# Project BUas: Ants

## Design concept- Frantic notation of concepts and ideas:

The first step in my design process was noting every single possible idea that came to me down, and seeing what sticks. The main ideas that were playing around in my head were the following:

* Something with random terrain generation, in whatever way possible. I had tackled a project like this before, but it was honestly done in a pretty tacky way (the way I did it was more Binding Of Isaac-esque, where I had a default set of rooms which I generated through a random number generator. This was done with C# and Unity, and would be very different this time around. Anyways, back on topic), and I wanted to somehow incorporate it into this project. The idea would have been some sort of randomly generated map, either a roguelike/roguelite, or some sort of survival game.
* The second idea I came up with was a puzzle game; although yes, I think pure puzzle games are quite boring and belong on mobile phones (that’s a joke portal 2 is one of my favourite games of all time), there are some interesting concepts here; I thought of a game where a character would shoot ants out of a gun. These shot ants would form clumps of ants, which the character could walk over or use as a bridge. The character would also have a vacuum to vacuum up any additional ants or vacuum their own ants back up. Another idea for a puzzle game was a game where you controlled a colony of multiple ants, and you’d have to get through obstacle courses without losing too many ants.
* What ended up being the concept I went with was something inspired by the game “Into the Breach”, which is an isometric turn-based strategy game where you control mechs to protect cities from invading aliens. I thought; what if I turn this concept on its head and have the player control the aliens (in this case, giant ants a la the tyranids from Warhammer 40k, but more ant-like). The goal would be to destroy human (or other fantasy races, for variety’s sake) settlements/power generators, all the while fighting off humans in giant mechs (which will possibly also be inspired by Warhammer 40k, at least visually).

So, in the current concept was as follows; an isometric turn-based strategy game, where you fight an AI controlled opponent in turn-based strategy combat. The goal would be dependent on what type of mission you are playing; destroy enemy structure, kill all enemy units, etc. You control a queen and several fighter-units. The queen is your most important unit (think of the king in chess). She can create new units (up to a cap), and once she dies the game is over for you.

So, once came to the conclusion as to what concept I wanted to realize, I began thinking more of some of the details. There could be varying factions you have to take over, perhaps a human faction which has incredibly beefy wall units which you have to either break through somehow or find a way around, or a faction of elves which instead use teleportation technology to zip around the playing field, making them hard to hit. So, I concluded I wanted there to be multiple enemy factions, which pose different levels of challenge. But, realistically, I wanted to finish just one level with regular humans before I add special mechanics, factions and multiple levels.

On a more technical level, I wanted the game to have an isometric perspective, for two main reasons: one, I think it looks better, and two; It would be more of a challenge to pull off than having a regular tile set, which seems like more fun than having it just be easy.

Some (admittedly far out of scope) stretch goals would be: a random level generator, where you get a random set of enemy placements for which you get rewards, some sort of upgrade system where you can customize your own troops with different upgrades or replace them all together.

## Design concept- In summary:

### In-scope goals:

-Isometric turn-based strategy game, where you compete against an AI controlled enemy.  
-You control various units, different fighters, scouts, and a queen which can produce more units (up to a cap)  
-Different mission types: destroy cities, capture objectives, destroy all enemy units, etc

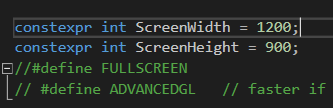
### Out-of-scope goals:

-Different types of enemy factions, which all have different powers and pose different challenges to the player  
-Upgrades to the player’s units, or different units the player can select. (different types of ants, long ranged ants vs melee ants, etc)  
-Randomly generated challenge, where you face off against a randomly generated map of enemies for rewards.

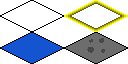
### Expected Difficulties

-Usage of C++. I’m personally more used to higher-level programming languages, such as javascript or java, so adopting C++ is going to be a bit of a struggle.  
-Implementing the enemy AI. I have never really tackled AI before (or at least not written one myself), so this will be quite difficult. I’m expecting this to be my biggest hurdle, but we’ll see.

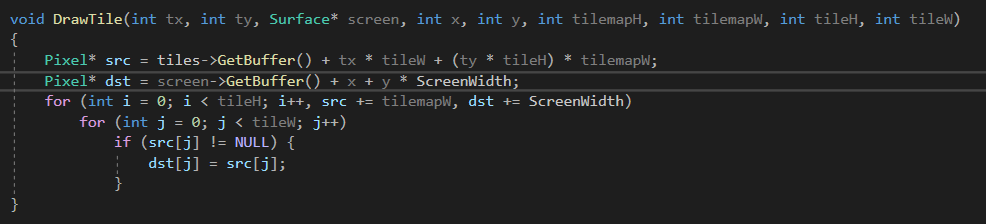
## Development- The beginnings 06/12/2020:

First things first; frantic googling about how to approach this. I already had some experience with programming at this point (mainly web-related, but I’ve made a game in C# before so the workflow isn’t all too alien for me), but I wanted to know what I was up against. First things first; I wanted to figure out how to resize the window. This seemed like quite a challenge at first, namely because I didn’t exactly realize there was already a setting for this in the template (the screenWidth and screenHeight variables, along with the FULLSCREEN and ADVANCEDGL defines, as you can see in the picture to the right). It took me longer than I’m willing to admit to   
figure this out, I even asked around in some C++ related  
community discord servers I’m in (discord server by youtube content creator and C++ developer javidx91, a name which will surely come up more down the line).

So, I’d figured this out. There wasn’t honestly that much reason as to why I wanted the window to be a little larger, but it I figured it was too small and I wanted to implement some sort of window resizing later on too, so this was a handy thing to find this early on.

After this was finished, I wanted to create the map for the game. I want the game to be isometric; which posed quite some difficulty, considering the fact that you can’t really use a normal grid, because the screenpixels and the gridpixels don’t line up. For example; in a normal grid, 0,0 would be the top left cell. In an isometric grid, this dead in the center at the top, and it grows diagonally. Luckily, javidx9 came to the rescue with a video2 about exactly this; isometric tilemaps. There was a problem with this video however; while it properly explained the mathematics and logic behind converting a regular grid into an isometric one, he used his own library in the project. And I didn’t want to add this library to the project, so I’d have to do It on my own.

First and foremost, I needed tiles to render. A quick trip on Aseprite3 to make some incredibly simple pixel art tiles, and after about 5 minutes I was ready with a very simple tileset (seen on the left). 3 regular, basic tiles for some color variation, and one ‘selected’ tile, which I would use to indicate where the cursor was currently hovering. After this was done, I copied a little bit of the tile rendering code from the Fast Track for Games Programming4 and some modifications (making it ignore transparent pixels through a simple if-statement), and it was done! I was able to render tiles.



(modified tile-renderer code, which admittedly: 1. Uses a lot of parameters and 2. Is too simple for the eventual purposes, but would suffice for now)

Another thing was, I noticed he used vectors in his video. I don’t exactly know why, but this prompted me to create a Vector2 class, which encompassed the most basic features of a vector I knew of. Mind you, it’d been a while since I’ve done math with vectors, so a quick refresher on sine, cosine and tangent was needed5. After figuring out how classes work again (I’d forgotten, again) through the use of w3schools4, I was ready to write my vector class (seen on the right).

I did the math by myself, which surprised even me, because at this point it was about 3:30AM and my brain was starting to slowly but surely give up on itself. But, I was hard-pressed to make this work, which it eventually ended up doing.

### References:

1Link to javidx9’s channel: <https://www.youtube.com/channel/UC-yuWVUplUJZvieEligKBkA>  
-References for learning C++/SDL2 sent by the people in his discord server:  
<https://lazyfoo.net/tutorials/SDL/index.php>  
<https://www.youtube.com/watch?v=QQzAHcojEKg&list=PLhfAbcv9cehhkG7ZQK0nfIGJC_C-wSLrx>  
<https://wiki.libsdl.org/APIByCategory>

2Javidx9’s video on isometric tilemaps: <https://www.youtube.com/watch?v=ukkbNKTgf5U&t=348s>

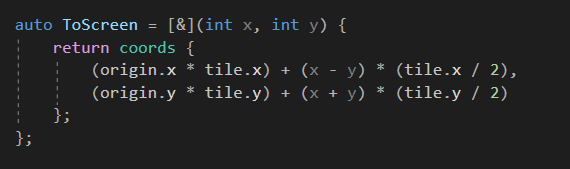
3Link to Aseprite on steam: <https://store.steampowered.com/app/431730/Aseprite/>

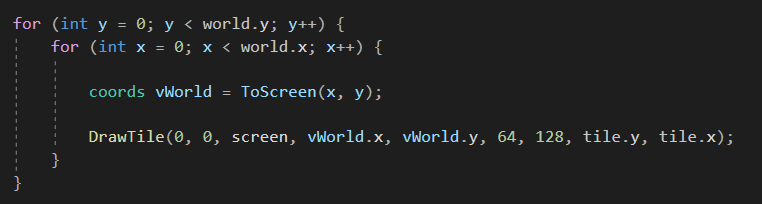
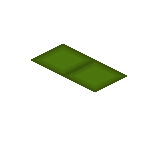
4Link to the Fast Track Part 11 about tiles where I copied and modified some code from:  
<https://www.3dgep.com/cpp-fast-track-11-tiles/>

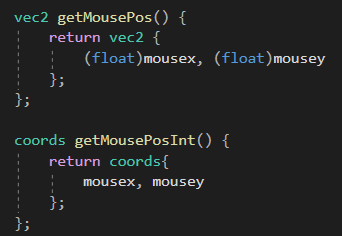
5Reference for Sine, Cosine and Tangent calculations: <https://revisionmaths.com/gcse-maths-revision/trigonometry/sin-cos-and-tan>

## Development- 08/12/2020

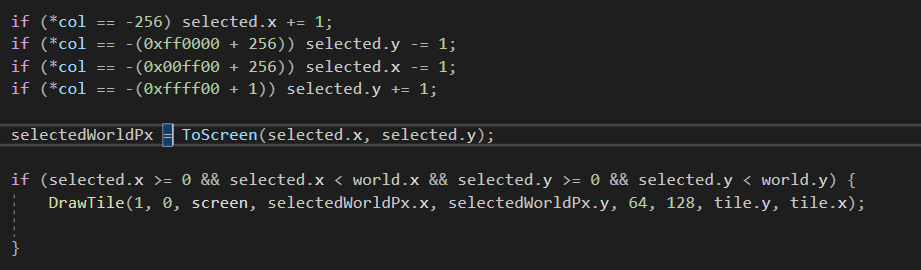
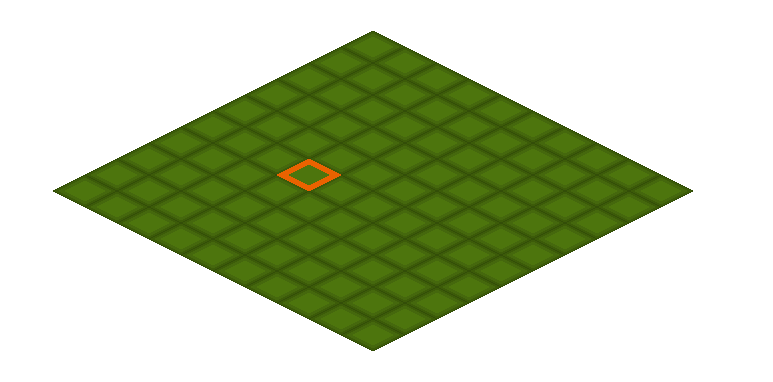
I took a break on the 7th (call it procrastination or clever work division), but on the 8th I got right back into it. I continued following the tutorial I had been following1. First things first; I realized the template had a built in vector2 class; so it was time to scrap my own code. This was quite a bummer; not only had I put in quite some effort (maybe a little too much effort for what it was worth, admittedly), but this meant a good portion of the time I spent on it on the 6th was essentially just wasted. But, this led to me finally taking a proper look and understanding the template code (I’d had prior difficulties with it due to me not understanding most of the SDL2 code in there). So, at least I got something out of it.

Now, for the tilemap; I’d been able to render 1 tile, even a grid of tiles, but there was a problem; this grid was only a grid in screen-space. So it didn’t properly look like a 3D square, as it should. So I had to make a separation between world and screen space, and a conversion method to do so. First things first was the arithmetic required for this; to generate a new properly next to the previous tile from an isometric perspective, I had to first get the tile height and width (which in my case was 32 by 64 pixels). Then, when generating a new tile, I had to offset it by half the tile width and height, in order for it to appear properly next to the previous tile. So, code-wise, the first step was to set up some necessary variables; I set up 3 main vectors which I would use to generate the world with; ‘world’ which held the size of the world, ‘tile’ which held the size of the tiles in pixels, and origin, which dictated the start of the generated world map in coordinates. So, next on the chopping board was to actually convert the world-coordinates to on screen coordinates; I did this with a small lambda function which took the world-coordinates, and calculated the screen coordinates accordingly. (seen on the right).

