

How are computer simulations developed, and how can we use them to mirror real life?

*A Study of the algorithms and methods used in the development of modern computer simulations*



James benjamin Brimelow gorman

The Arthur Terry School

**Contents**

1. **Introduction**
2. **Research**
   1. **Conway’s Game of Life**
   2. **Cary Huang’s Evolution Simulator**
3. **Development Diary**
   1. **World Building**
      1. **World Generation**
      2. **Name Generation**
   2. **Theory**
      1. **Storage & Run-Length Encoding**
      2. **Resources & Manpower**
      3. **Science**
      4. **Religion**
      5. **Rulers & Politics**
      6. **War & Harmonic Series**
   3. **Evaluation**
4. **Sources & Project Download**

**How are computer simulations developed, and how can we use them to mirror real life?**

*A Study of the algorithms and methods used in the development of modern computer simulations*

Introduction

Arguably the most valuable invention in the modern age is the personal computer. Its intense processing power and ability to follow instructions at nigh-instantaneous speeds makes it useful tool in workplace and home environments. Despite these incredible powers, current computers are merely an object, a device used by humanity to achieve their goals, but what if we were to change that? What if we were to give the computer the ability to think? the ability to solve more abstract problems once thought reserved for human minds.

Obviously with modern technology the idea of creating a thinking computer is preposterous. However, humanity has discovered a way around this problem, using complex algorithms to produce results that *mimic* those that would be given by a human mind; In other words, a simulation.

In this document I shall be detailing my endeavour to create my own simulation, one which simulates a history of a randomly generated world, with its own religions, countries, cities and culture.

Research

Before the project can even begin, intense research on existing simulations must be done to decide how the program will function and what methods need to be developed in order to meet the finish product, to do this I consulted two main sources; John Conway’s Game of life, and Cary Huang’s Evolution simulator.

John Conway’s Game of life is an interesting idea. Officially classified as a ‘zero-player’ game it presents a simple simulation of ‘life’ that abide by specific rules to survive. The presentation of Conway’s game of life is a grid, in which each square is able to be given two values; Dead or Alive. After the starting grid has been input, the program will then follow the following rules [1];

* Living cells with less than two living neighbours become dead cells each generation
* Living cells with two or three neighbours survive in the same grid space they originally inhabit
* Living cells with four or more neighbours becomes a dead cell
* Dead cells with three living neighbours become living cells

While the rules seem simple enough, the actual result of these rules is incredibly interesting, as structures can (and almost always will) form by correct patterns of cells being formed, essentially creating randomly generated machinery, some of the more interesting ones have been listed below [2]:

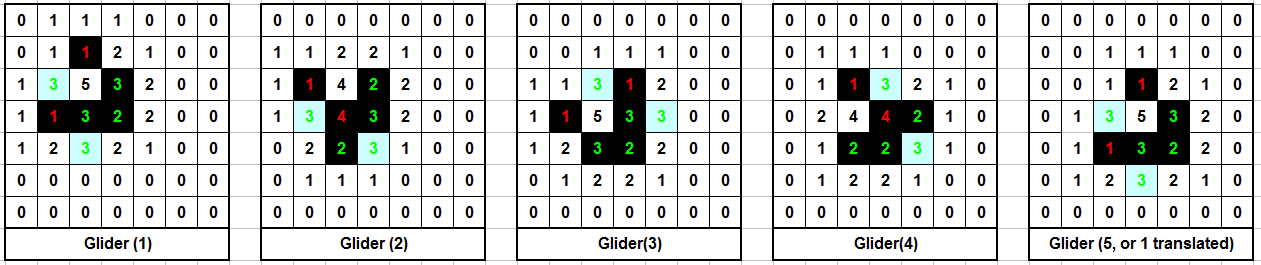
*Block* –

This structure is unable to die unless third parties interact with it, the fact that all of its cells have two living neighbours means that the structure can neither die nor reproduce, and if it isn’t interacted with, it will stay this way eternally.

*Blinker* –

Like its distant brother, the block, without outside interaction a blinker is able to survive forever. However, unlike the block, the blinker has two main forms which it cycles between each generation, forming a ‘blinking’ line which rapidly switches between lying horizontally and vertically. The reason this occurs is because of the fact that while the centre cell will always have two neighbours, the outside cells only have one neighbour, causing them to die each generation, additionally, the cells north/south or east/west of the centre block will always have three neighbours, prompting them to becoming living and repeat the cycle.

*Glider* –

Jumping incredibly far ahead in terms of simplicity, we next have the glider, named as such for its ability to move diagonally eternally, this occurs because the shape is able to replicate itself over 5 generations with one lower x and y value than previously, essentially meaning the shape has transformed by one x and one y.

