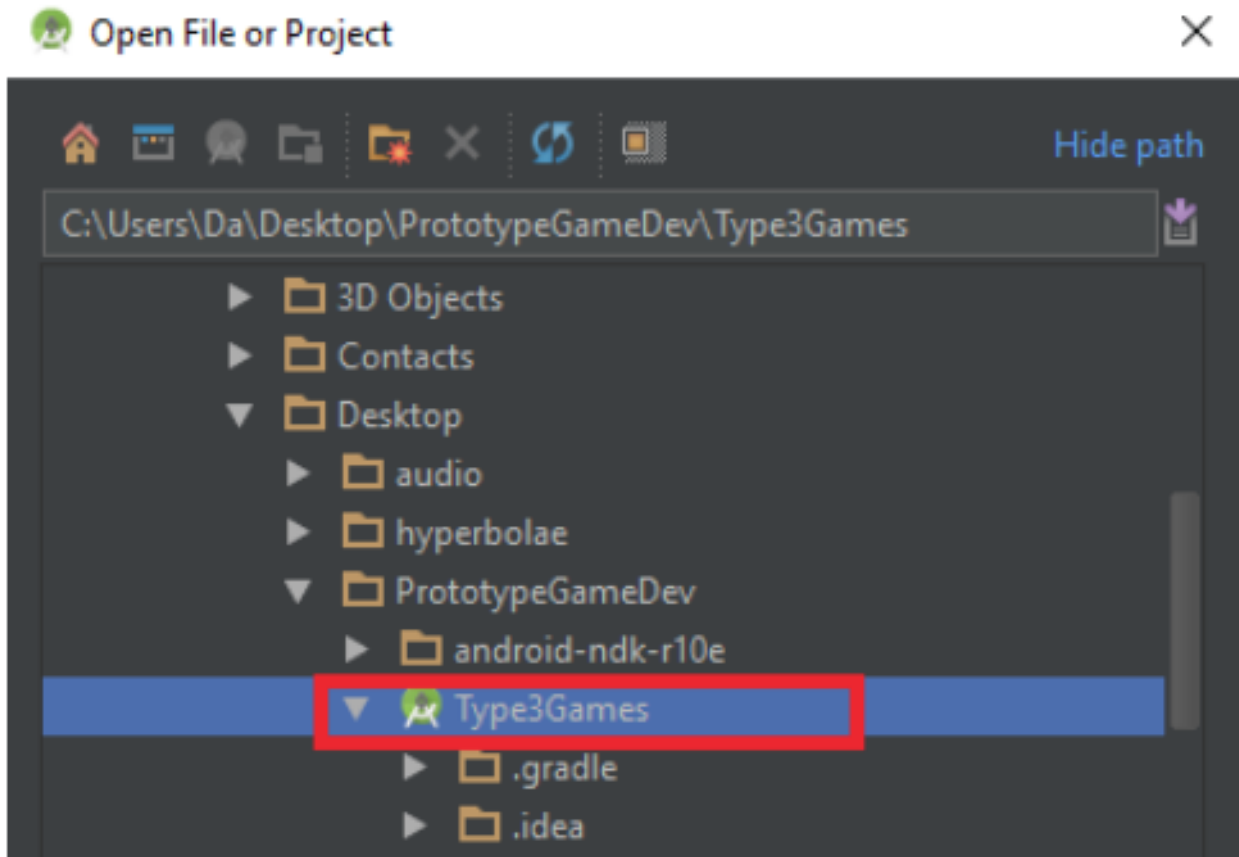
Opening the project and running it

Note: it seems mac has a different interface, so you might have trouble following these steps on mac.

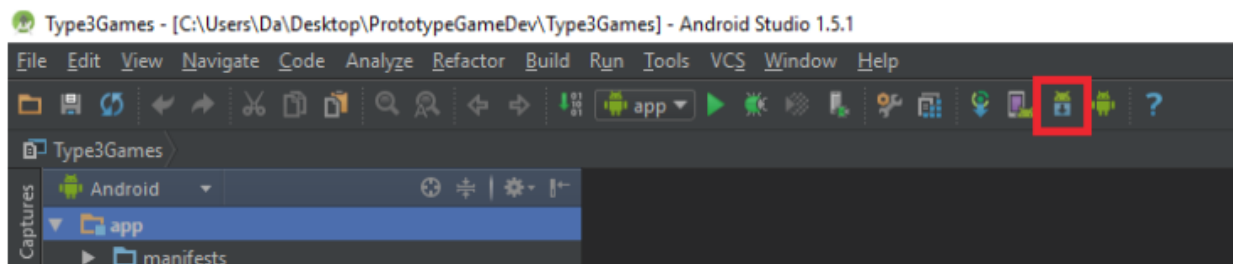
- If you just cloned the repository or just pulled some changes, go into Type3Games and double click the file named `BUILD_NATIVE_CODE.bat`
- If you have already opened the project before, opening android studio should automatically open the project as well.
- If this is the first time you open the project, open android studio and select 'Open an existing Android Studio project'