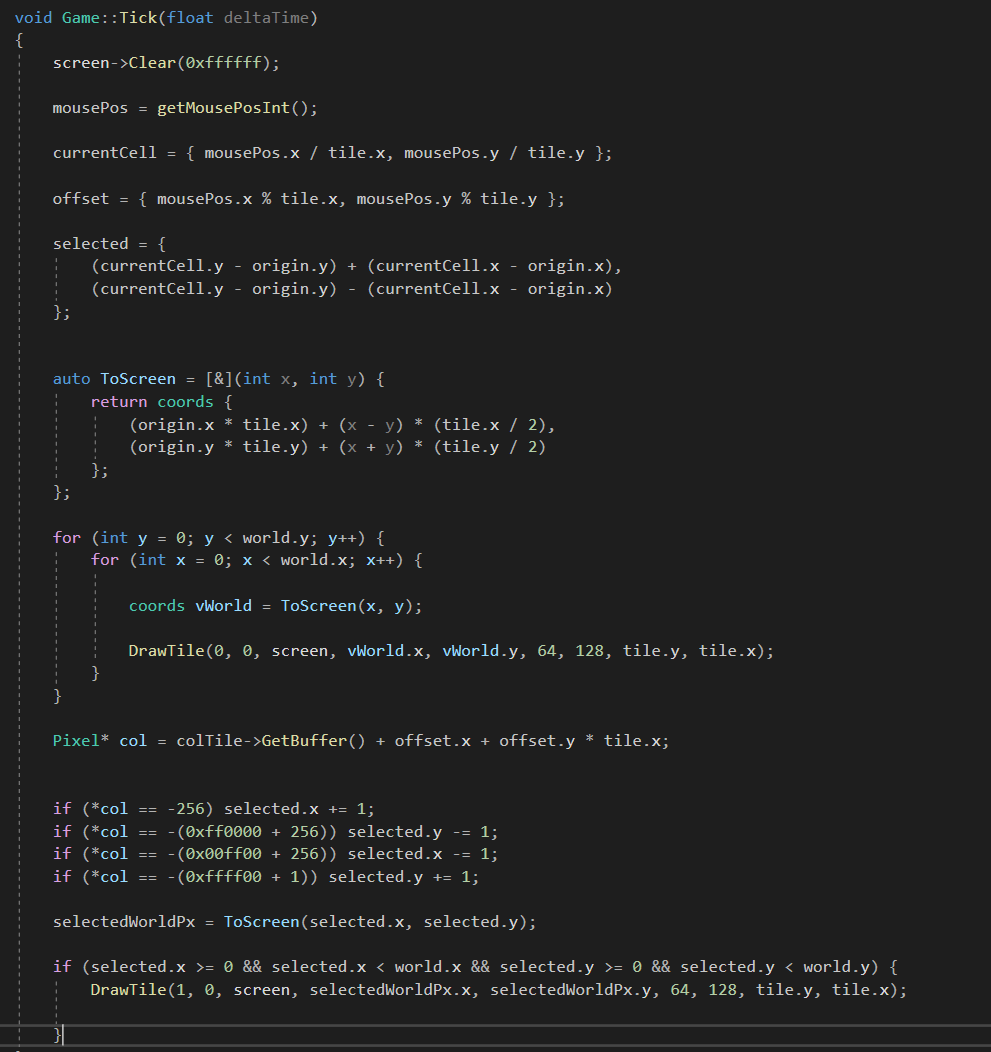
As you can see, it takes the coordinates of the worldspace (x and y), and turns them into on-screen coordinates as follows: It first adds the offset on the screen determined by our origin-coordinate. So in this case, the map would be generated starting at about the middle of the screen. Once the offset is added, it adds the offset for the tile to be generated correctly; take the x coordinate for example. Say, you increase your x coordinate by one. That would imply the tile generates a bit below and to the right of the tile (as seen to the right), more specifically it generates with exactly half of the tile’s width and height as an increment. After doing this with both the x and y factors (using addition with the y values because they have to be displayed to the left instead), and sure enough, after running the generation code in a loop we had our full map.

Above the loop in which it is generated. It generates the on screen coordinates from the world coordinates, and then draws the tile accordingly. What you might have noticed is that I am not using the vec2 that’s included in the template, instead using a struct called ‘coords’. The reason why is fairly simple; the vec2 class uses floats, while most of these values (the values of the world coordinates, for example), are integers. Rather than fiddling around with conversions which take time and memory, I made a quick struct called coords containing only two integers, namely X and Y, and used that for everything instead.

Now, for the tile selection; selecting a tile normally was fairly simple, I just had to check which tile the cursor was hovering over, and draw a ‘selection’ tile over it. This was done through adding the methods seen to the left to the Game-function, which would return the mouse position, either in floats using vec2 or in integers using coords. Although, this posed a problem; it was selecting the tiles in screenspace. What this meant is that, essentially, it only covered the tiles that occupied full screen coordinates, and not the ones in between. In essence: It only selected half of the tiles. Now, There would be two solutions for this; I calculate whether the cursor is located outside of the main tile through some complex math, or, as the video proposed, I use a different method;   
Using the tile seen on the right, using the offset of the tiles, I do something very simple; I check the color of the pixel in this image based on the location of the cursor within the tile. If I’m in the top left of the file, this would read a red pixel from the tile to the right, which would signify that I’m on a halfway tile. All that was left was adding this for every side, and then incrementing either the x or y coordinate with 1 based on what side it’s on.

above the code for this incrementation of the world coordinates based on the colour of the tiles. I added 256 or 1 to the code here and there and turned the values negative, because it didn’t exactly match with the colours on the tile (I was able to get the colours of the tile by reading them out and printing them to console).

Above is the complete tilemap with a single selected tile. On the next page is the complete code for the generation of this map. (I did a few texture changes to make things look a tad cleaner, as the previous tiles looked kind of bad.)

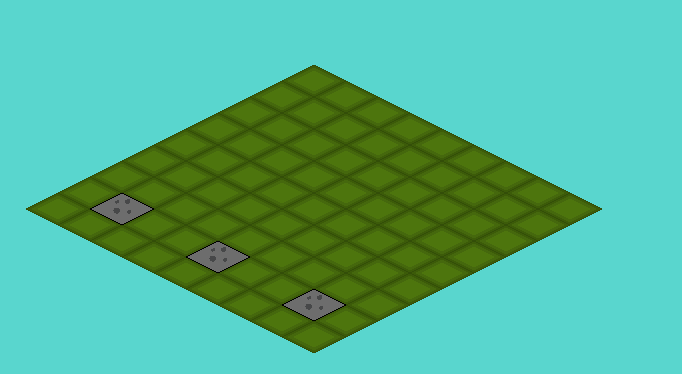
Above is the entire code for the generation of this tilemap. A few unexplained parts are

* GetMousePosInt() is a method I added to get the value of the cursor position in integers
* currentCell is the current cell the cursor is in in screenspace
* offset is where in the current cell the cursor is located

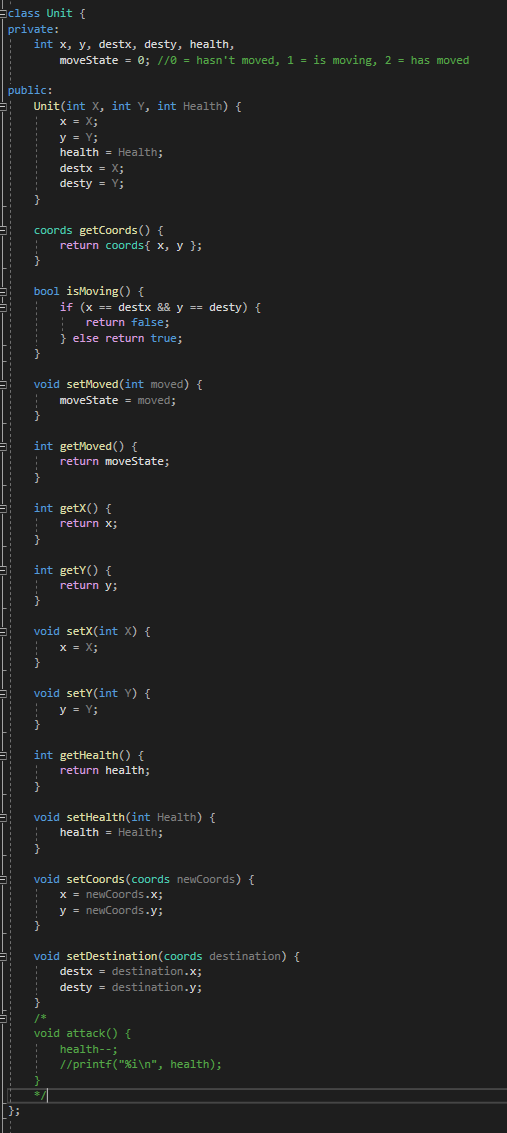
After this I limited where the selection tile could appear so it only appeared on the map to prevent the program from crashing once I went out of screen space with the cursor, and there we had it; a perfectly functional tile selection process. Next to do was the interaction with the tiles through clicking, but that would be for a later date since it was around 3AM again and I had work the next day.

## Development- 16/12/2020:

At this point, there had been a lot of small changed I’d made over the ~10 days that had passed, but overall I hadn’t been as productive as I would’ve liked. While this was due to various school- and work obligations, procrastination also definitely played a part in it. But, to recap;

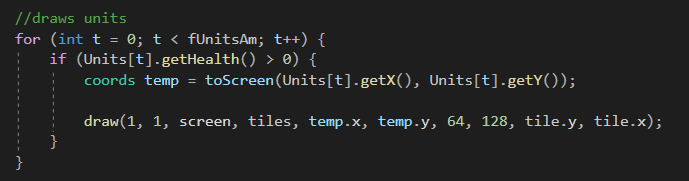
I had created a ‘unit’ class, which holds all the data a unit needs (well, at this point it didn’t, because I wasn’t finished with writing it). I struggled a lot with figuring out the header file, where the main problem I faced was as follows; I want access the “drawTile” function (which had now been renamed to draw()) from the ‘Unit’ class in the header file. I tried moving drawTile(), but this would give errors since it used various parameters of datatypes the header file didn’t recognize (Surface, Pixel). Even when implementing them I encountered errors. After trying for what felt like hours, I decided to do some googling. I found various1 sources concerning the topic, and one thing quickly became clear to me; I shouldn’t have even been trying to do this, seeing as the draw() function would have to exist in game.cpp anyways, so there was no use declaring it again in game.h. So, with this out of the way, I finished up the Unit class and created 3 units that would be placed on the board. (Units seen below, represented by the gray tiles for now).

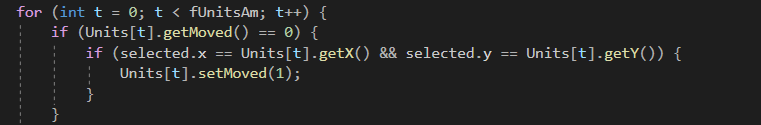
(Code for the unit class on the next page)

Let’s go over what was included in the Unit class at this point (this point being, the final rendition for this session, as I didn’t really log much of the previous renditions, which, admittedly, was stupid of me.); for attributes, it had:

* x – current X coordinate on the field
* y – current Y coordinate on the field
* destx – X destination for moving
* desty – Y destination for moving  
  (I will go more into moving later on)
* health – total health of the unit
* moveState – the state of movement, I will go deeper into this later.

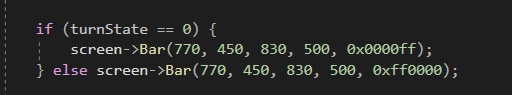
I declared an array of 3 friendly units, which all held coordinates for where they would be placed down. Once they were placed down, I needed to somehow find a way to move them. I dove into this, admittedly, way too quickly without considering the logic behind it. I started out by looking how to register mousepresses, which I found in the template. There was a slight problem I faced; the mousepress had to interact with variables stored in the main file itself, so it couldn’t just be a method. I solved this in an admittedly roundabout way, a real brute-force solution, but it served its purpose for now. What I did is; I declared a function prototype in the header file, which the method would then call when the mouse was pressed, and which would then be declared further in the game.cpp file to use the variables stored there.

Now, once it registered mouseclicks, moving it was fairly simple; by changing the x or the y coordinates of the unit directly, it would move to the new x or y coordinates. Fairly simple stuff (also seen in the draw command below with which I drew the array of units). This would loop over the array and draw each unit individually, at their respective coordinates.

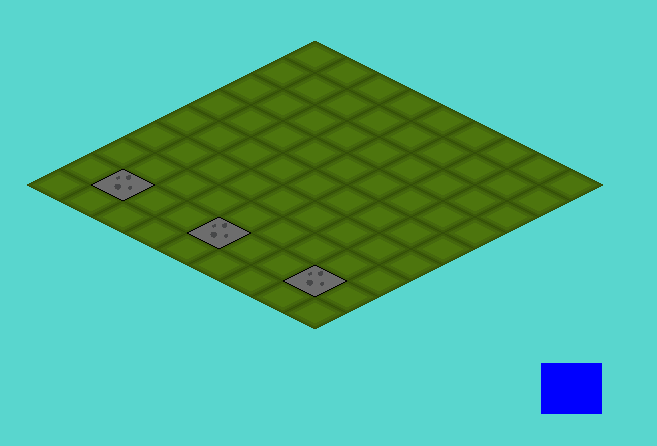
So, to test some things, at first I had the x-coordinate increase by 1 whenever the button was pressed. I used the same selection method as I used earlier to indicate hovered tiles, and used the currently selected variables to check if the cursor was hovering over a unit. 

Above is included in the function doSomething(), which is called on mouseclick. It compares the selected tile to the coordinates of the Units, and moves them by 1 on the X axis when clicked on. So, this worked for now, I was able to move the units along the X axis when I clicked on them. Great. But this was the point where I actually started thinking about the general flow of the game. So, to break it down, the gameflow I came up with goes as follows:

* Player starts their turn in a move-phase, where they get to move their units anywhere on the field.
* The player moves their units by first clicking on the unit he wants to move.
* The player then selects a tile the unit should move to, depending on whether the unit can actually move there. The units themselves will function kind of like chess pieces, with different movesets, e.g. some can move straight forwards, others can only move diagonally, etc. For now, the units are only going to be movable straight forwards in one of 4 directions.
* Once the desired tile is clicked, the unit moves there by increments, as if to play a walking animation.
* Once the unit arrived at its destination, it can no longer move any more.
* Once all units have moved, the player enters an attacking phase and can attack with their units.
* This attacking phase will follow the same principal; the player clicks their units, and then selects a direction for their attack to take place in. The units or objectives in the selected tiles then take X amount of damage.
* Then the enemy takes their turn.

So, not only did I realize that this project would be immensely more complex than I initially gave it credit for, but now I had to figure out how the player would move their units using increments. First things first; to establish a turn-flow, I added an integer variable called turnSate. The value of this variable would determine where the game currently is; 0 being movement phase, 1 being attacking phase, etc. Now, once this was in place, I added some more things. I added a button to the right of the field, which would change the state of the game manually. I intended to keep this button for the player to manually skip his turn. 

Here is the code to generate the button; nothing fancy, just a bar that is red when in the attacking phase and blue in any other phase (there were no other phases at this point).

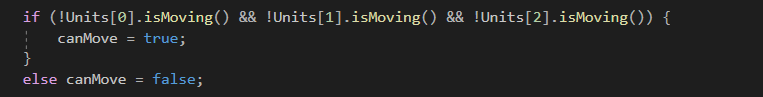


Above the field, with the button included.