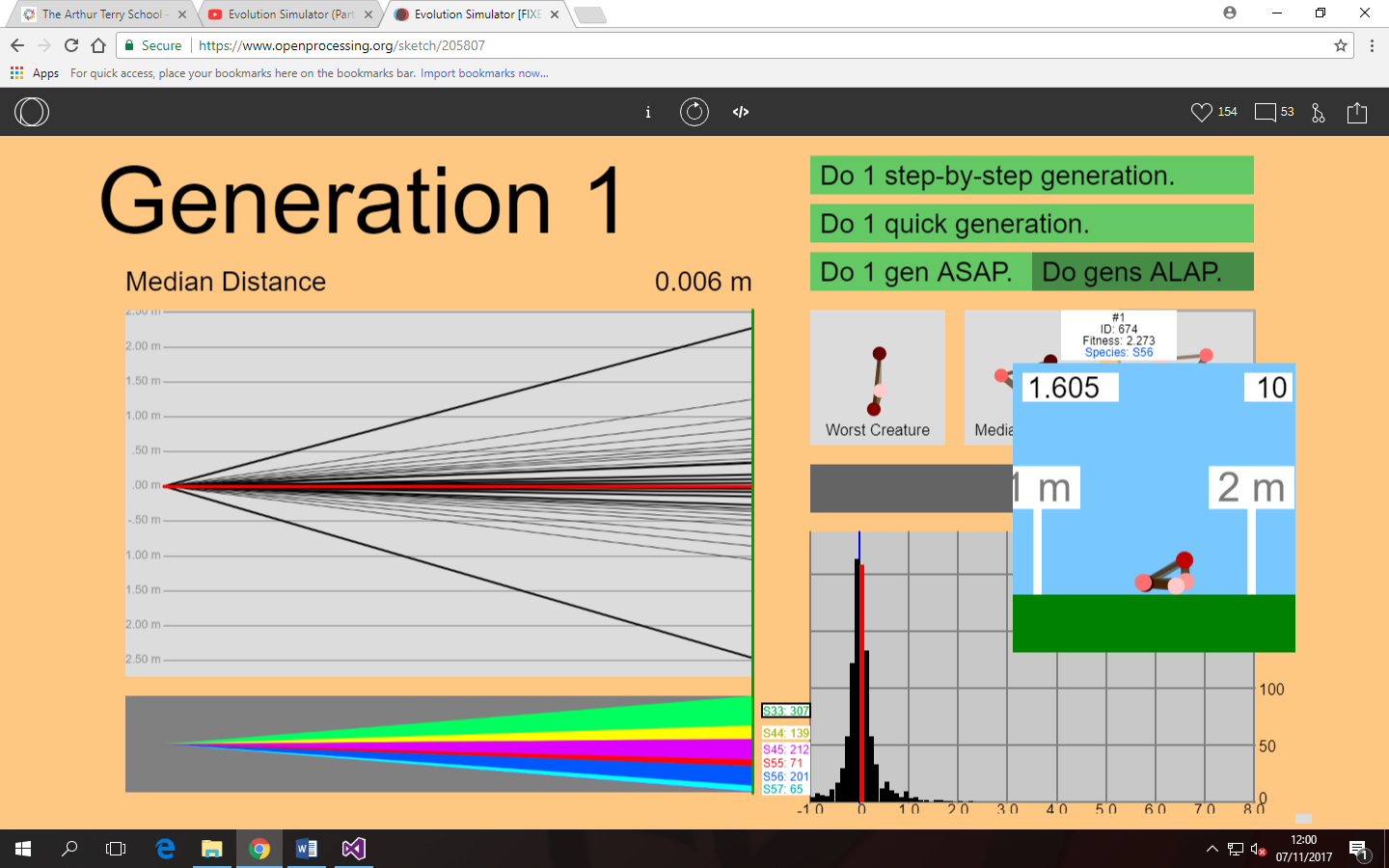
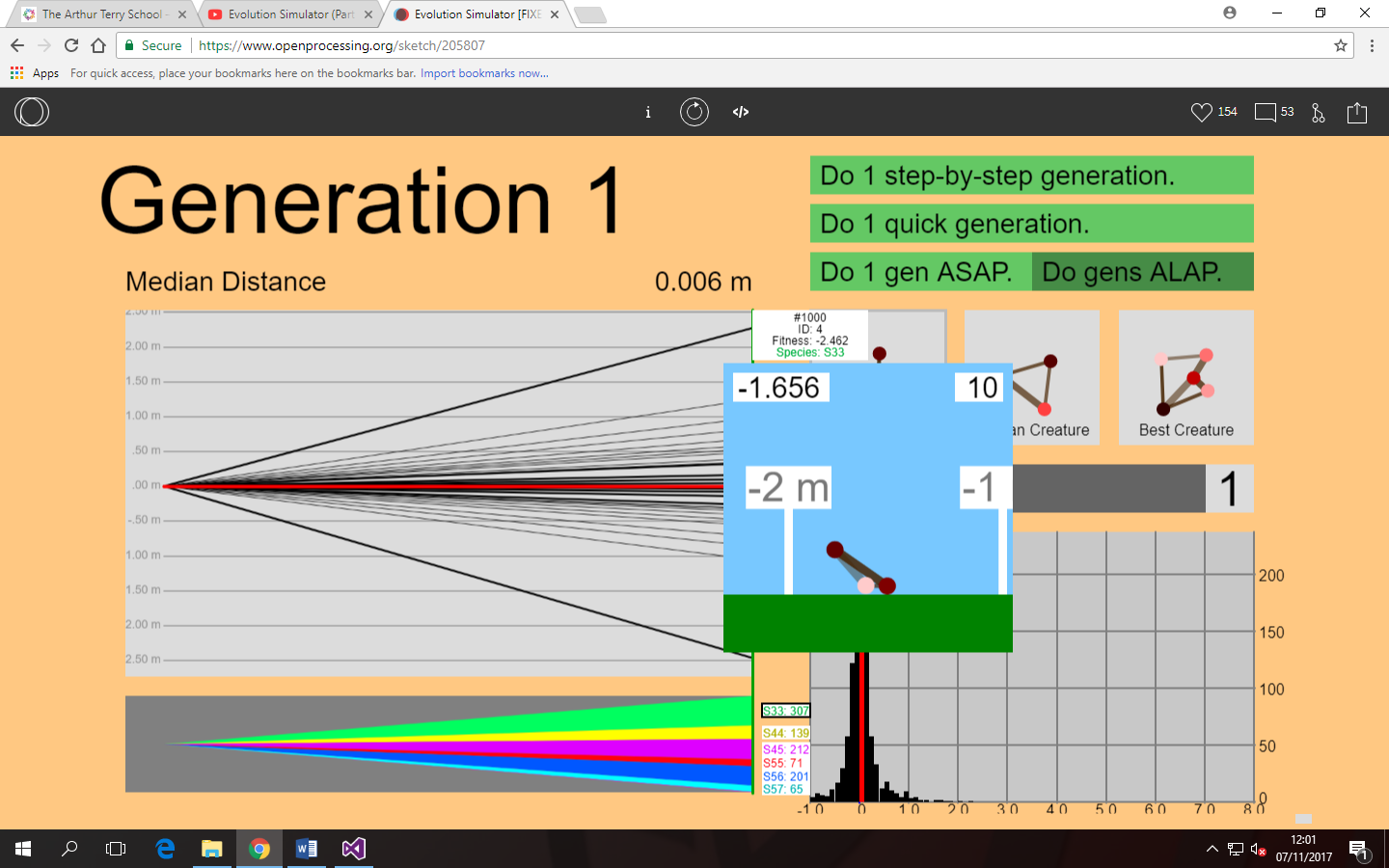
Conway’s game of life is interesting not because of its ability to realistically simulate a mind, but rather as a demonstration of how simple rules can forge a universe of possibilities, as shown by the randomly generated structures which abuse the rules of the game to their advantage. This idea interests me greatly, and a similar concept is incorporated into my next article of research;