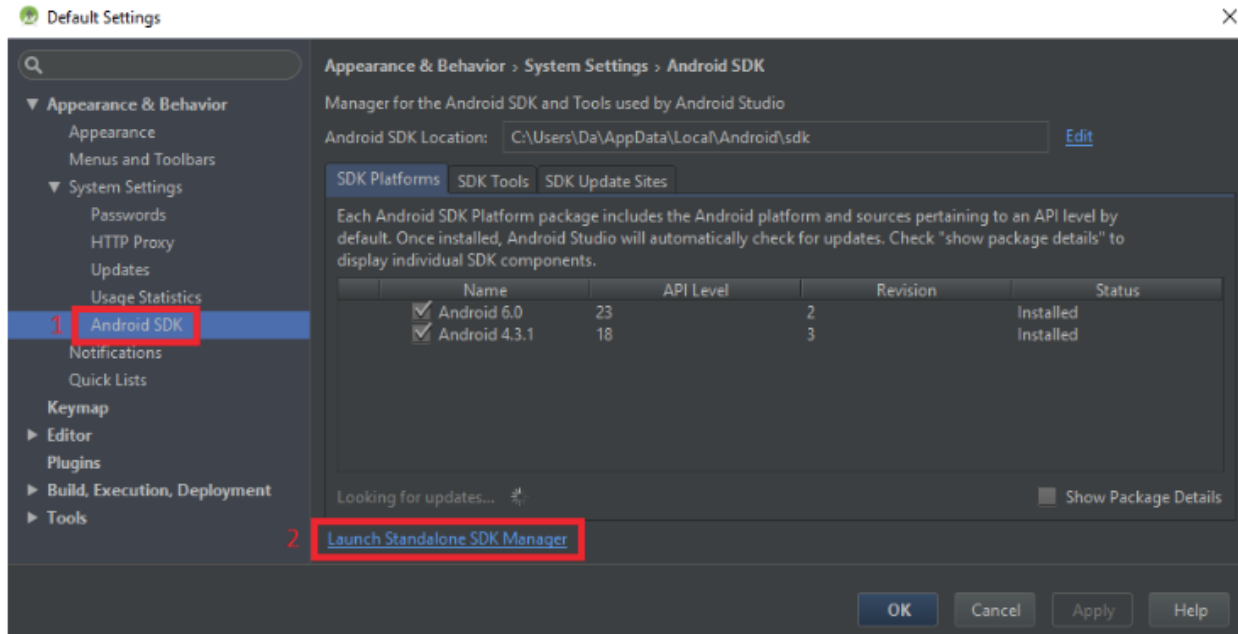
- Navigate to the Type3Games folder and select it