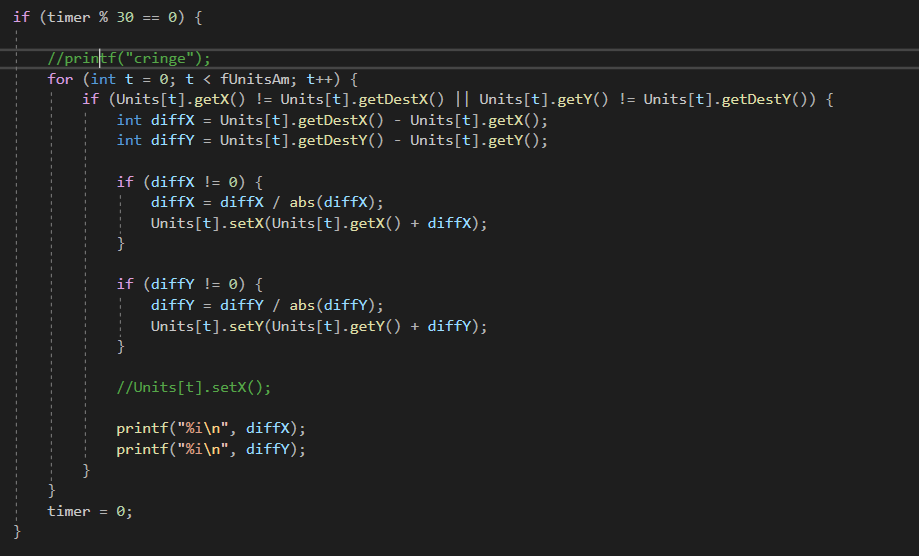
Once this was implemented, I could go and work on the actual movement of the units. First things first, this was the point where I added the “destx” and “desty” variables, instead of changing the x and y variables directly as I had previously been doing. I also added the moveState variable, which would indicate in what stage of movement the unit currently is; 0 meaning the unit hasn’t moved, 1 being the unit is currently moving, and 2 meaning the unit has moved and can’t move anymore.

Now, there were a few things I had to make sure;

* The player shouldn’t be able to move his units to a spot occupied by another unit
* The units should move over time, e.g.: 4 fields per second, to indicate a sense of movement rather than teleportation.
* The units shouldn’t be able to move when another unit is moving
* The units should have a max range to travel to, which should be indicated when clicked on them.

Now, the first problem on this list I tackled was the units moving over time. I had been able to make units move, and making them move over time required a little change in how I handled it. While previously I had just made the units increment whenever the player clicked, or had them teleport once the player had selected the new position, now I had to increment them over time. This was actually surprisingly easy, all I did was add the aforementioned ‘destx’ and ‘desty’ variables, which would create a destination for these units to go to. So, now I could do another thing; I could check if a unit is in the process of moving by checking if the current coordinates and the destination coordinates were the same or not. If they were; the unit wasn’t moving. If they weren’t; the unit was moving and no other unit should be able to move. 

Above is the code that checked whether all the units are moving or not. If they aren’t, the canMove Boolean would be set to true, indicating a new unit can move and vice versa.

So, now all that was left was to add the code to increment the position of the units over time, towards their destination. This wasn’t hard at all; I had the destination coordinates and the current coordinates, so all that was left for the units to be able to move properly was to put a move function in the game::tick method, do some calculations and voila! There it was, the perfectly functioning move phase (Although, it still didn’t account for obstacles such as other units, nor did it have a max range yet, but those limitations would be for later). 

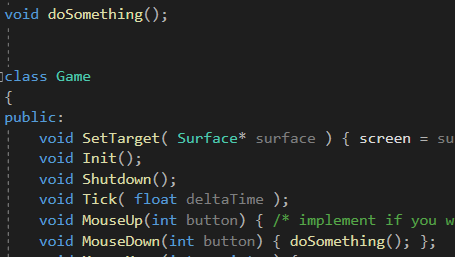
Through using a timer which would increment by 1 on every tick, checking if the modulus when divided by 30 (this could also be 60, 20, or any other whole number depending on how fast the units should move), and resetting it afterwards to avoid possible future errors concerning integer overflows, we can make this code execute twice a second (60 ticks a second, every 30 ticks the code executes = 2 times a second). The math here is also fairly simple, we get the difference between the current X and Y coordinates and the destination X and Y coordinates, divide it by its absolute value to get a value with a range of [1, -1], and make sure there is an exception for when the value is 0 to avoid division by 0 errors. And, there it was; functional movement! At this point I felt satisfied enough for me to get some sleep, as it was again around 3:00AM.

1Sources about header files:  
<https://stackoverflow.com/questions/25274312/is-it-a-good-practice-to-define-c-functions-inside-header-files>  
<https://stackoverflow.com/questions/16313174/c-cannot-i-assign-values-to-a-variable-in-header-file>  
<https://stackoverflow.com/questions/38942013/declaring-variables-in-header-files-c/38942057>

There weren’t much resources this time around, as most of the coding was me fiddling around without really looking anything up.

### Reflection

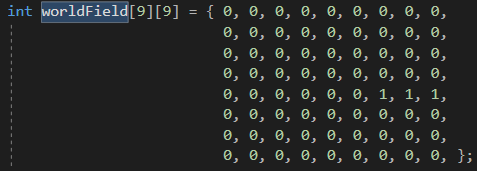
I was definitely unproductive these last 10 days. While work and school certainly played a role, I didn’t work on this project as hard as I ought to have over these last few days, and I intend to change that. I also made a large error not looking into how the game would flow, and instead wanting to make the units move so early without properly thinking about it. It could’ve saved me a lot of unnecessary work and time if I had just sat down and thought about it.

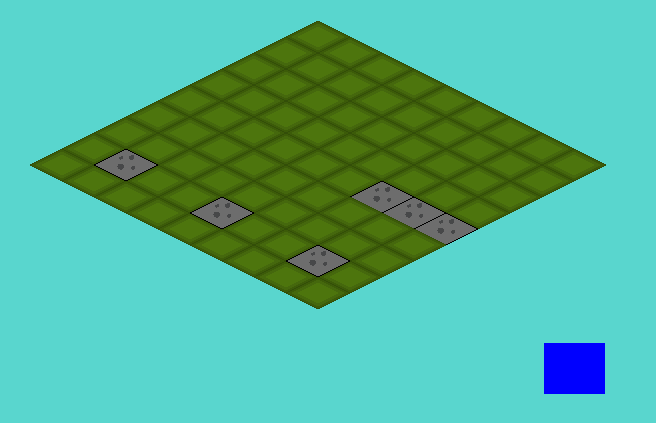
Additionally, I struggled a lot with figuring out what exactly should go in the header file and what shouldn’t. I eventually figured it out (although I’m still not a fan of this particular piece of code, where the doSomething function gets called from within a method, only because I don’t know exactly how to have the method interact with variables which exist in game.cpp. I’ll be looking into this for sure in the future.)

Aside from that, I’m really happy with how simple the movement itself turned out to be. I do imagine that setting the limitations and checking for terrain will be a little more challenging, but we’ll see.

Another thing I can do better is the logging of my progress; I made a lot of small changes over the course of these 10 days, which I always deemed too small to log, but which ended up piling up and becoming a pretty big change in the end. This has also led to me not noting down every single source I used over these last 10 days, which is probably the part I’m the least happy about. I tried looking through my browsing history to find the sites I used, but to no avail, aside from some very basic stackoverflow pages, out of which I already listed the most important ones. But, I intend to change all of these things in the future.

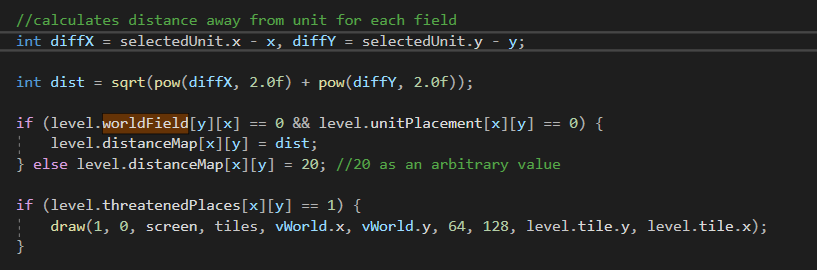
## Development 22/12/2020

By now units were able to move, but a few restrictions still needed to be implemented. Mainly; a maximum range for units to move based on the speed of the individual unit, and some sort of way for units not to pass through terrain/detect collisions. While the first part went rather well, the second part really ended up being quite the challenge. First things first, I wanted to see how to handle this situation. I initially thought to count the amount of steps required to get to a certain place; but this quickly got scrapped for a simpler solution. I added two 2 dimensional integer arrays. One called distanceMap which is the same size as the map, initially filled with 0s, and one called worldField, which would essentially contain the layout of the map, 0s meaning walkable ground, 1 meaning walls, etc. This array was kept to only those 2 right now, but would be expanded later with various different types of tiles. 

Above the map layout for testing purposes, which results in the following map:

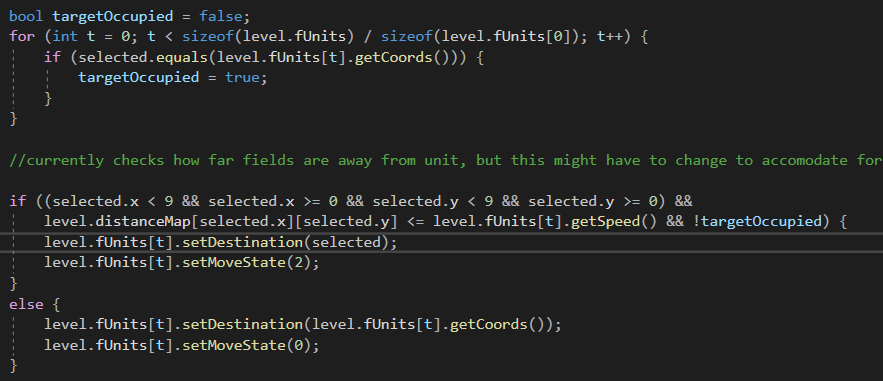
The player’s units should neither be able to pass nor land on these tiles.

With these arrays implemented, the following task was simple; set up a maximum range for the units, make sure they can’t land on terrain tiles nor other unit tiles, and make sure they can’t pass through these tiles. First things first was setting up a maximum movement distance for the units.

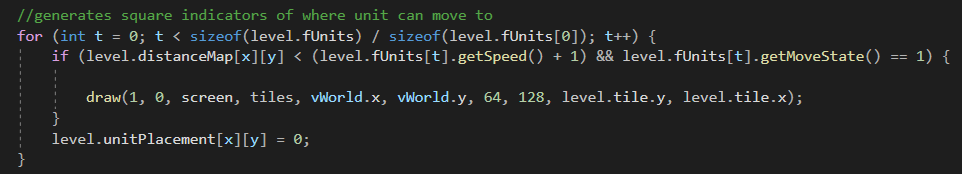


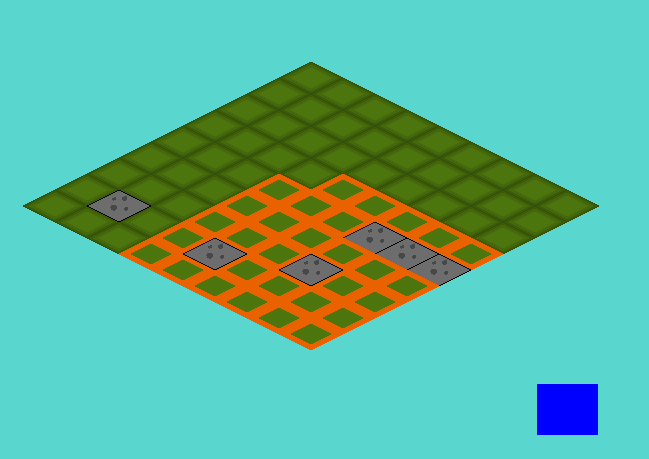
This code happens as the map is being generated, so will automatically loop over every cell. To break this down; the first thing that happens is the calculation of how far every cell on the field is from the selected unit. This is done fairly simply, just using Pythagoras. Doing this fills the array with numbers; 0 for the field the unit is on, 1 for every field right around the unit, 2 for every field beyond that, etc.

If the field at the given coordinates has a value of 1 on the worldField array, it gets a distance of 20. This value is completely arbitrary, and just had to be a value that the user couldn’t possibly access. (Note; the x and y values of worldField are swapped; this is simply for clarity’s sake in declaring the array)

When the distanceMap array is filled, I added a ‘speed’ attribute to my units, which would indicate how far a unit could move on the map. Once I added this, all that was left for me to do was add an if statement comparing the 2 values.

As can be seen in the code above, which gets executed in the doSomething function as described earlier, which happens when the player clicks, checks if the value on the distanceMap is less than or equal to the speed of the unit. Based on this, the unit can either move this distance, or, when clicked outside of its range, cancels its movement. This allows for very quick and simple movement of your units, and is also very easy to cancel if you’ve made a mistake.

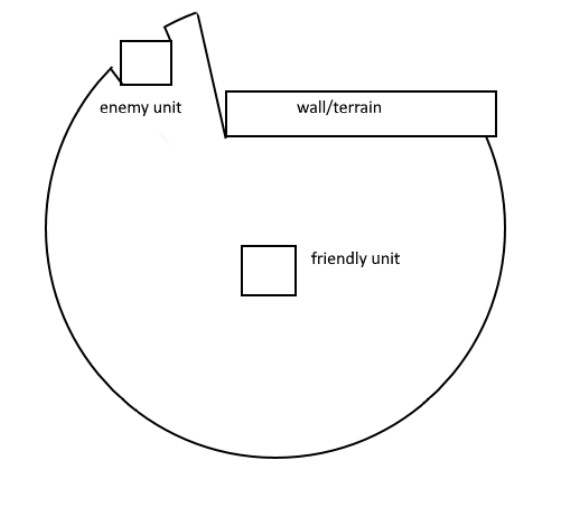
There is another piece of code there, which determines whether the destination of the unit (called ‘selected’ for selected space) is occupied by a friendly unit or not. After this, I wanted to implement some sort of indicator for where the unit could move to. So, in the map generation, I added this piece of code:

So, as it generates the field, for the selected unit, it generates all the fields the unit can move to. This looked something like this;

Which, for the time being, I was happy with. I decided to wrap up here, while I didn’t really accomplish much this day, this would suffice. There were still some problems; units could phase through other units or terrain as long as the destination wasn’t on another unit or on terrain, but at this point I decided to stop working for the day.