*Cary Huangs Evolution Simulator*

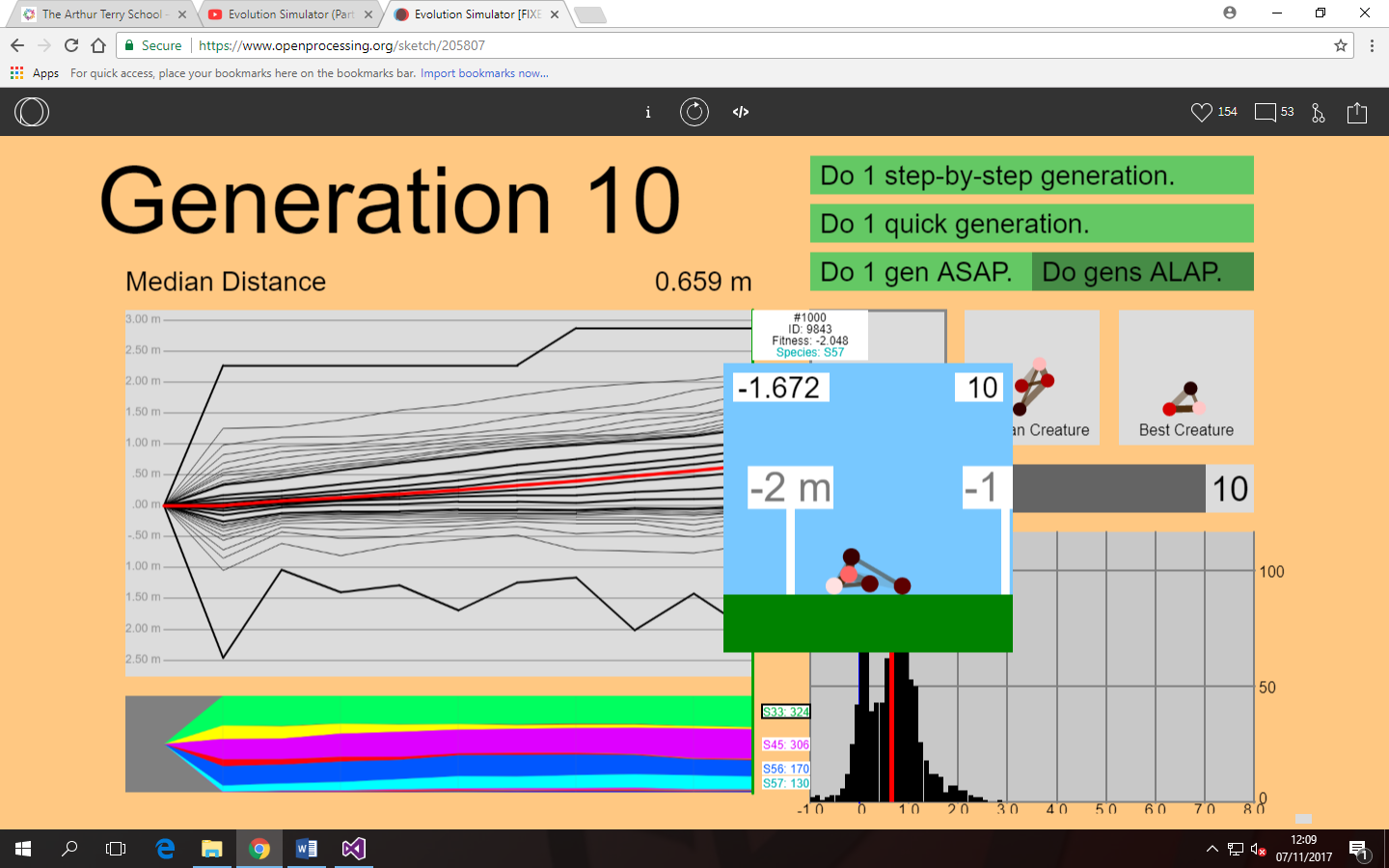
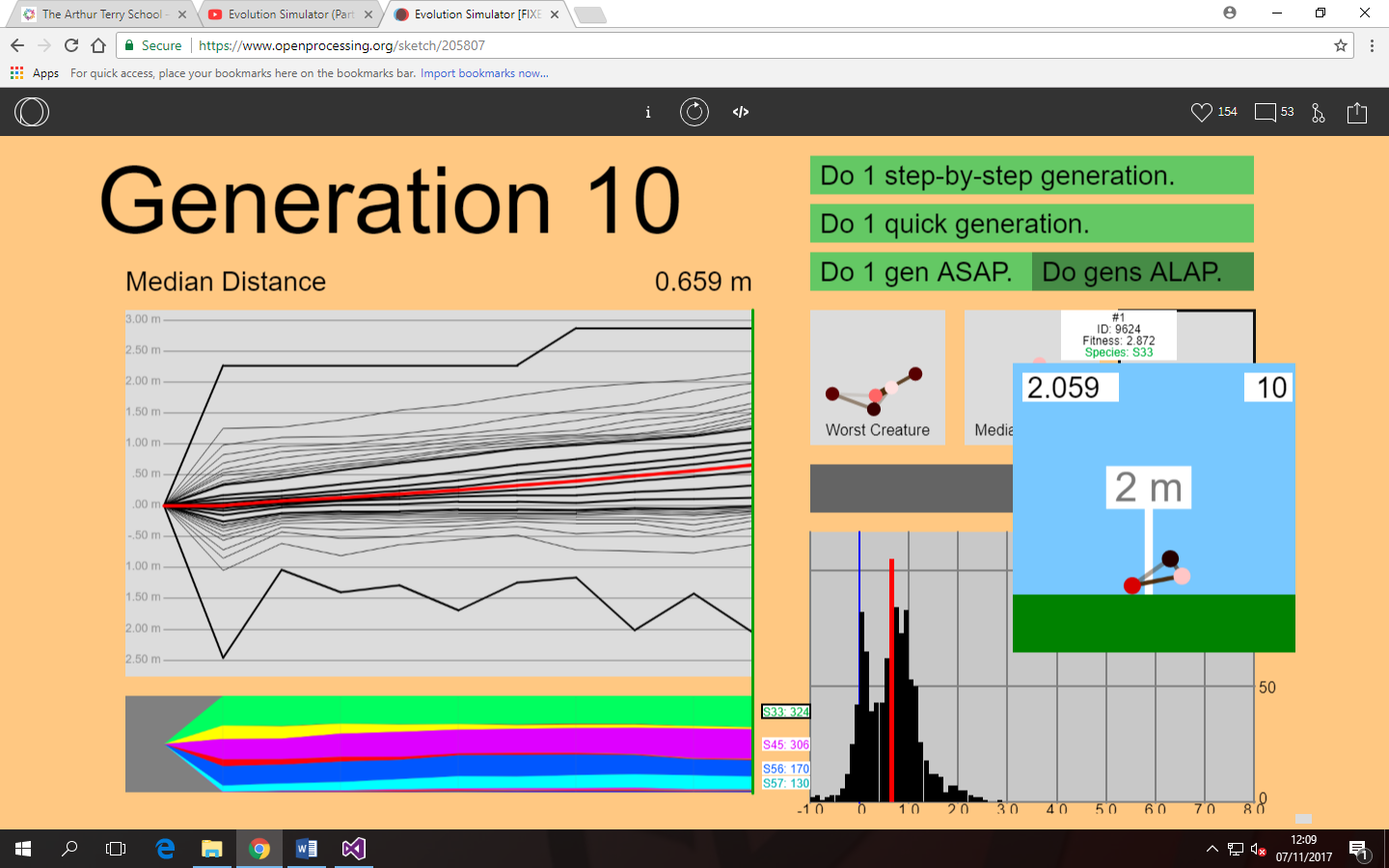
Allowing a computer to create its own structures based on specific rules is certainly an intriguing idea, but it lacks use. Conway’s ‘life’ serves no purpose other than to survive, meaning it is not a particularly good simulation of real concepts. So what if we were to use concepts discussed in Conway’s model for a more practical model, say a model of evolution?