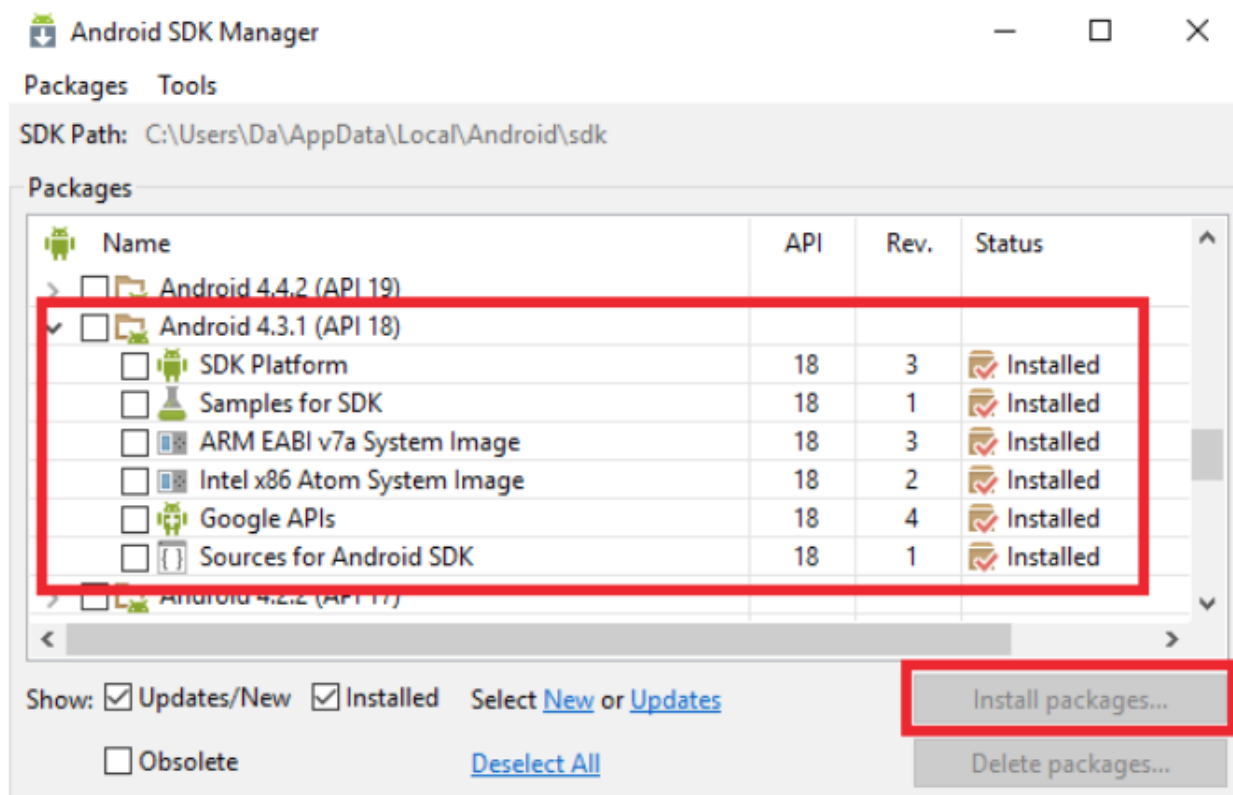
- If you don't have API 18 installed (or if you want to check), click on the SDK manager icon



- Under system settings select Android SDK then click on 'launch standalone SDK manager'