## Development- 24/12/2020

For this day, the plan was originally to implement collision. The idea was; if a wall or enemy unit obstructed the way, all fields that would be available behind them would become unavailable (see picture, but it would be simpler in game due to the grid).



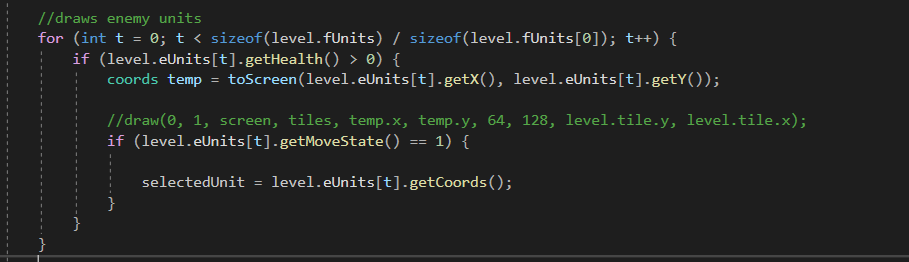
There were a few ways I initially thought to handle this: At first I thought of looking up ways dynamic lighting is done from a top down view12, and to try and replicate this. Secondly, when this proved too advanced and complicated, and just generally not really applicable, I looked into some other solutions. I thought of using something I’ve grown used to using through working with unity, namely raycasts. I figured trying to create something similar that would somehow mark all the tiles it passed, which would stop at tiles that aren’t indicated with 0 on worldField. This, however, also seemed to complicated for this. I broke my head about it the entire day, but never ended up actually doing any work, and ended up putting it off and deciding to work on it a different time.

1 <https://www.reddit.com/r/gamedev/comments/bpqqdg/2d_dynamic_point_light_c/> reddit link going into this topic

2 <https://www.youtube.com/watch?v=6B1IA_Tera4> video which ended up being pretty pointless, and I only really skimmed through it in the end.

## Development- 26/12/2020

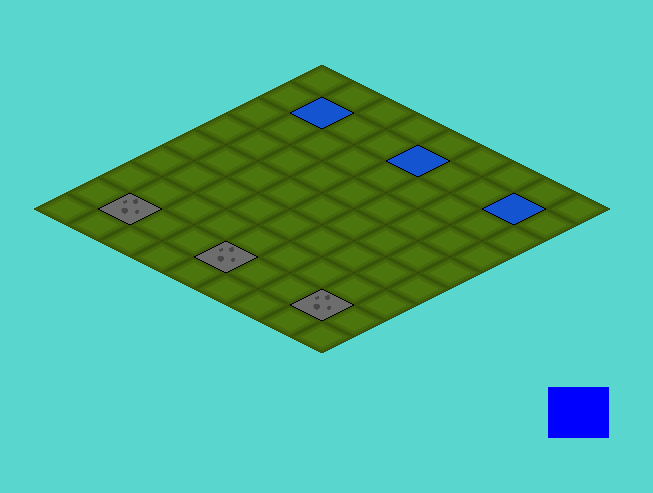
Putting off movement for a moment, I decided to instead focus on creating enemy units and just the combat system in general. First thing I did was create an array of enemy units, which would be indicated by blue squares (as seen in the picture). These enemy units would remain completely stationary for now, but would come back later when working on the combat.



It just uses an array called eUnits (The screenshot was taken later, hence why it is also inside ‘level’, this will be elaborated on later). This array is basically a complete mirror of the fUnits array, with the same exact attributes, namely;

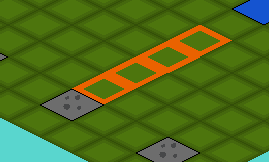
-X and Y coordinates  
-destination X and Y coordinates to function with the movement  
-health: the amount of damage a unit can take  
-speed: the maximum amount of spaces a unit can move  
-movestate: the state of the unit, either; hasn’t moved, is moving, has moved  
-isMoving: a check whether or not this unit is moving or not

Not that anything would be done with these enemy units for now, aside from them standing still, functioning as dummies for the friendly units to attack.

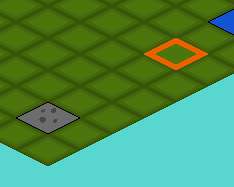


Additionally, I added a few parameters, which would check whether the destination of the friendly unit overlapped with any of the locations of the enemy units. So, to actually get started with the combat; I decided that, like in the game Into the Breach that this was in the end partly based on, for units to have 3 health (at least for now).

I had 3 ideas for different attack modes, namely the following;



A long range attack with low damage, namely dealing only a single point of damage, which can hit multiple enemies at once.



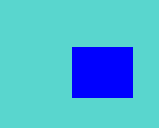
A long range mortar strike attack, dealing a medium 2 damage at long range.

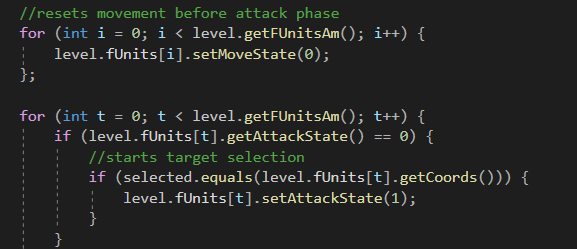


A short range melee attack, dealing a full 3 damage.

(These values were not set in stone at this point. Additionally, these screenshots were taken afterwards, for demonstration purposes).

I wanted these attacks to function as follows; as opposed to how the movement behaved, where units could move freely within a certain range, I wanted these attacks to have very strict patterns, and only work in either of the 4 main directions. How this exactly would be done was as follows; when the turnState variable was 1 (indicating it was the attacking phase), clicking on a unit would both reset their movement completely, just to be sure no odd bugs showed up while attacking, and then turn attackstate to 1, provided their attack state was 0. This was done in the same doSomething() function that was called for movement. Currently, I used the button I had implemented to switch between attack and move phase.

  
(Button for switching between attack and move states)

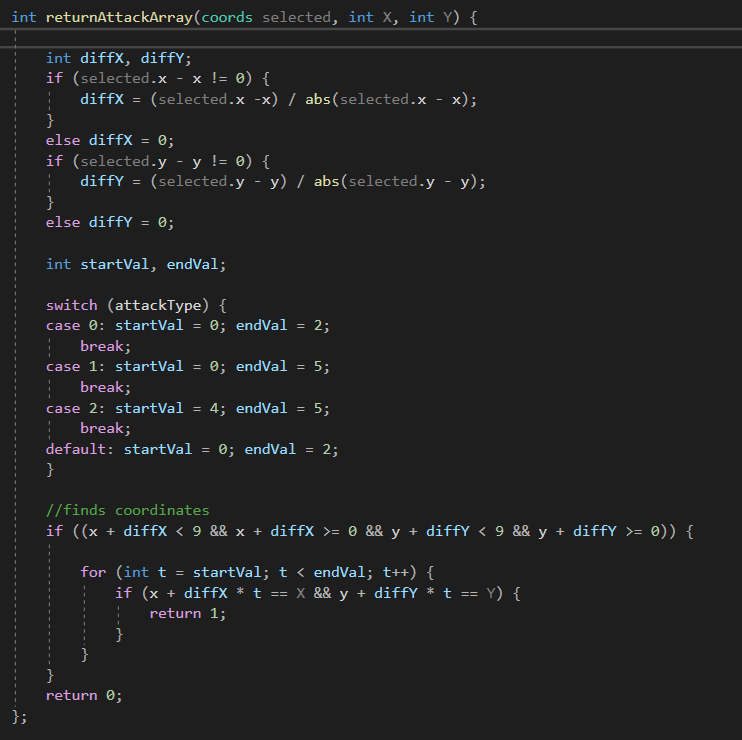


Seen above, the movement gets reset and then the attackState of the selected unit is checked; if it’s 0 as seen right here, it sets it to 1. Now, when a unit has an attackState of one, it has initiated its attack and is selecting a target. I knew that doing something similar as what I did with the movement would probably work, thus I went with that. I again made an array spanning the entire field, in this cased called threatenedPlaces, which would indicate all the threatened squares with a 1 and all the other squares with a 0.

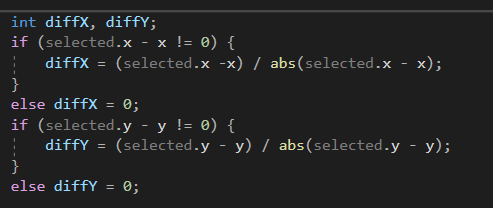
Having said that, this was easier said than done. My initial plan was to add a method to the Unit class called something along the lines of ‘getThreatenedArray’, which would return a 2D array containing the entire threatenedPlaces array. I found 2 sources, one on Stackoverflow1, which I ended up basically copy pasting, and a video on youtube2, which never really ended up helping.

But returning a 2D array from a method deemed no easy feat. What essentially had to be done to accomplish this effect was to create an array of pointers, which all pointed to different arrays, essentially creating a 2D array. This ended up only half working, as I struggled a lot with actually getting the array from the method in ‘Unit’ in game.h to a variable I could use in game.cpp. It gave me a plethora of errors (of which I sadly have no screenshots as this was written after the fact and my code had already changed, regrettably), mainly one telling me I couldn’t copy an entire array into another one. After doing some debugging (aka logging the entire array through prints), I discovered the variables returned were pointers themselves. After doing a whole bunch failed problem solving, I attempted a different approach.

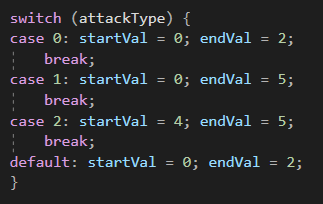
Instead of attempting to return an entire array through one method, I instead had the method determine whether a single field on the board was threatened, based on some given parameters.



What happens here is fairly simple, and I’ll go over it step by step.



First things first, it got the difference between the selected field (the field the cursor was currently in) and the given parameters. If the difference wasn’t 0, I could get the normalized value out of it (either -1, 0 or 1). I had to use this roundabout and, honestly kind of ugly, piece of code to make sure no dividing by 0 errors happened at this stage.



After this it checked what type of unit we were dealing with, according to a new attribute of the Unit class, which determined the attack type of the unit. Using this, I could set a certain starting and ending range of the attack, or I could potentially add even more complex attack structures in the future, but for now I would limit myself to these.

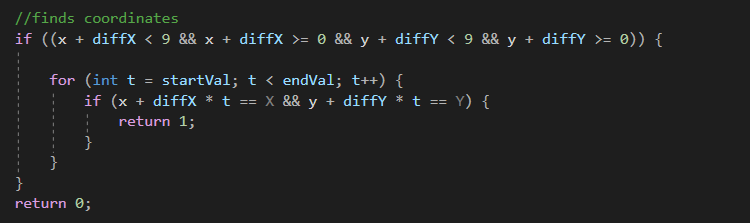
It’s fairly simple;

0-2 for melee, aka only the field in front of the unit,

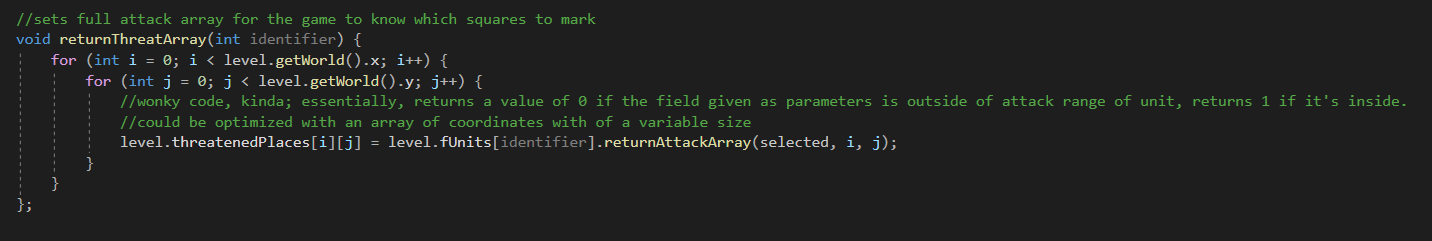
0-5 for ranged, aka 4 fields in front of the unit,

4-5 for mortar, aka 1 field in front of the unit, 3 units away.

After this it was only a matter of checking whether these values actually existed, and then calculating whether the field given in the parameters is a threatened one or not through this piece of code, which returned 1 if it was and 0 if not.

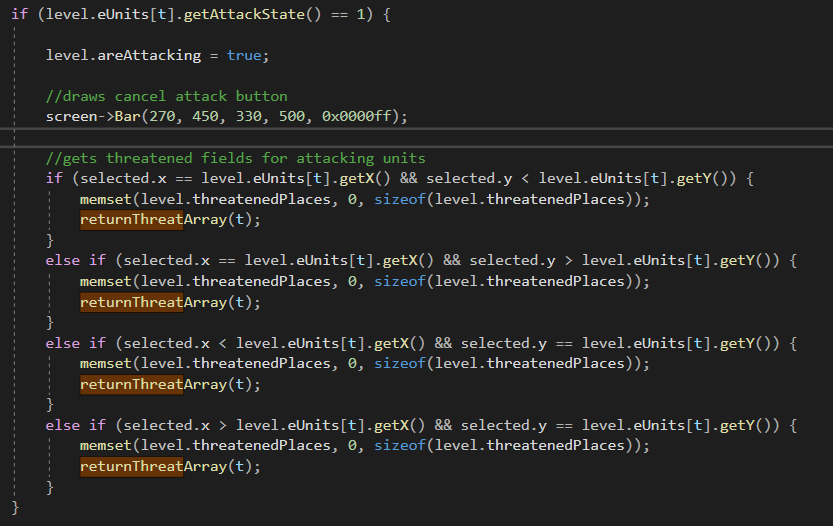


With this, all I needed to do to get the threatenedPlaces array was call this method;

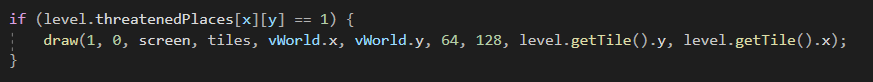
  
(The comment is cut off but is non-important)

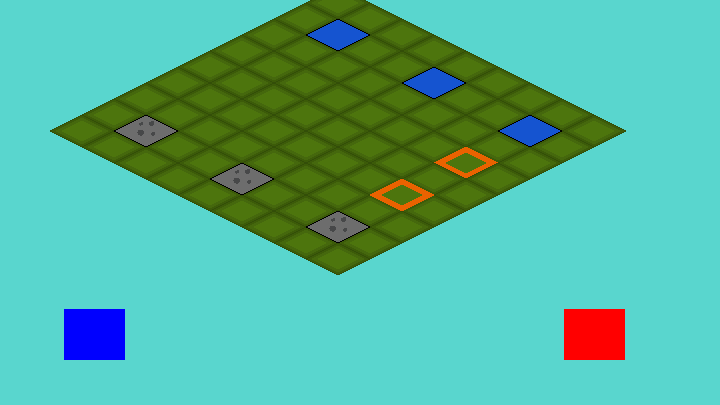
First things first, it takes the ‘identifier’ parameter, which essentially just means the index of the Unit that is attacking in the fUnits array. So for the first unit in the array, this would be 0, etc. Then it just goes through 2 loops, the vertical and horizontal length of the board, and fills up the threatenedPlaces array as it goes along, and ends up creating the entire array like this.