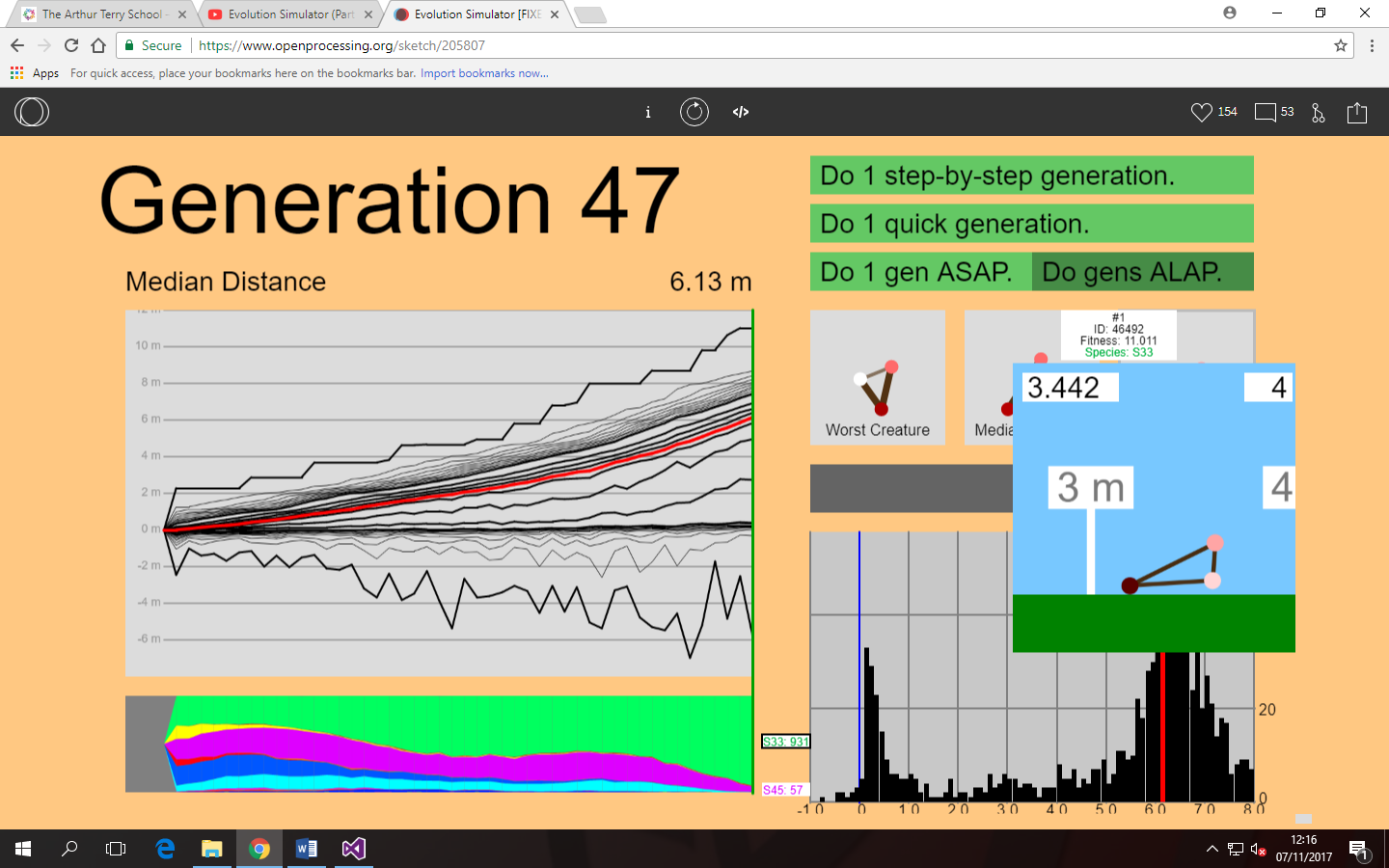
This is where Cary Huang’s Evolution Simulator[4] comes in, in essence it attempts to simulate a population of shapes as they evolve to reach a specific purpose; the ability to travel as far right as possible within fifteen seconds. While Cary goes over his simulator in a lot of detail in his video series on this project[3] the basics are as such:

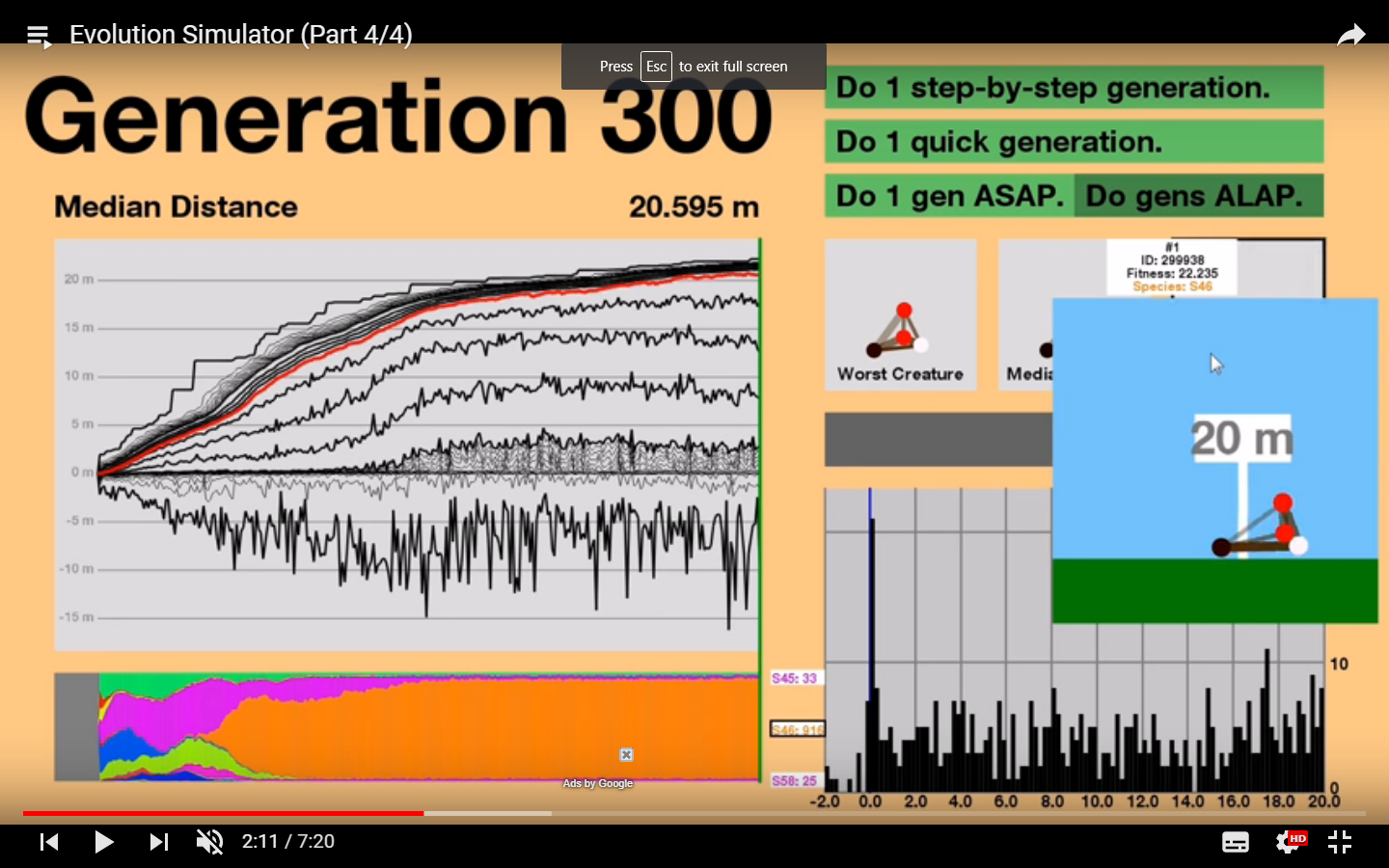
Each ‘creature’ consists of nodes and muscles, the nodes are represented by circles that use the colour red to represent friction (White circles have low friction, red circles have medium friction, whereas black circles have high friction). Muscles are used to bridge gaps between these nodes and have specific contraction and expansion lengths, which it will alternate between slowly over a period of time specified by the strength of a muscle (which is represented by the transparency of a muscle, low strength muscles will be transparent and will have a lower expansion/contraction length due to it being unable to reach a greater length within its allotted expansion/contraction time. On the flip side, a muscle with a high strength will be opaque, and will be able to reach further distances due to its ability to expand/contract more in its allotted time)

From these rules, one thousand unique ‘creatures’ will be created, each with different positions for nodes, different node frictions etc. These creatures will then be timed, and the top 500 shapes that moved the most to the right survive to the next generation, duplicating themselves (with a chance for mutations) to replace the 500 shapes that were unable to place within the top 500 for movement.