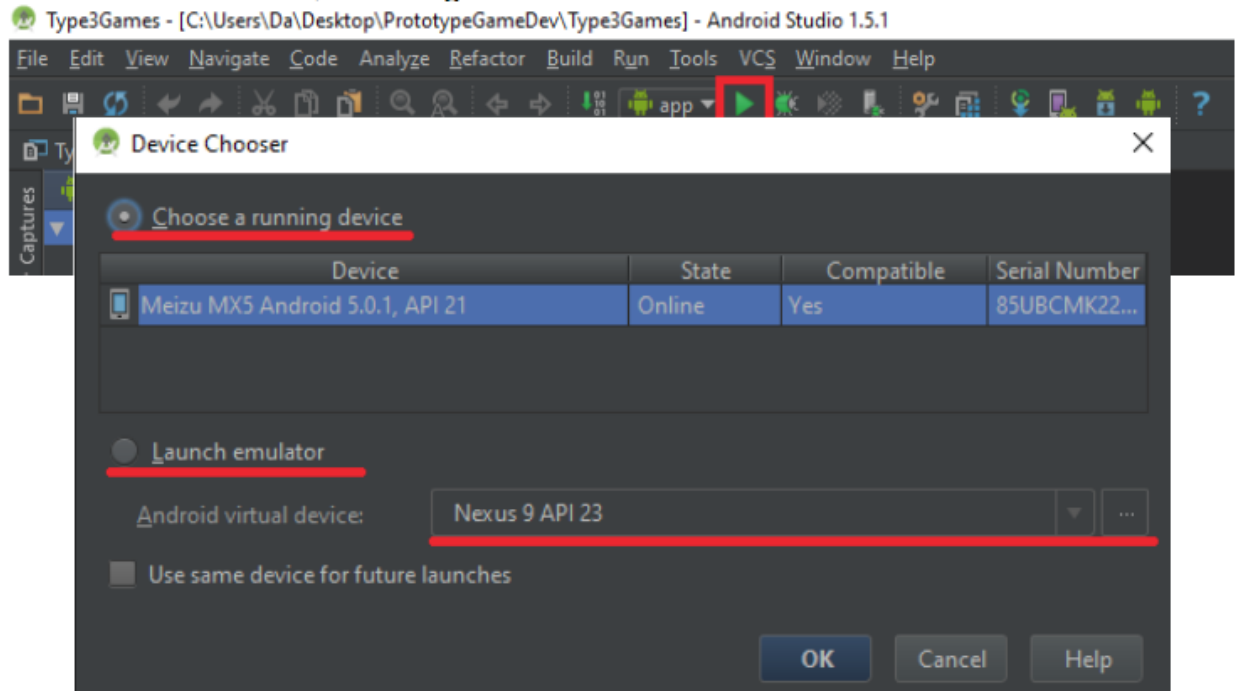


- Scroll down to Android 4.3.1 (API 18) and make sure the status of all components is 'installed'

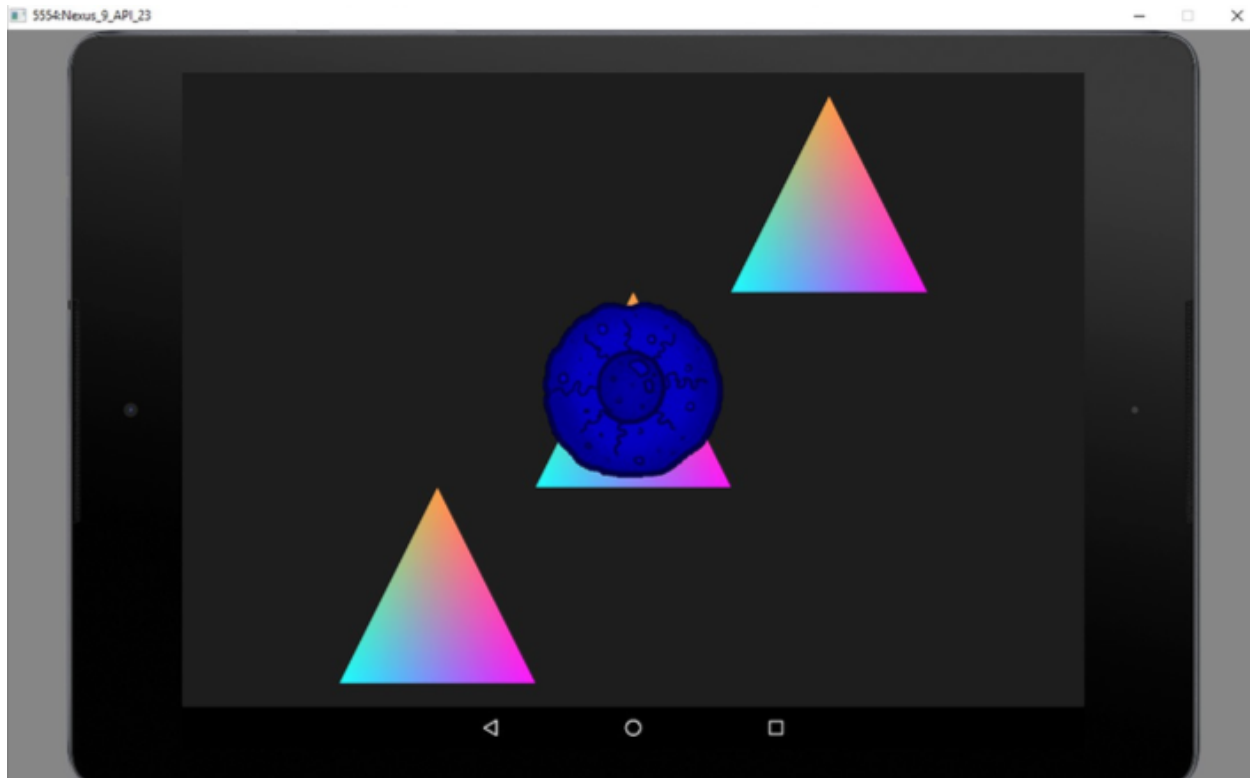


(if it isn't, check the related checkbox on the left and click on "install packages"). If other things are checked or you are unsure of something, installing extra stuff won't cause problems so just go with it.