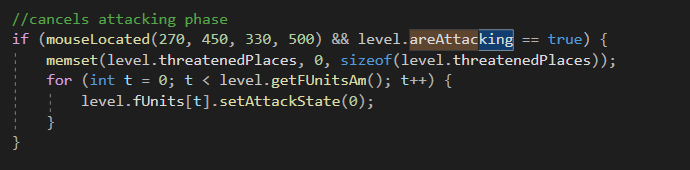
Following this, I had to call this function. So, in the piece of the game::tick loop where it loops over the units to draw them, I conveniently added a piece of code that would call this function when a unit is attacking.



This is again not too dissimilar from how I handled it with the movement, although this time there are a few complications. First thing at the top is a variable called areAttacking. This is essentially just used in the doSomething function to check if units are attacking. Then it draws a bar. This is because; later on in the code I made it so the attack can not be canceled like one would cancel movement, e.g.: by clicking off the board or outside the range, so instead I had to find another way to let the player stop their attack. This was done by adding another button to the left of the board, which, when clicked, cancels any and all attacks being made by the player. The drawing of the attack indicators happens just as the movement indicators are drawn; when generating the board; if the threatenedPlaces array is 1 at the given x and y coordinates, it draws an indicator square.

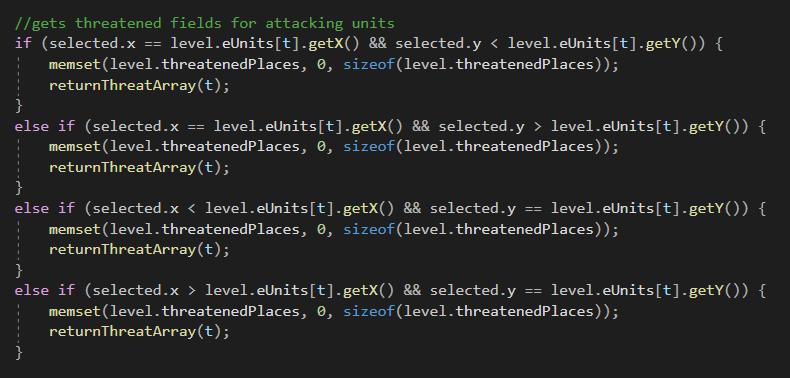


  
(Cancel button can be seen on the right. Note; there are 2 orange squares, one is the mortar target, the other one is just the cursor.)



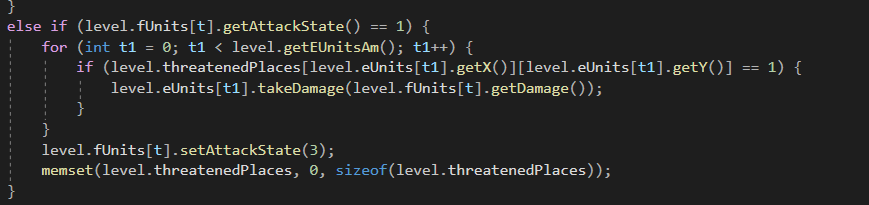
And this is the code for said button, located in the doSomehitng() Function. It checks if the cursor is on the button and if the areAttacking function is true, at which point it enables the button’s functionality. When clicked, it sets the threatenedPlaces array to all 0s, using memset34.

Now this function is out of the way, to get back to the code shown earlier, namely this code;



What exactly happens here is as follows; whenever the cursor hovers any distance over any of the 4 directions, the attack angle is set to that angle and stays there until the player hovers over another direction.

So there it was; all that needed to happen now was check if the attack actually hit. This was surprisingly easy; whenever a unit is attacking, when the player clicks anywhere (aside from the cancel button), the attack goes off, and when it does, check whether any enemy unit is located on a field where threatenedPlaces equals 1; et voila! They took damage when the player clicked again.



### Sources

1 Source to the stackoverflow page about returning 2D arrays: <https://stackoverflow.com/questions/8617683/return-a-2d-array-from-a-function>

2 Youtube video about passing 2D arrays: <https://www.youtube.com/watch?v=QEKmS221MtM>

3 Stackoverflow page about setting an entire array to 0: <https://stackoverflow.com/questions/5636070/zero-an-array-in-c-code>

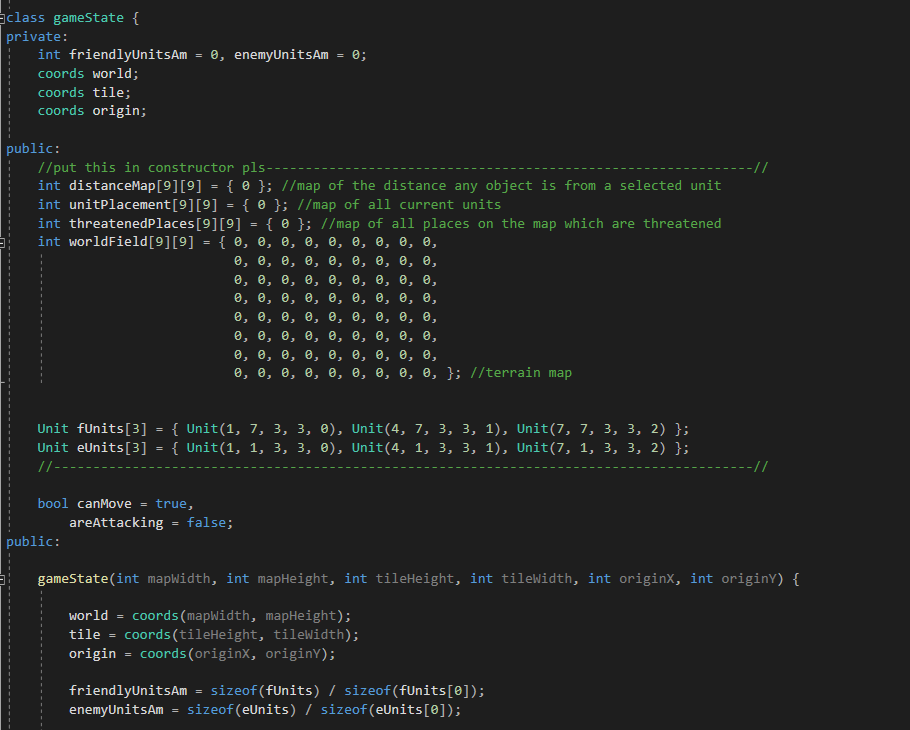
4 Reference page about memset: <http://www.cplusplus.com/reference/cstring/memset/>

### Reflection

Everything went really smooth this day, and a lot of work got finished. At the end of this worksession, I decided that I wanted to put every global variable I had (of which I had a lot, mind you, e.g.: threatenedPlaces, placesToMove, etc) into a single class, but I decided to do this another day.

## Development- 28/12/2020

With the combat finished for now, I wanted to do some cleaning up in my code. I had a lot of global variables at this point, which I all wanted to add to a gameState class.



This is the gameclass I ended up with after adding any variable that I deemed to belong in this class to the class. This was an extremely simple, yet tedious and boring process; I essentially had to change every mention to variables in the code currently, and I still was not happy with the state of this class at this point. The main problem is the fact that all the arrays that are used (distanceMap, unitPlacement, etc) are both set to public and their values are hardcoded. The reason for its tediousness however was that, in the game.cpp file, I had to replace every occurrence of these variables to one that works with the new setup.

Now, this wasn’t good yet; I still wanted to clean this up by making both the unit arrays and the map arrays private. Additionally I wanted to make both the unit and the map arrays variable in size, in order to quickly and more easily change the layout in order to create more complex levels really easily and quickly. This would be tricky, as C/C++ doesn’t really allow dynamically sized multi-dimensional arrays, and passing them through a get-method would be one hell of a task to take on. I knew it was possible, or at least I knew something similar to what I wanted to achieve was possible by using pointers1, however, at this point my exams were slowly coming up and I started to get less and less time to actually work on the game, thus I decided to leave it at this and continue working after the exams.

I knew I’d have to do a lot of tedious work in doing this, and I decided I would rather spend my time studying than I would spending it fiddling with variables, for the time being. I would, however, return to this after the exams as soon as possible.

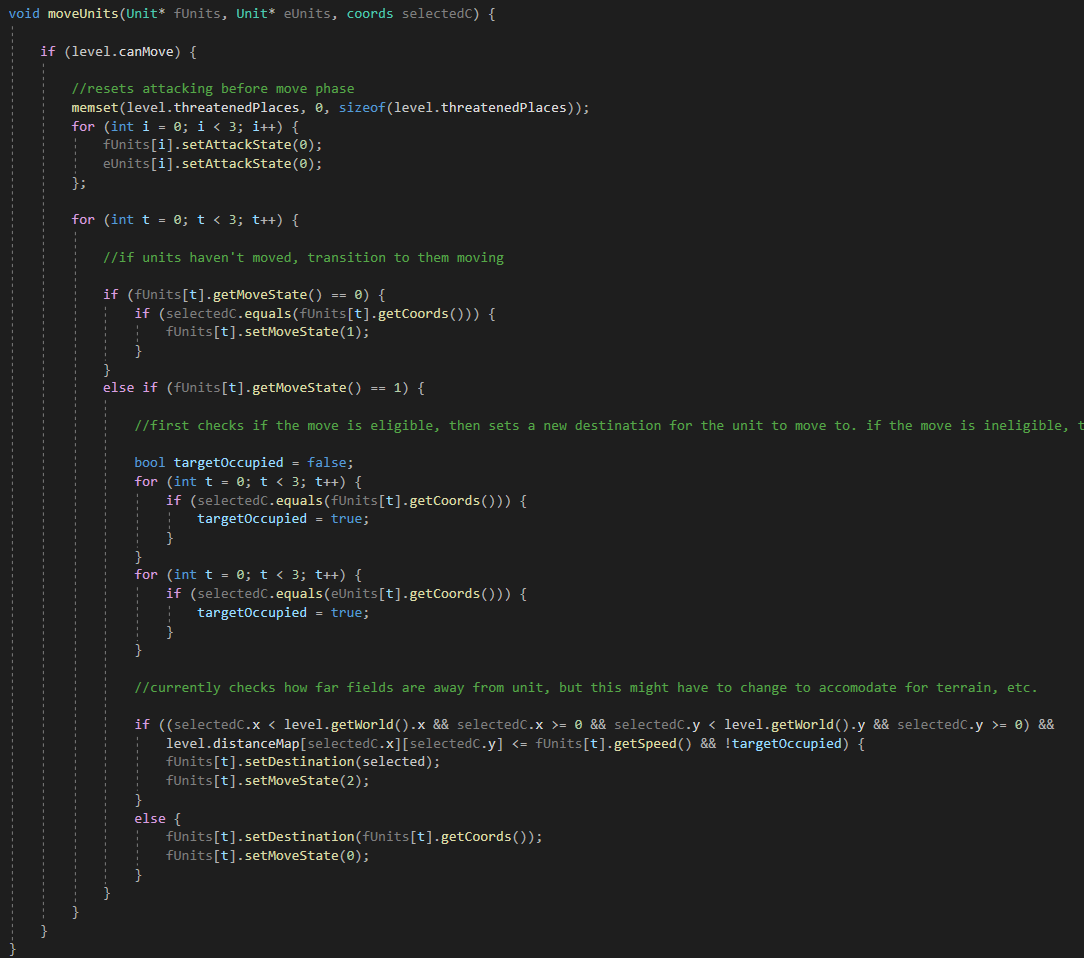
1Source for how multidimensional arrays would work in C/C++ <https://stackoverflow.com/questions/1946830/multidimensional-variable-size-array-in-c>  
I ended up putting this off for now, deciding to get back to this later.