In the images to the right, you can see how within ten seconds, the worst creature was able to move 1.6m to the left, whereas the best creature was able to move 1.6m right, allowing it to reproduce into other creatures, which will eventually overtake it due to mutations, forming a shape which is even faster than itself. The worst creature will be removed from the gene pool, meaning its traits which hinder its abilities to move right will not be passed down to the next generation.

Jumping forward ten generations, the effect of these changes to the gene pool can be seen clearly, with the average distance achieved by the creatures increasing by almost 0.6m. Additionally, we can see that the best creature in generation 1 has passed on its traits to the best creature in generation 10, with it having a similar shape but a mutation in the form of the removal of a node present within the generation 1 version that only served to slow the creature down. It is also important to note that in this early generation, the worst creature still moves left, an issue that will no doubt be removed in following generations.

Almost 30 generations later, while the issue of creatures moving left has not been fully removed, the average creature is able to move 6m within 15 seconds, a large increase from its original, which had a best of 2.2m. The best creature in this generation is able to move almost 11m within that time, signifying a clear progression of the species since its early generations. It should also be noted that the best creature has inherited the triangle shape from its ancestor, which is likely the cause of its ability to move at such fast speeds. The triangle shape is able to move so quickly due to the way its nodes work in conjunction, as the top right node will drag the bottom left node off the ground, forcing the bottom muscle to pull the bottom right node towards the bottom left node, effectively demonstrating movement.

This basic structure can be seen within the programmer’s demonstration of his own program, in which after 300 generations, a modified triangle formed which was able to move 22 meters. Notable advantages this creature had over the one I generated includes the low friction in the bottom right node, which allows it to push itself forwards easily, it also includes a counterweight within the centre of the triangle, which pulls the bottom left node quicker than the top right node, essentially decreasing the delay between the bottom right node being risen from the ground (to move right) and it reaching the ground again. This allows this cycle to repeat more times per second, and as this mechanism is used to move the creature, the more of this mechanism that can be performed per second, the faster the creature can move.

While these two simulations have little relevance to the actual theme or concept of my project, they represent a fundamental theory behind simulations. While my history program will not incorporate things like the mutation of limbs, the idea of artificial intelligence attempting to use the means at its disposal to overcome the rules set out by the program and come out on top is a central idea present within my simulations; after all, the main focus is to see how history would go in an alternate world where almost everything is new, and more specifically which countries would come out on top.

Development Diary & Further Research

*World Generation*

When attempting to tackle the creation of an alternate world, the obvious place to start is in the world itself. World generation took arguably the most time out of any part of the programs development, as I decided such a present part of the program had to be as close to perfect as possible, after all, it is the part of the program that will be viewed the most, and therefore needs to look somewhat realistic. To achieve a realistic world map, I attempted multiple methods of generation, including:

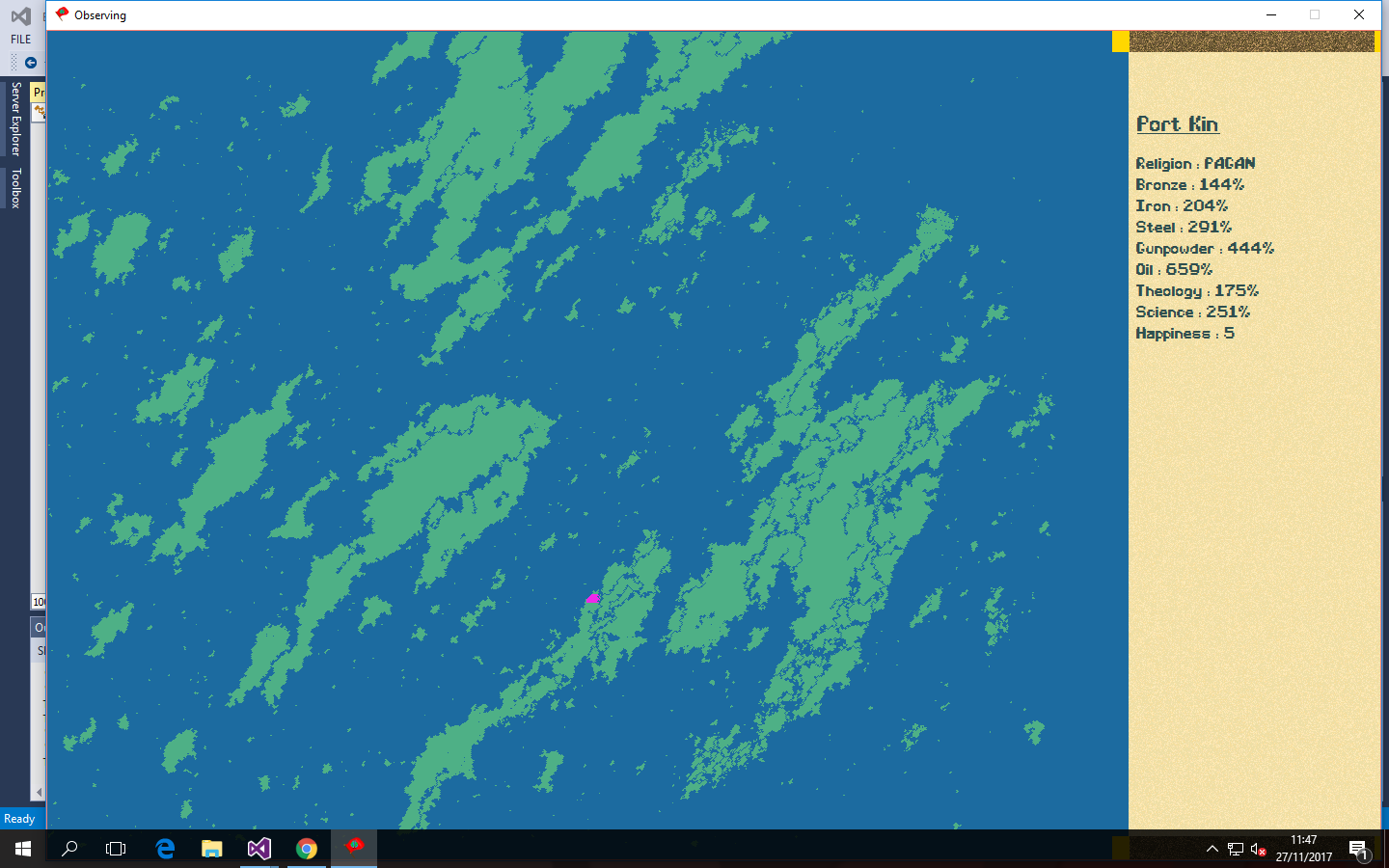
*Snake Method*

The first method I used was a recycled algorithm from an old program I worked on, which I named the ‘snake method’. The concept of the snake method involved a single dark green square being selected on a grid, and choosing an adjacent square to move to each ‘turn’ randomly, this developed a line of dark green pixels, which I would then use to place three light green pixels in each adjacent direction, forming an island that had a seemingly random shape, fit with small bumps and indents that could be found on real islands. While this algorithm was no doubt functional and realistic, its problem came in the form of its implementation. The program I had originally developed it for used a 250x250 grid, whereas the program I was currently working on was 1000x1000. This meant that even when the algorithm was given large amounts of ‘turns’ it would only generate small islands, meaning the (already fairly slow) algorithm would have to be repeated many times over to achieve a realistic world map. After a week of attempting to implement this algorithm successfully, I eventually moved onto a new, more suited algorithm.