- Back in the IDE, click the green 'Run' arrow

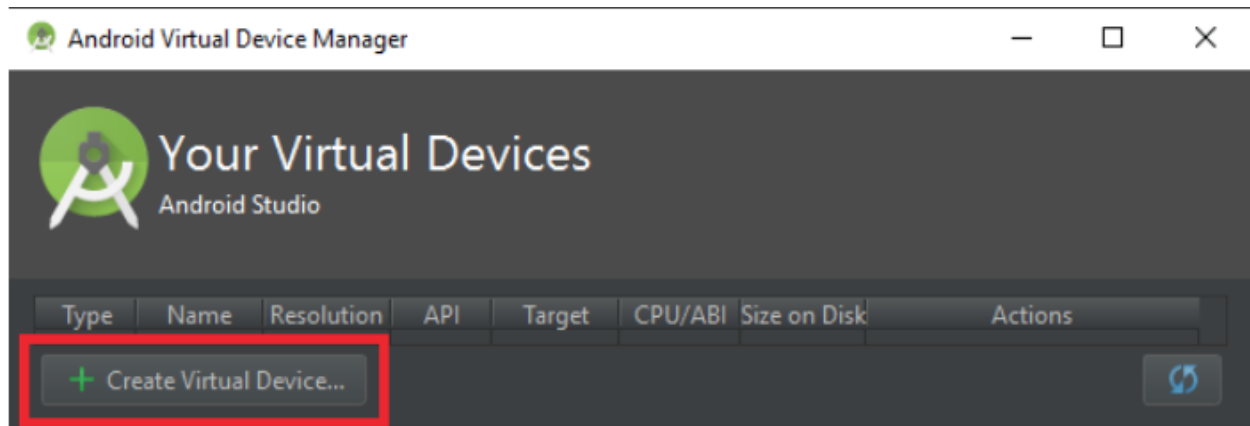


- This should open the device chooser window
- If you have a real tablet/phone plugged in, select “Choose a running device”, click on the device and click ok
- If you want to use the emulator, select “Launch emulator”
- If you have already created a virtual device in the past, it should be listed in the drop down list, so select it and click ok.
- If the drop down list is empty, click on the “...” next to it to create a new virtual device. Go to the “Creating a new virtual device in android studio” section of this document
- Wait a bit and the program should start automatically