Finished combat

Notes: added all game-related variables to gameState object

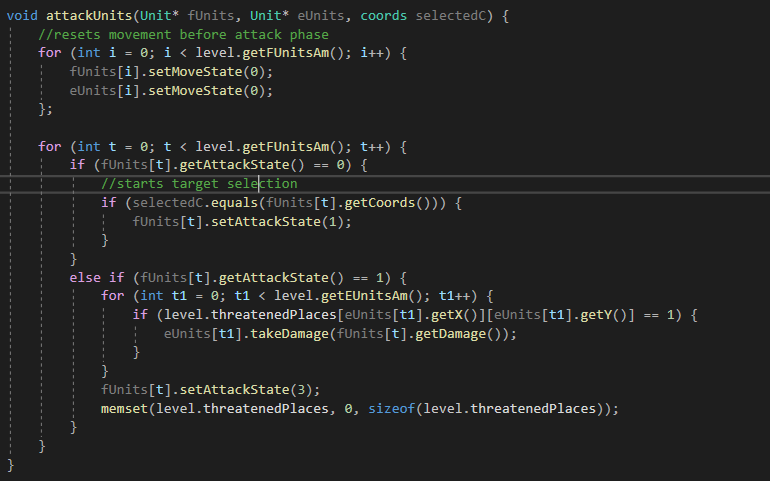
Note: planning on making all attributes private, as code is dirty rn

## development – 14/01/2021

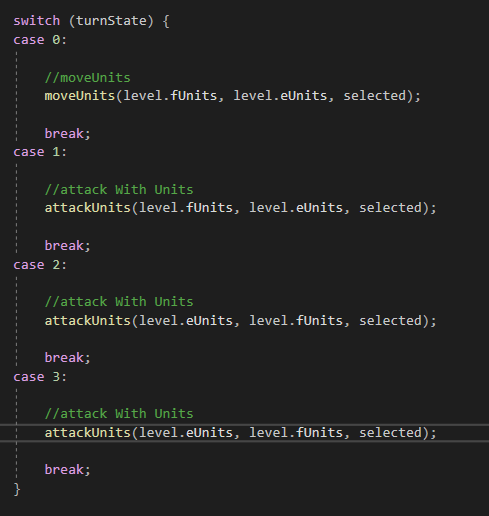


(moved movecode to single function so both teams can move, small changes throughout the code to accommodate for this change)

## development – 15/01/2021

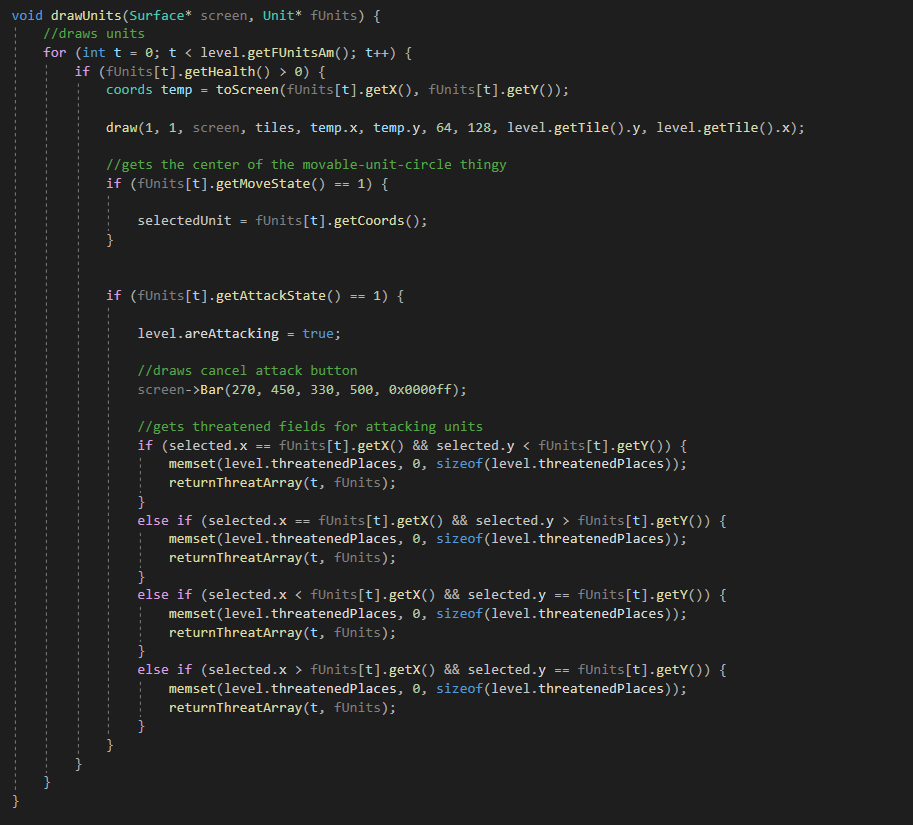


(moved attackcode to single function so both teams can move, small changes throughout the code to accommodate for this change)

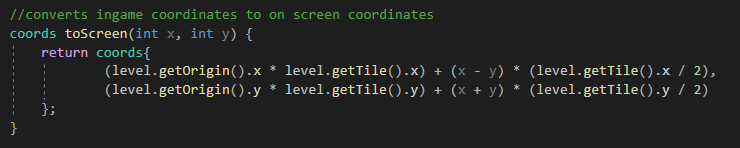


## Development- 01/02/2021

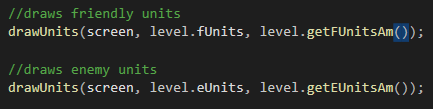
Changed drawUnit function to dynamic function that accounts for both sides of the field



Changed toScreen lambda function to proper function to accommodate for this change



Added unit parameters to make functions dynamic for higher unit counts





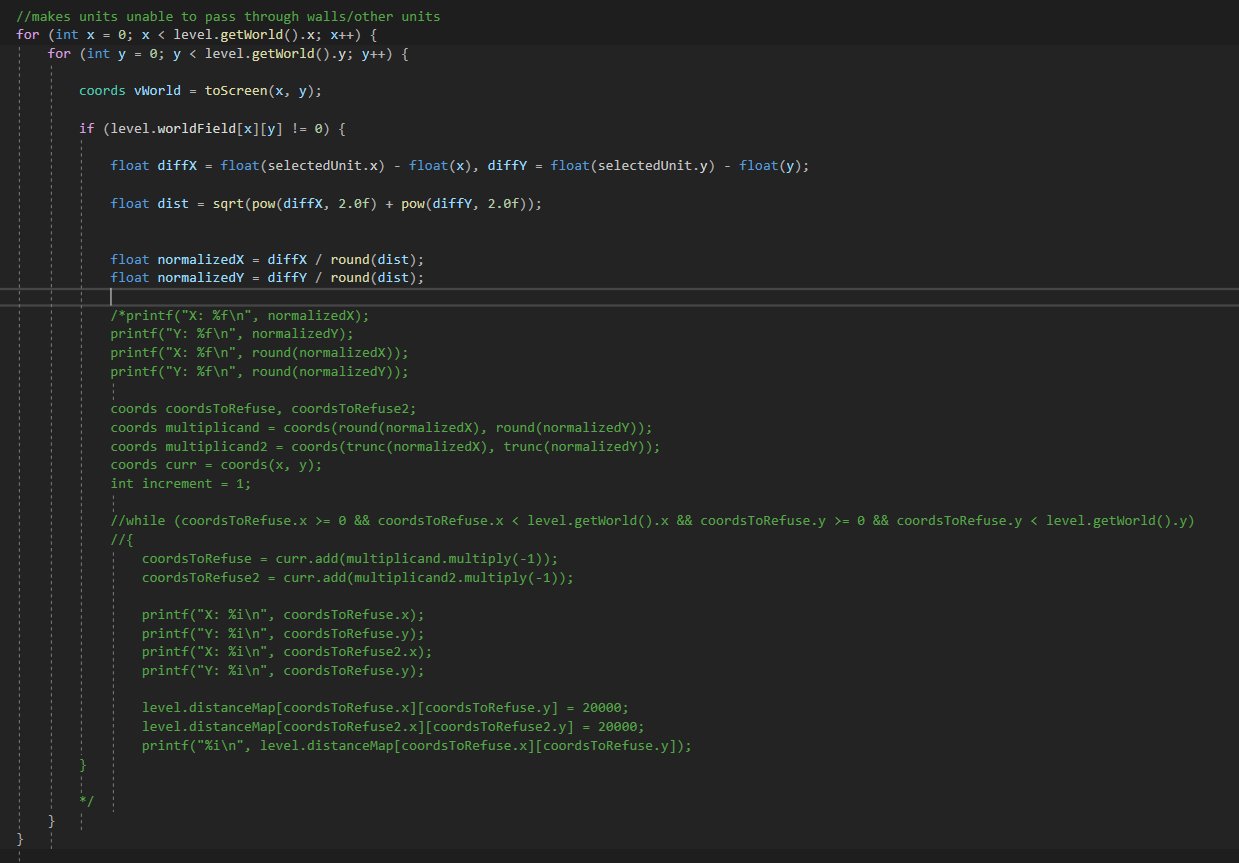


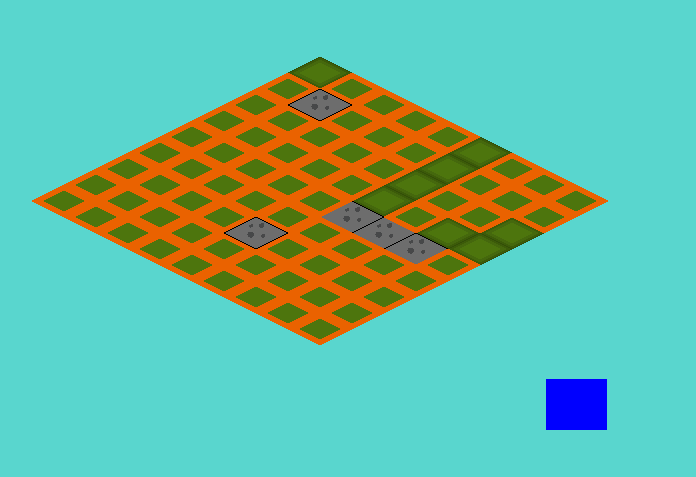
Source for vector normalization

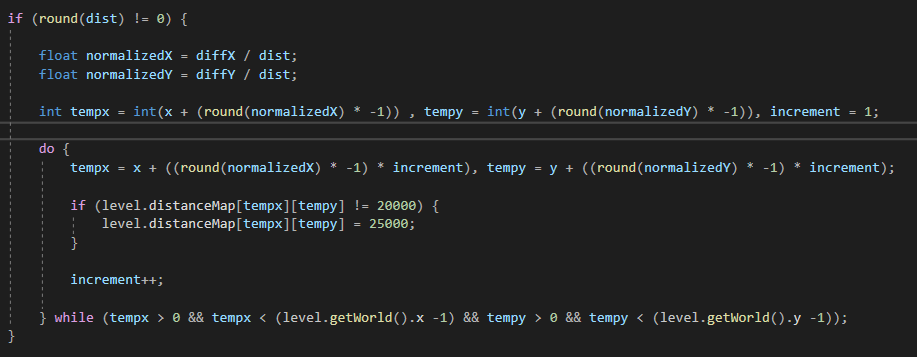
<https://www.khanacademy.org/computing/computer-programming/programming-natural-simulations/programming-vectors/a/vector-magnitude-normalization>

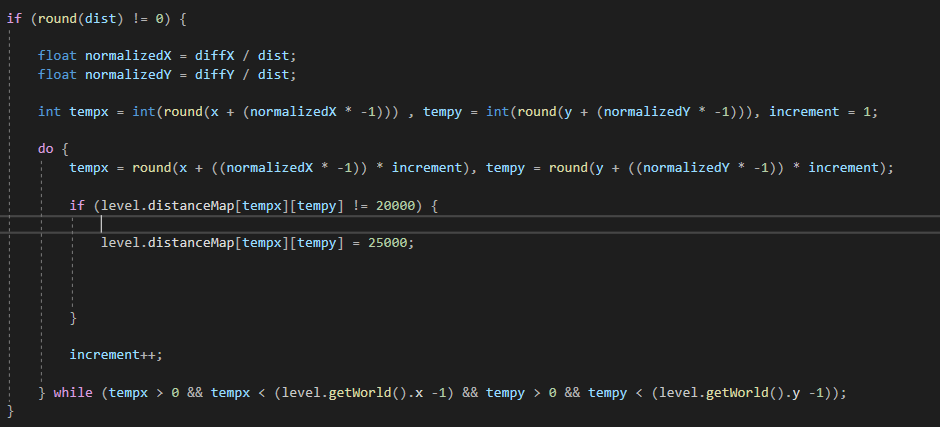
Source for float conversion from integer division

<https://stackoverflow.com/questions/16221776/why-dividing-two-integers-doesnt-get-a-float>



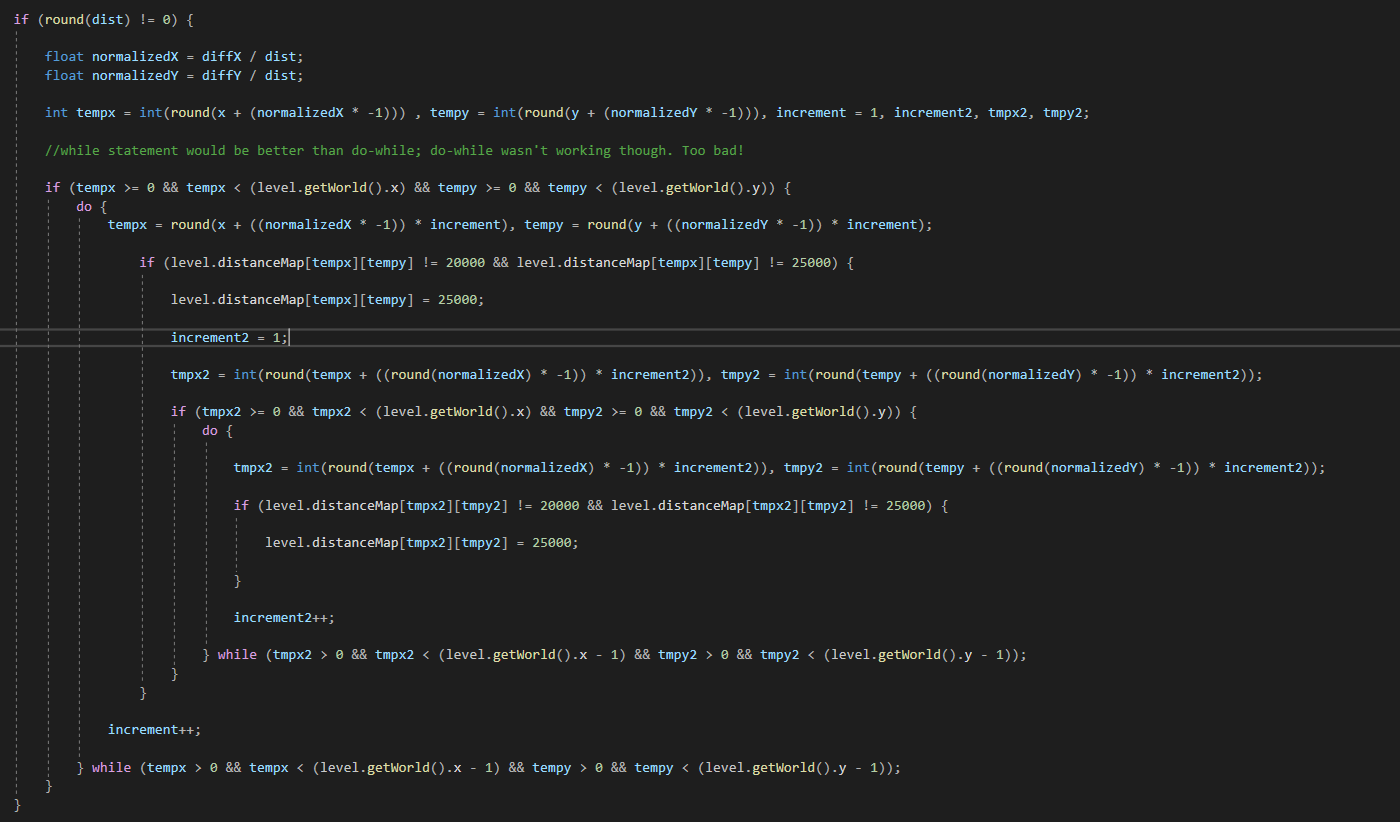






Changed placement of Round so the lines are more accurate

## Development- 06/02/2021



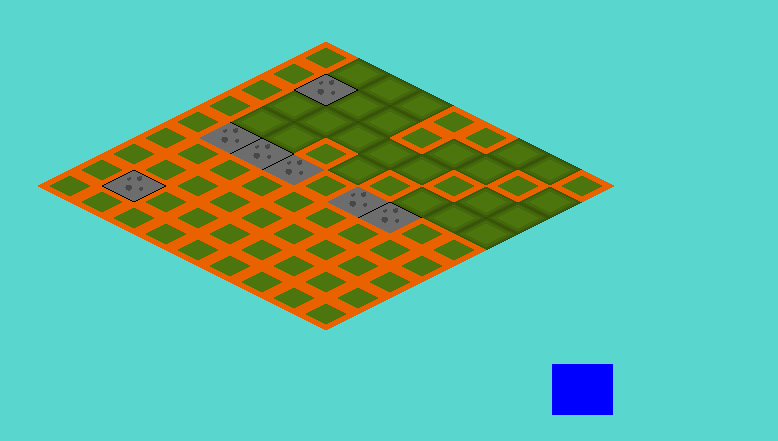
“finished” code for blocking out spaces behind walls