*Flood Method*

If you were to take all the water out of the ocean and look at the world, it would look like a barren desert of mountains and hills, this new world would obviously look somewhat similar to the real world, after all topographical maps would still show the distinct shapes of islands and continents among all the sand. If you were to fill in this theoretical world with water, eventually you would reach a point where it once again looked identical to our world, and this was the concept I used when developing the flood method. The flood method would generate a large topographical map, choosing single pixels to represent mountain tops and then slowly filling in its adjacent tiles with slightly lower down pixels. This, in theory, would eventually generate a large map with distinctive island shapes, which could be slowly filled with water (by replacing each pixel with water tiles in order of lowest depth until the water to earth ratio resembled our world) to create a realistic map. This algorithm failed spectacularly, making square shapes and unrealistic cliff faces that split continents in two. While this algorithm had to be completely scrapped, the semi-realistic method of island creation influenced my next (and final) idea greatly.

*Pangaea method*

This algorithm began with a large circle spanning approximately 20% of the maps surface. From there, the circumference would be replaced with red pixels, which would move randomly to adjacent tiles, replacing its former tile with ocean. If these red pixels couldn’t move, they would randomly place themselves in a new coordinate (which could be ocean or land, if it was ocean the red pixel would simply disappear) and attempt to pursue their purpose once again. From there, the program would wait until all the pixels had died out or split up the circle enough to form continents, these continents would then be sorted by connection, and randomly placed around the world, forming large continents and small islands. Out of all the methods I formulated for the task of generating a map, this was by far my favourite, after all, it was decently quick, looked somewhat realistic, and was based off the real concept of continental drift. Because of the advantages this method presented, I decided to use it as my final world generation method.

An example of a map generated using this method, including a selected province.

*Naming*

After the world was finally done, it needed to be populated by humans. These humans would go about the same process as our real ancestors, forming languages, religions and cultures unique to their environment, but this posed a problem; How would I achieve this? Creating new words for these places, people and ideas isn’t something you can do completely randomly, if these new words were simply generated from a group of letters, the output would be strange incomprehensible phrases. For my program, I wanted to keep the idea of random names, but have them based on real naming conventions, essentially creating words that sound like their real world counterparts.

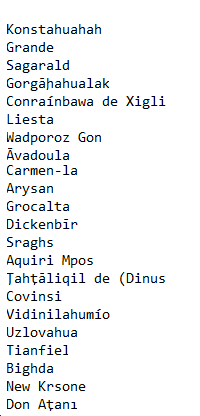
When researching this concept, I came across the mathematical function known as ‘Markov’s chain’[5], which from my understanding seemed to find the occurrence of letters following another letter within a database, and randomly generate letters based on the chance it would follow its previous letter (e.g. the letter e might follow the letter a more often than the letter u does, meaning the letter e would have a higher chance to appear following the letter a). While this concept was incredible in its realism, actual implementations of Markov’s chain into c# took multiple files and generally were too complicated for me to use successfully. Because of this, I decided upon a new algorithm based off Markov’s existing algorithm. The idea of this algorithm was that it would randomly select a word in a database, and choose a few letters from this word, it would then find the next time the last letter it took appears in the exact position of its original, I would then take some letters from this word, and string them onto the end of the previously pulled letters. This cycle would repeat until eventually it would either lose the ability to find new letters (due to excessive length) or a random generator would end the process prematurely, outputting the incomplete word.

Figure 1

An example of some names generated

Theory

Now that the basics were complete, I was able to begin the development of the actual simulation. This posed problem after problem, however, mostly concerning the actions of the artificial intelligence; My intention was to create artificial intelligence that would act similarly to a real human, however since the AI can only act in response to pre-written questions, I needed to develop the algorithms used to impose these questions to formulate a more realistic response.

*Storage & Run-Length encoding*

For the user to be able to use the program, it was essential that a mechanism of storage be developed, as it was unlikely the average user would be interested in watching the simulation run through its entire cycle in a single sitting. Because of the importance of this factor, it was equally important that the method used to store the files was as efficient as possible. I first attempted to make the storage efficient by storing files with a specific syntax the program could interpret; This was as follows:

|  |  |  |
| --- | --- | --- |
| Name | Function | Symbol |
| Start | To notify the program of the beginning of a new line, and thus a new record of data. | $ |
| Next Field | To tell the program that the current data field has reached its end, and the following data is for a different field | % |
| New Entry in the same field | This function allows for multiple sets of data to be stored concerning a particular topic, usually in a separate array. | , |
| End Line | This function tells the program to move to the next line, as the current dataset has been read. | ~ |

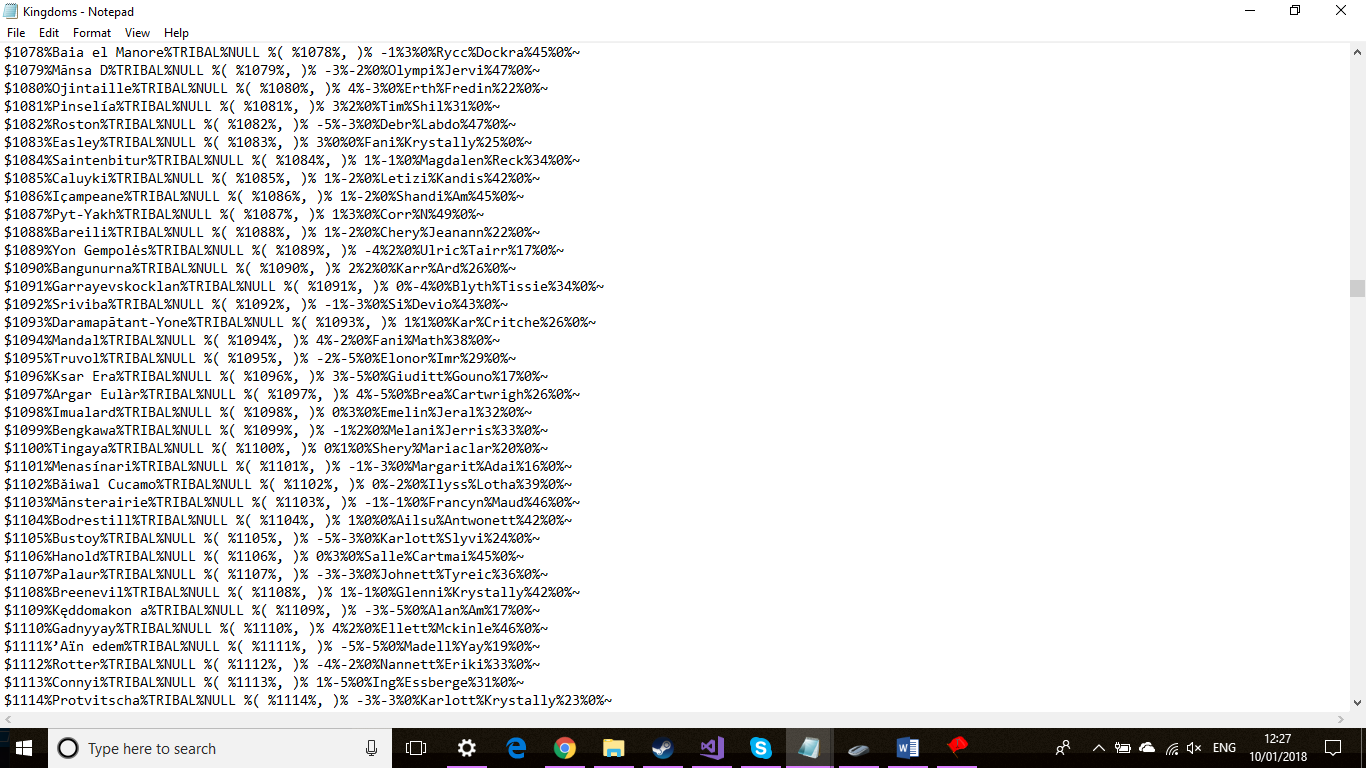
This method of storage, while hard to read, meant that a large amount of important data could be stored for very little cost in terms of storage. In the example to the right, the text would be interpreted as the Unique ID of the kingdom, the name of the kingdom, its type, its official religion, its owned lands and various other minor data.