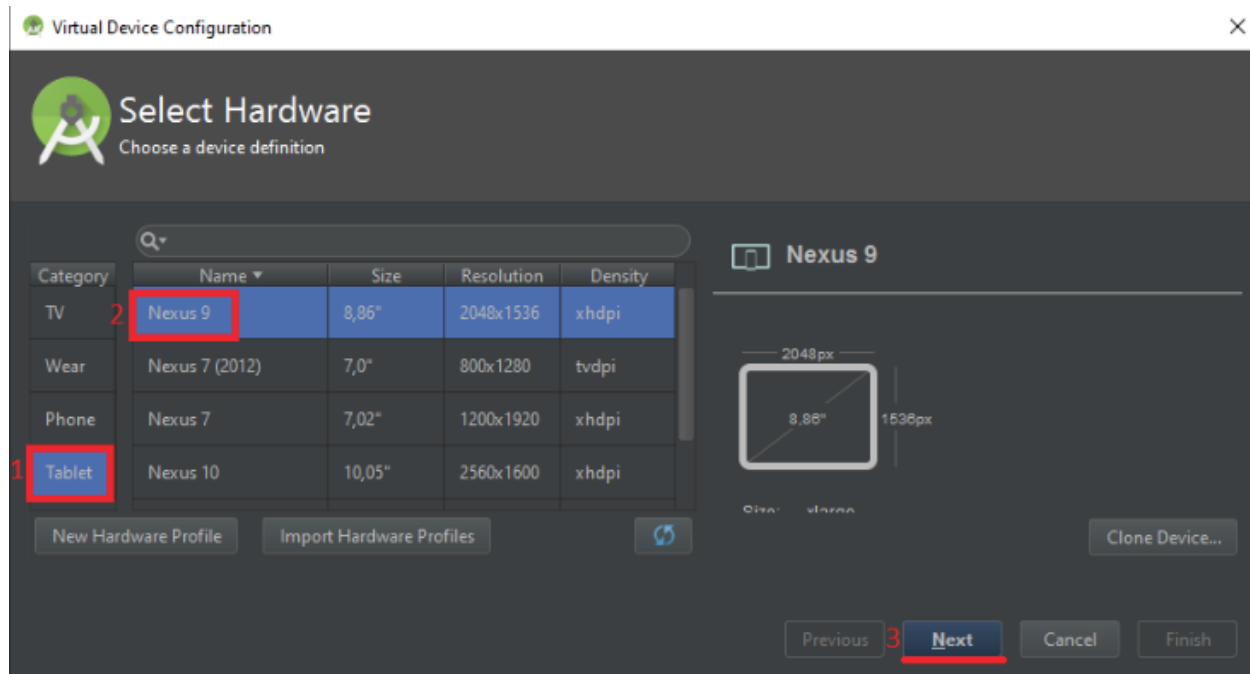


Creating a new virtual device in Android Studio

- Clicking on “...” in the device chooser window will open the AVD Manager. Click on Create Virtual Device



- In the window that opens select tablet, nexus 9, and then click Next, Next and Finish (if your computer is old/slow you might want to pick one of the phones, like the nexus 5, but try this first)



- Now close the Android virtual device manager window, select your new device from the drop down list in the device chooser window and click ok to run the program.

Emulator trouble shooting and tips

- don't drag the emulator around the screen, or if you have to do it do it quick because otherwise it might freeze for a while and you will have to wait.
- If you have the emulator running and you're not done working, don't close it. You can run the project again to see your changes by clicking the run arrow in the ide and avoid waiting for it to restart.
- sometimes the emulator will crash while starting. Try 3/4 times, if it keeps not working try one or more of the following:
- if your internet connection is bad, go offline before starting the emulator.
- When creating a device, instead of clicking next, next, finish, click next twice but before clicking finish find the "Use host GPU" checkbox on the left and uncheck it.
- try a less demanding virtual device (e.g. Phones , nexus 5)
- restart your computer
- If there is an error about "HAXM / hardware acceleration not working" or similar, when creating the device click next once, then before clicking next again, look at the list on the left and select an entry that has under ABI "armeabi-v7a" or similar instead of x86 or x64. Pick the "API Level" with the highest value.
- If the emulator is stuck on the android loading screen and in the command line an error regarding HAXM is mentioned, try downloading and re installing HAXM from <https://software.intel.com/en-us/android/articles/intel-hardware-accelerated-execution-manager> and selecting 1gb or more memory when asked.

Changing an asset (shader, texture, audio)

We will probably slightly change how this works, but the concept should remain the same:

- Go to `Type3Games/app/src/main/assets`
- this is where all shaders, textures, sprite sheets, audio files, etc. are kept

- find the file you want to change and take it out (or rename/delete/...)
- put the new file with the same exact name in its place
- run the program clicking the run arrow in the IDE like normal

Changing the source code

If you want to make changes to the source code you will need to rebuild it separately with the ndk before running from the ide:

- our code is in `Type3Games\app\src\main\jni\src`, remember we can't see it in the IDE, so open it with notepad++ or drag it into android studio or whatever you like
- If you create a new file, remember to add it to the `LOCAL_SRC_FILES` list in `Type3Games\app\src\main\jni\src\android`. Also remember to save changes before building (sounds stupid but I forget a lot when using 2 ides)
- If you set an environment path variable to your ndk folder (see guide in TDD on how to build sdl on android), you can write the following in the android studio terminal:
 - `cd app/src/main/jni`
 - `ndk-build`
- Alternatively you can run the batch file `BUILD_NATIVE_CODE.bat` I put in the project folder that does the same thing (for the batch script to work the ndk folder must be placed where I described in the first setup section)
- if everything builds successfully click the run arrow in the IDE