Calculated vector from unit to wall tile

Drew a line behind the wall tile where the unit couldn’t go

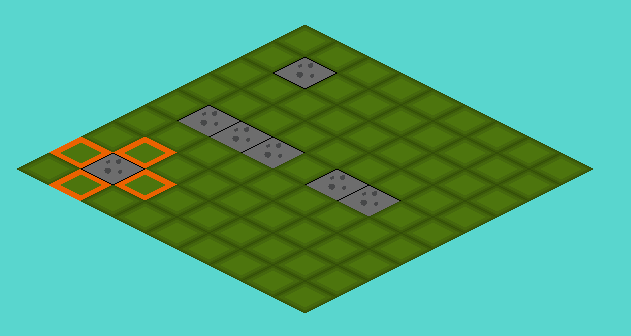
Filled in the other tiles not covered by the initial lines with a second loop

Yielded unsatisfactory results

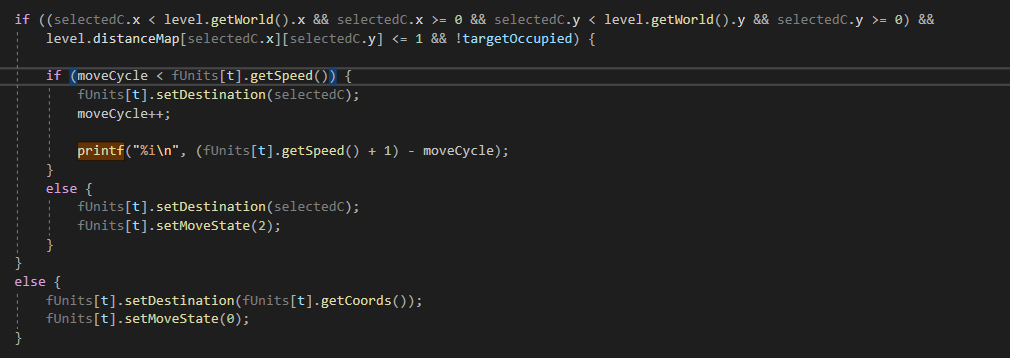


Results were choppy; new idea for moment jumped to mind

Move units by selecting a path, square by square -> more precise movement, no need for system like this



Some minor adjustments in the code later; units now move one space at a time, a maximum amount of times equal to their ‘movement speed’

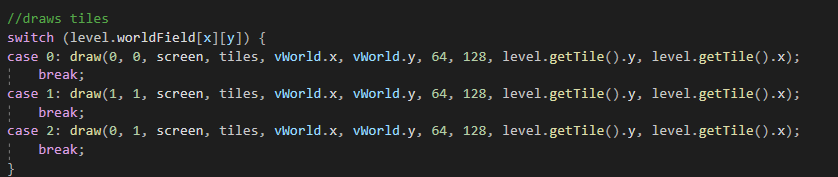


Part of the code responsible for this, ‘movecycle’ gets reset whenever attack phase starts

Reference for dynamically sized arrays in C++   
<https://www.geeksforgeeks.org/variable-length-arrays-in-c-and-c/>

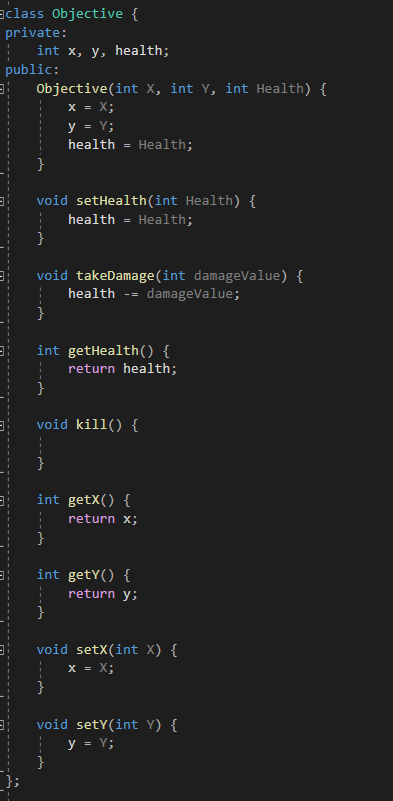
<http://www.cplusplus.com/forum/beginner/1601/>

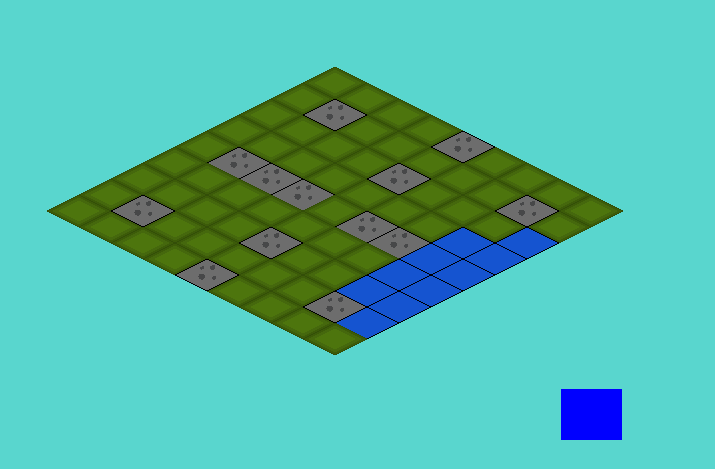
(to create an array that stores the unit’s previous moves, so it can premove without actually moving. This is just to increase PEBCAK errors 😉)



Added water tiles

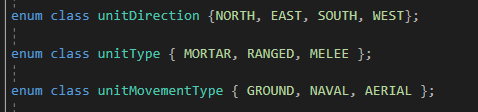
Added objective tiles





Fixed unit death by moving units to -1, -1 when they reach 0 or less hp

Added enumerators for unit types and unit direction

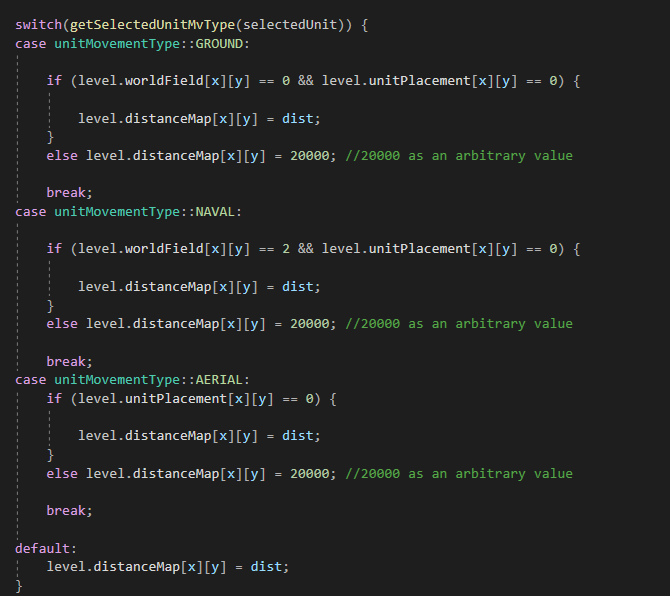


Divided units into 3 types of mobility; naval, ground and aerial

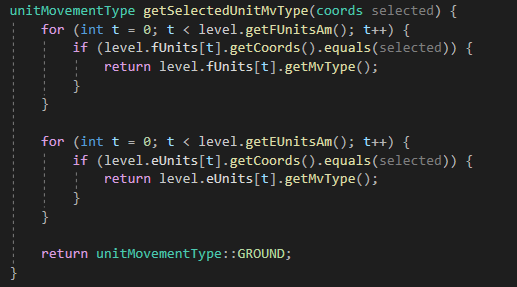
Naval: only water

Ground: only ground, blocked by mountains

Aerial: van move anywhere

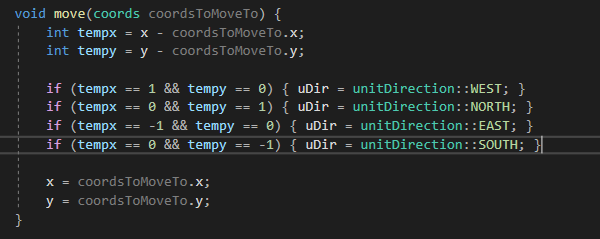


Modified unit interaction with terrain based on movement style



Gets movement style from selected unit

## Development- 08/02/2021



Gets direction of unit (for visual representation/ sprite)

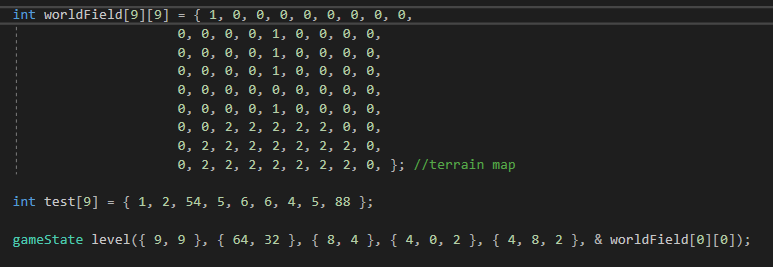
<https://stackoverflow.com/questions/9426932/how-do-i-pass-an-array-to-a-constructor>

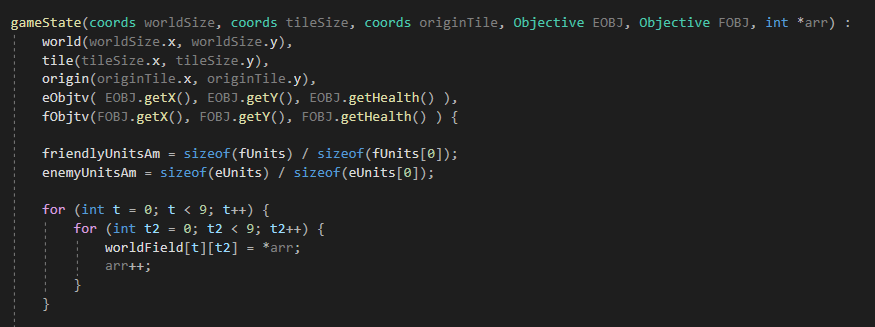
<https://stackoverflow.com/questions/9802341/constructor-and-initialization-of-custom-classes-objects>

<https://en.cppreference.com/w/cpp/language/constructor>

reference used for change in constructor of gamestate object

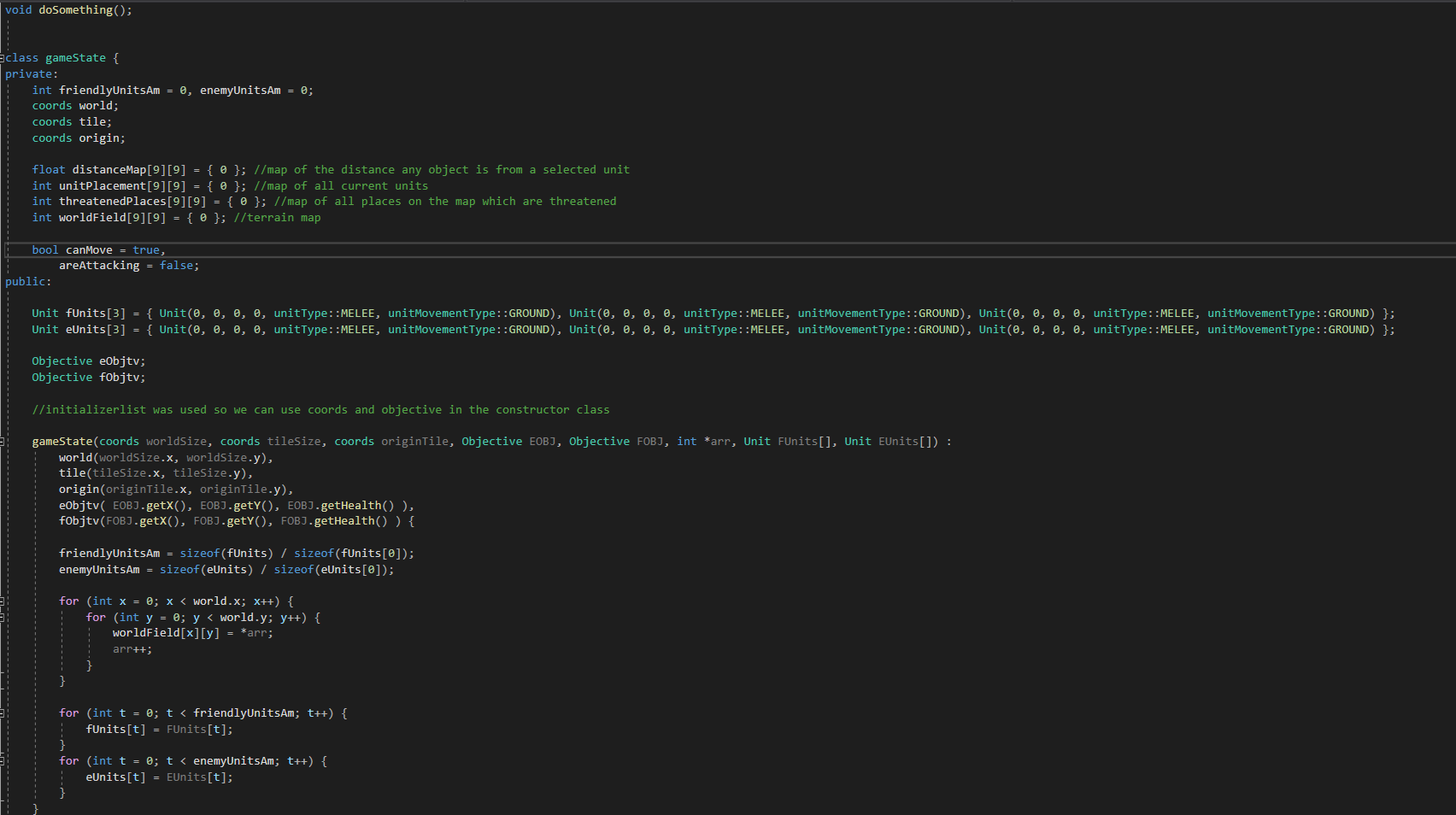
<https://www.youtube.com/watch?v=iChalAKXffs>