Figure 2 An example of data used in the storing of a file

This, while somewhat efficient, was not suited to storing excessively large amounts of data, such as the storage of the world map. For this, I employed a different syntax.

|  |  |  |
| --- | --- | --- |
| Name | Function | Symbol |
| Ocean count start | Signifies the beginning of a blank space (or rather, ocean), any numbers that come after it will be interpreted as the amount of ocean tiles to add. | w |
| Ocean count end | Ends the ocean count and stores any number inbetween it and the start as the amount of ocean tiles to place. | m |
| Tile start/end | Numbers inside these symbols will be interpreted as a tile, and a singular land tile will be placed of the id inbetween them. | % |
| Skip | If placed within a w m, this will skip to the next line, essentially signifying that there is no more land tiles to be placed on this row. | f |
| End Line | This function tells the program to move to the next line, as the current dataset has been read. | ~ |

The conversion of my previous method (which involved placing a \* symbol per ocean tile) to a run-length encoding based method cut down the file size of the world file by almost tenfold, as it condensed the ocean tiles (which, like real life, made up most of the world) into much smaller sets of characters. This method of compression is referred to as “run-length encoding”, in which all sets of data with similar values are encoded under a smaller set of characters, which notes their value and count until unique data is reached.

*Resources & Manpower*

On generation, Provinces will be given a random amount of bronze, iron, steel, gunpowder and oil which will be vital to the generation of each nations militia. These values will be stored as percentages, with the greatest resource available with the nations being used to calculate the amount of manpower generated by a country each month.

Where B = 50 if the country is a chieftain + capital., 100 if the country is a kingdom, 200 if the country is an empire.

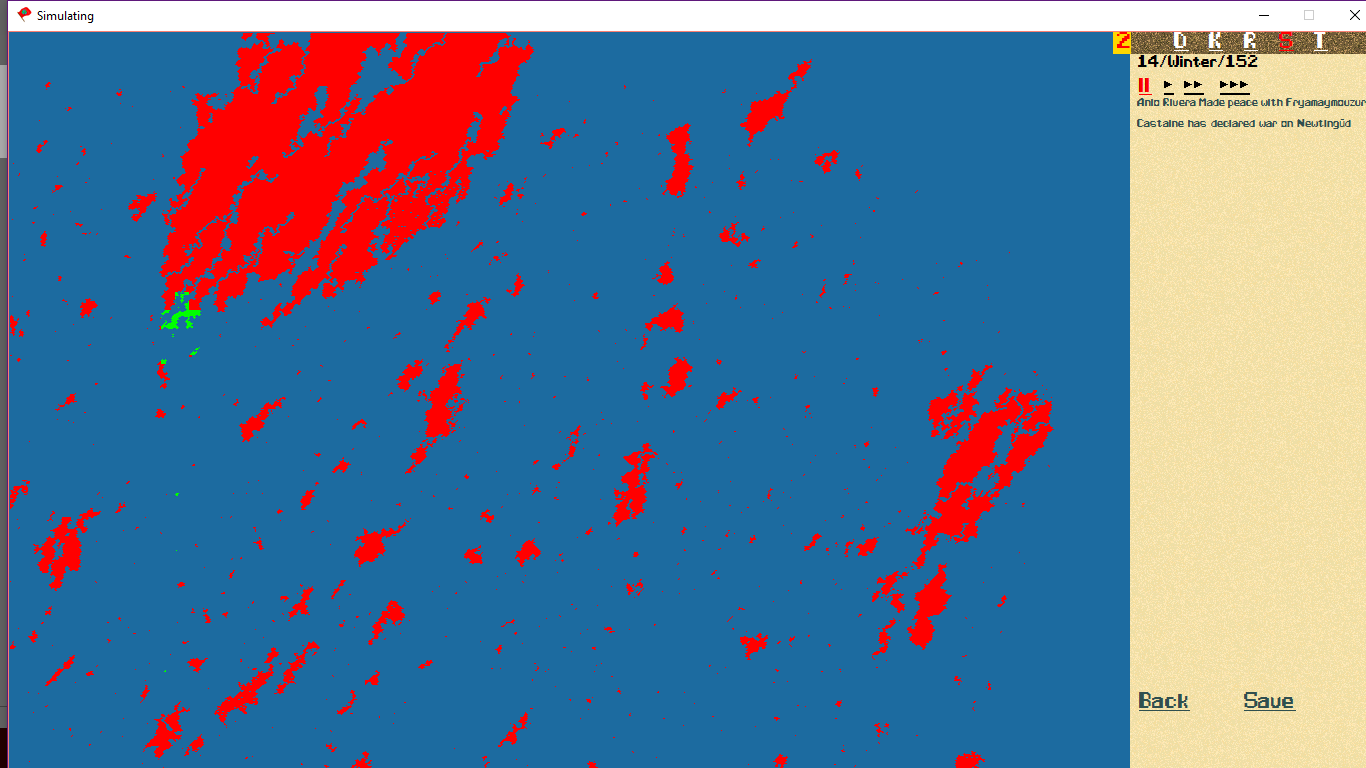
B = 10 if the country is not the capital and has no science – 30 if the country is not the capital and has atleast one science level.

B = 1 if the country is tribal

*Science*

The tools of war have changed and reshaped countless times throughout human history – moving from simple sharp rocks to metal blades, and finally to the modern gunpowder weapons. This is undoubtedly a crucial part of human development, as it has been a major factor in a large amount of conflicts since the dawn of man – as such it was imperative that I simulate this to accurately reflect the evolution of civilization.