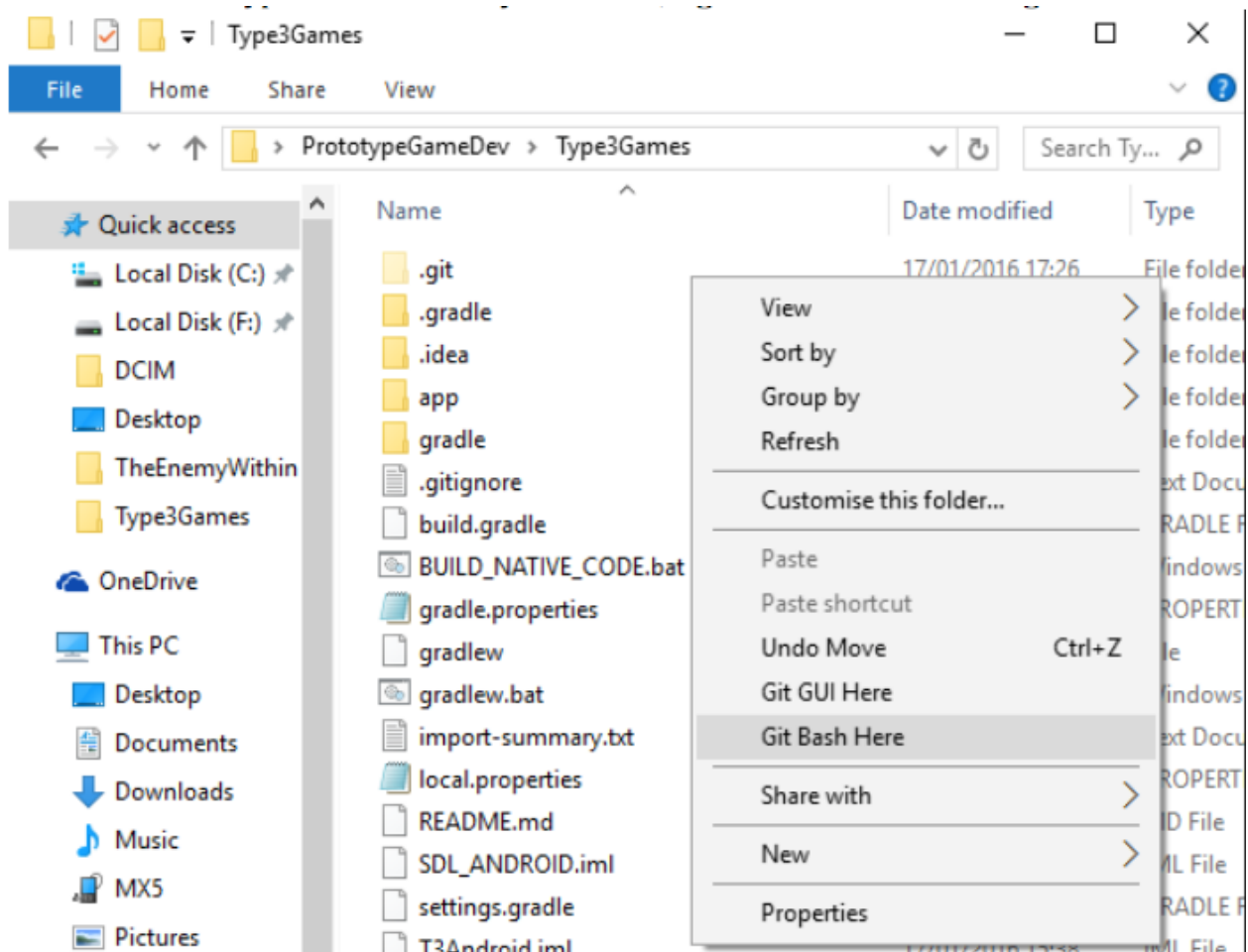
Saving your changes

There are two ways you can save your changes:

1. In the local repository on your machine
2. on the remote repository (on GitHub)

Saving changes locally

- Go to the Type3Games folder you created, right click in it and select git bash here



- In the console write `git add .` to add all the files you changed since the last time you saved your changes using this method. If you want to add only a few of the files you changed, write `git add path/to/the/file/nameOfTheFile.extension` for each file instead. Then Press enter.



- To double check what you are doing, you can write `git status` to list in green all the files you added and that will be saved in the next steps
- If you made a mistake and added a file you didn't want to add, you can write `git reset` to undo all added files, or `git reset path/to/the/file/nameOfTheFile.extension` to undo single files
- After adding the files you wanted to add, use `git commit` to save changes
- you can write `git commit -m 'write what you changed here'` to add a message to the commit
- If you don't add a message, you will be forced to do it in the Vim editor that will open. If you don't know how to use it and can write a concise message to describe your changes, I suggest you use the

first method, otherwise check the “Using vim” section of this document

- You can call `git status` again to check everything looks cool
- You can also call `git log` to see the commit history of your repository with the related commit messages. If the log does not fit in the window, you can navigate it with the arrow keys and you will have to press `q` to close it and go back to `git bash`.