Finally got the constructor working, passing a 2D array using pointers

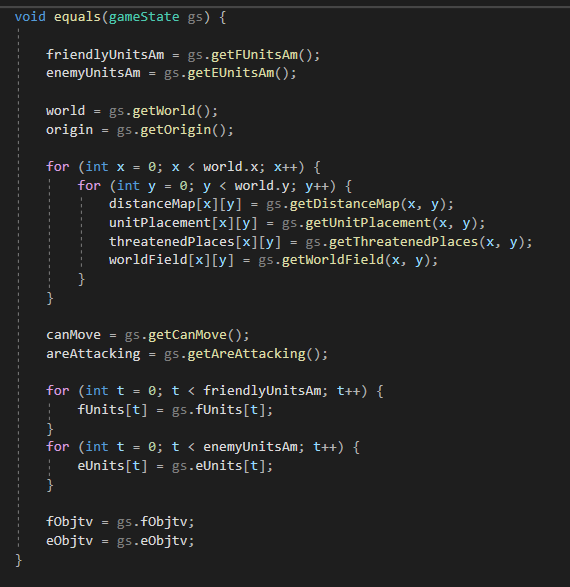
## Development- 09/02/2021



Updated gameState object

Finished constructor

Moved attributes to private, added get/set



Added ‘equals’ method which overwrites an instance of gamestate with another instance of gamestate, which would be used to restart and load different levels.

Moved tilesize to a global variable instead of having it inside level, as tilesize wasn’t dependent on level, instead being a ‘static’ 32 by 64 (not really static as it would change with 3D-ish tiles, )

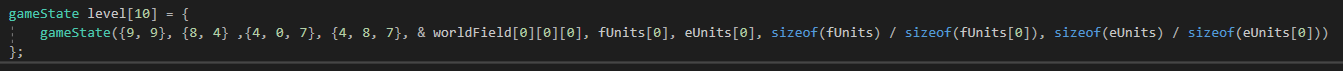
## Development- 12/02/2021

Changed gamestate constructor again;

now they take integers and form the coordinates manually instead of taking coordinates. The reason for this was fairly simple; in the old version of the constructor, I take the attributes of the objects I passed as parameters and assign them to new objects, which are attributes of the main gameState objects. Essentially what I was doing was as follows; making an object, breaking it down to its individual attributes, and then building it back up in the constructor. Of course, it makes no sense to do this; why build up the object only to then break it up again, only to build it up yet again? While this wasn’t my only reasoning, the other one being that I had been fiddling with getting this constructor to work for so long now that I was just completely done with it, so I opted to do it this way. I couldn’t do this for all attributes however; the map- and unit arrays were simply too large to do this cleanly. But for these few attributes, consisting on exclusively just integers, it was more than manageable.

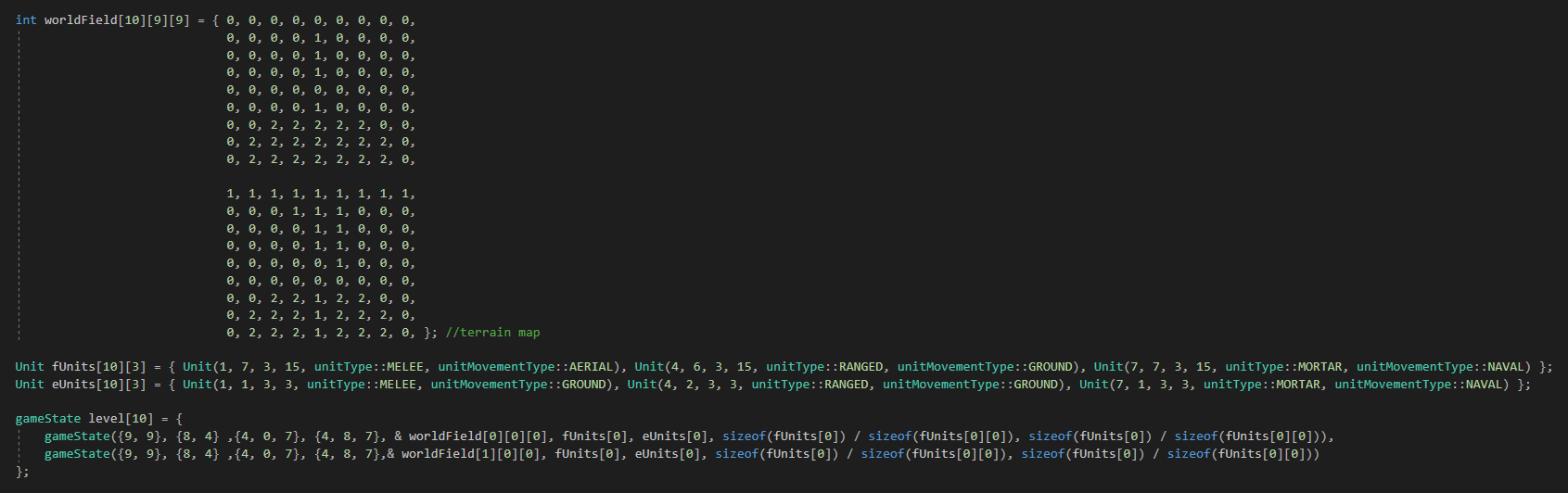
-ignore previous block of text; I ended up writing it with objects in the constructor just to what I was writing cleaner to me personally.

Created an array of gameStates to act as levels the player would go through



I created a three dimensional array worldField, which would store all the layout data of the levels. I wanted to find a cleaner way of storing this, such as in a datafile, but for now opted to just hardcode them.

Alongside this, I changed the fUnits and eUnits arrays, which I use to assign these arrays to the gamestate object, to 2D arrays to accommodate for the multiple levels.

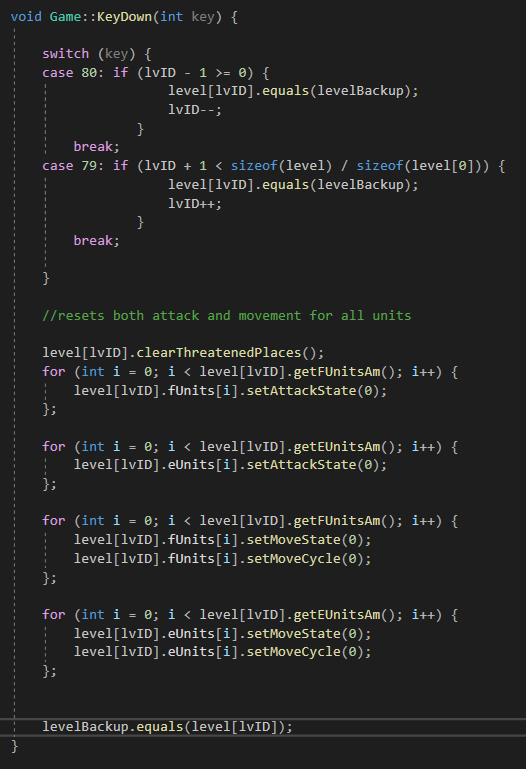


With this, given it would take some work, I could change the level the player is currently on by just changing a single value, ‘lvID’, a global variable which I would use as the index for my level array.



Finally figured out how to give this function a body in game.cpp! Low brain energy moment

After figuring out what keycodes the arrow keys were mapped to, I created this simple function which would allow me to switch in between levels with a single button press.

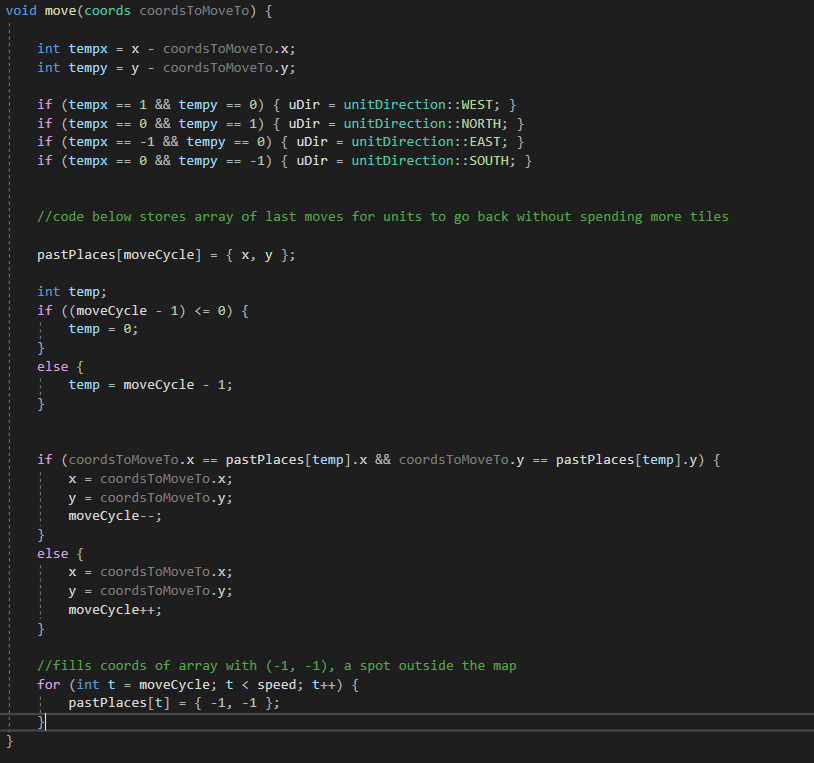


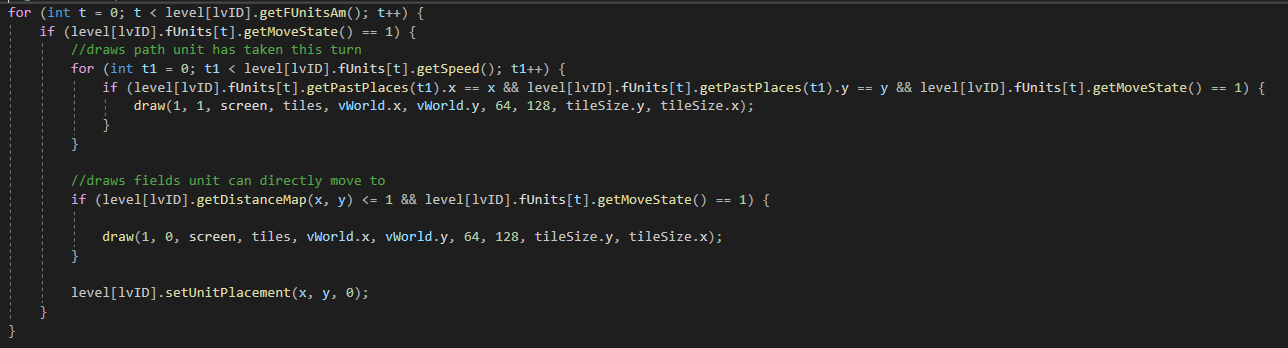
The game is basically functionally functioning now; the only things left on the to do list are the following:

-add a backtracking element to unit movement, so you can undo your own movement so to speak, just to add some user-friendliness  
-add the enemy AI  
-add all the spritework  
-add a UI and a menu

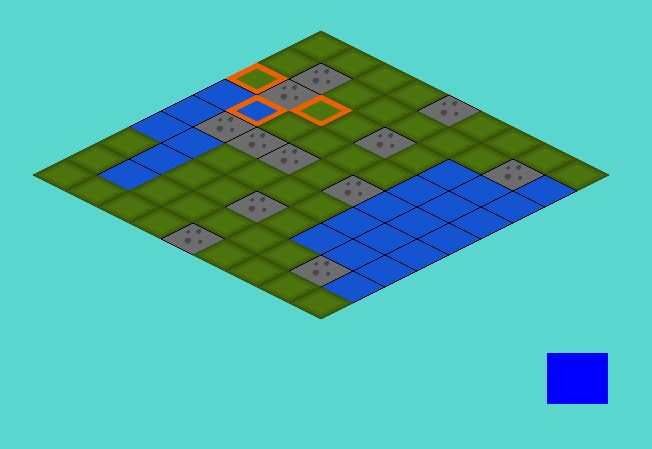
## Development- 13/02/2021

Added backtrack function to movement; storing an array of all previous moves a unit has taken and checking the last position in that array for every move the player makes, to check if they’re moving backwards or not. This is done to prevent users from accidentally wasting a tile of movement due to a mistake. The final move still locks the unit into place, which I will fix with a UI element.





Code for rendering the past tiles (for now I am still using one of the 4 sprites I currently have)



Blue squares behind the selected unit (as indicated by the orange tiles surrounding it) are the path the unit has taken. These tiles are only visible when moving the tile itself.

## Development- 14/02/2021

THE AI (finally)