Saving changes on github

- **Make sure you saved your changes locally first**
- If your `git bash` isn't open, go to the Type3Games folder, right click in it and select `git bash` here
- Write `git push origin android:android` to push your local android branch to the remote android branch
- You will be prompted to input your GitHub username and password
- If everything uploads fine you can stop here
- If the upload fails, it's most likely because someone pushed his changes before you, so the version you were working on is not up to date and `git` doesn't know how to behave. In this case the easiest thing to do is pulling from GitHub to get your local repository up to date. See the “Pulling changes” section of this document

Pulling changes

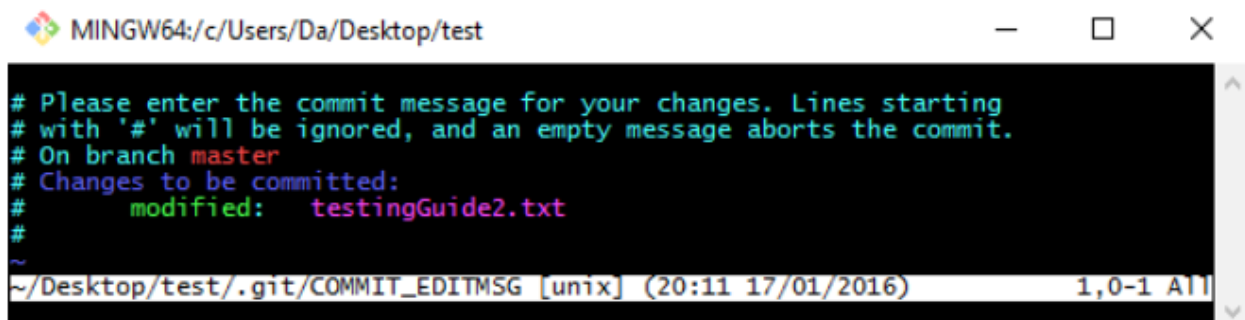
Davide Says: “I suggest you do this every time someone uploads changes, there will be an automated message in the programmers chat on slack”

- If your `git bash` isn't open, go to the Type3Games folder, right click in it and select `git bash` here
- Write `git pull origin android` to get any new changes from the android branch
- You will be prompted to input your GitHub username and password
- If there are any changes, they will be merged in your local repository. In this case you will also need to add a message with Vim (See “Using Vim” section)
- The files will download and merge, or “Already up to date” will be displayed.
- If you want to run the updated project remember to run `BUILD_NATIVE_CODE.bat` first (it's in the Type3Games folder, double click on it)

Using Vim

When committing changes or merging, you will need to add a message to describe why/what you did. If you are committing and have just a short description I suggest using the `-m` parameter as described in the “saving changes” section above since `vim` can be confusing.

`Vim` will open automatically in those situations and will look like this:



```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#       modified:   testingGuide2.txt
# ~
~/Desktop/test/.git/COMMIT_EDITMSG [unix] (20:11 17/01/2016) 1,0-1 All
```

- If you don't want to be here anymore, press ESC, write :q and press enter, or if weird stuff starts happening use shift+z twice to force exit.
- To add the message:
- press the insert key on your keyboard
- write the message, which should appear in yellow at the top of the page
 - when you're done press ESC on your keyboard
 - now write :wq and press enter to save and quit vim

- You can then call `git status` and/or `git log` if you want to double check everything is cool

Resources for Git and Vim

- Tiny Vim commands reference: <http://www.fprintf.net/vimCheatSheet.html>
- Git commands reference: <http://gitref.org/basic>
- Git explanations and tutorials: <https://www.atlassian.com/git/>

Schedule

Week	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Finish main engine features	x	x														
Engine on Android		x	x													
Cell generation / death		x	x													
Cell movement				x												
blood vessels				x	x											
Score system					x	x										
Mutations								x	x							
Menus									x	x						
Cell Energy											x	x				
1 hallmark of cancer												x	x			
Bug Fixing														x	x	x

About This Document

Compiling

This document was written in the [pandoc markdown](#) format. It is currently being compiled to pdf using the `md_to_pdf.command` file included in the repository. At the time of writing the file runs the command `pandoc TDD.md -o TDD.pdf --toc -fmarkdown-implicit_figures -V geometry:margin=1in --template latex.template -V urlcolor=Aquamarine`. `latex.template` is a custom template used to insert the Type3Games logo at the head of the file.

TDD Version Control

The document is held under git version control.

The repository is viewable at <https://github.com/Cyphre117/PrototypeDevTDD>.