In order to do this, I developed a system by which each kingdom has a “Science Level” on a scale of 0 to 255. Kingdoms or above are able to improve their science by randomly being selected by an algorithm, this science can then spread to neighbouring kingdoms (or above). For each 50 levels of science, a new technology becomes available to the countries – these are as follows:

* Bronze (100 – 150%)(Default)
* Iron (150% - 250%)
* Steel (250% - 300%)
* Gunpowder (300%- 500%)
* Oil (500% - 700%)

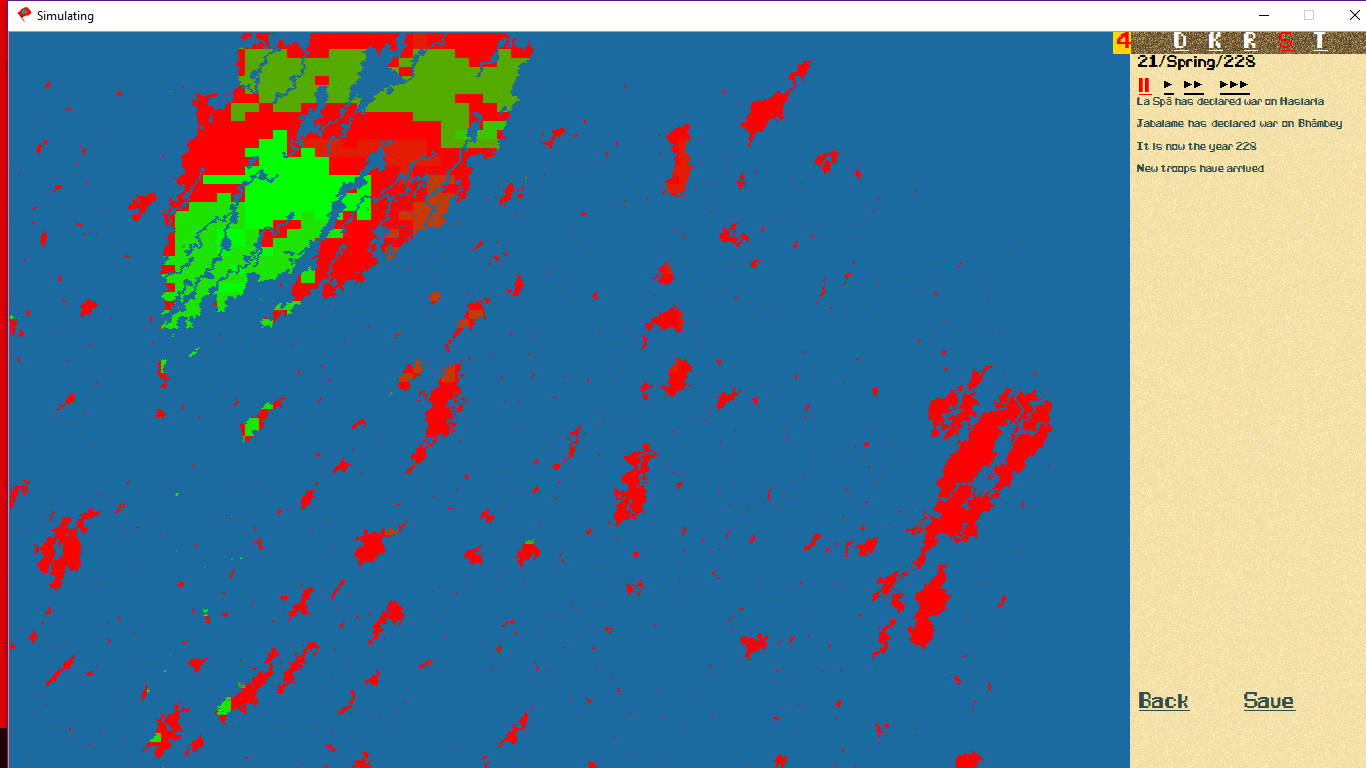
These percentages are randomly determined by the world generation algorithm and are used as a multiplier for the manpower each country can generate – meaning a country at oil technology could generate up to 7 times the amount of manpower a bronze age country could produce. This obviously provides a huge advantage for countries that are in positions to generate science (Meaning countries on the biggest continent on the map, which will be able to gain more science from spreading and will be more likely to generate kingdoms due to the availability of land and religions).

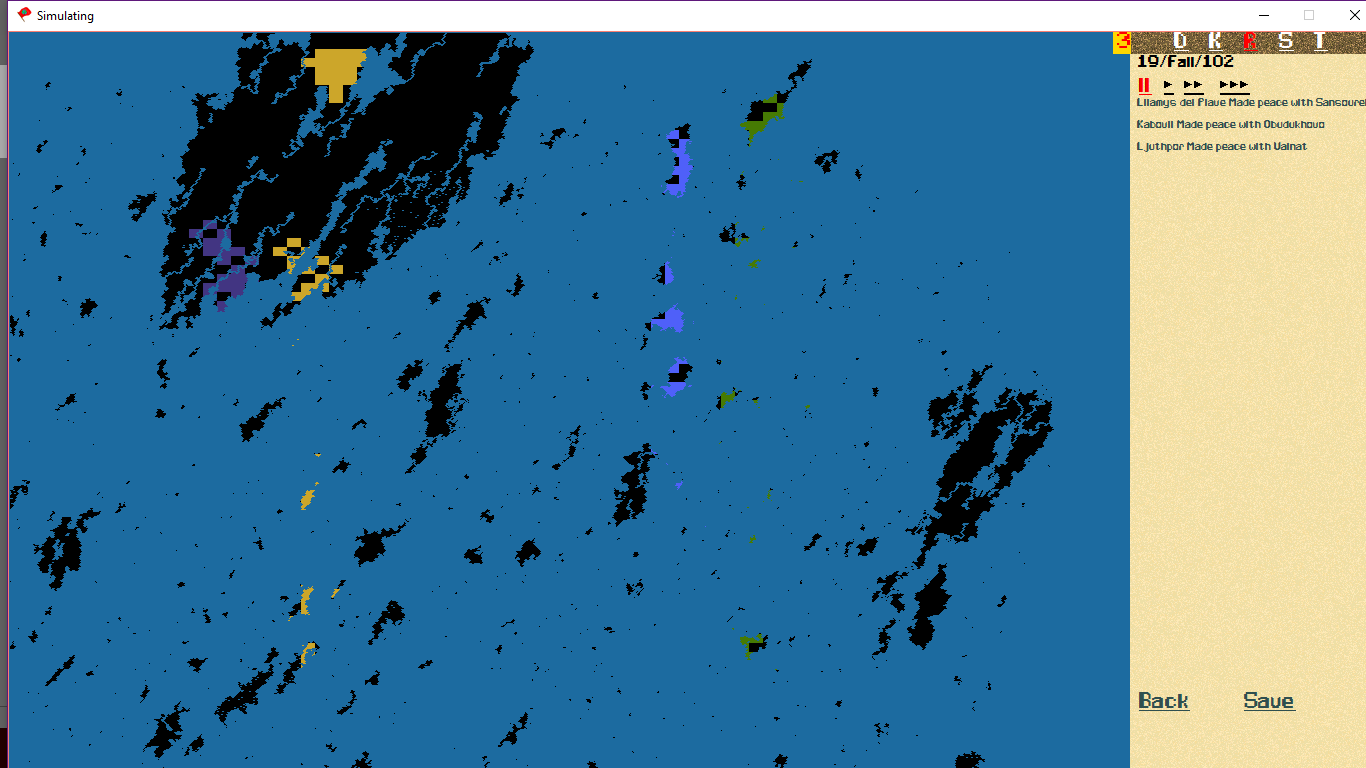
Figure 3

A country that has developed technology, compared to its illiterate tribal neighbours

Figure 4

A continent with varying levels of technology - notice that the two green blobs are disconnected, allowing one to be far ahead of the other in technology

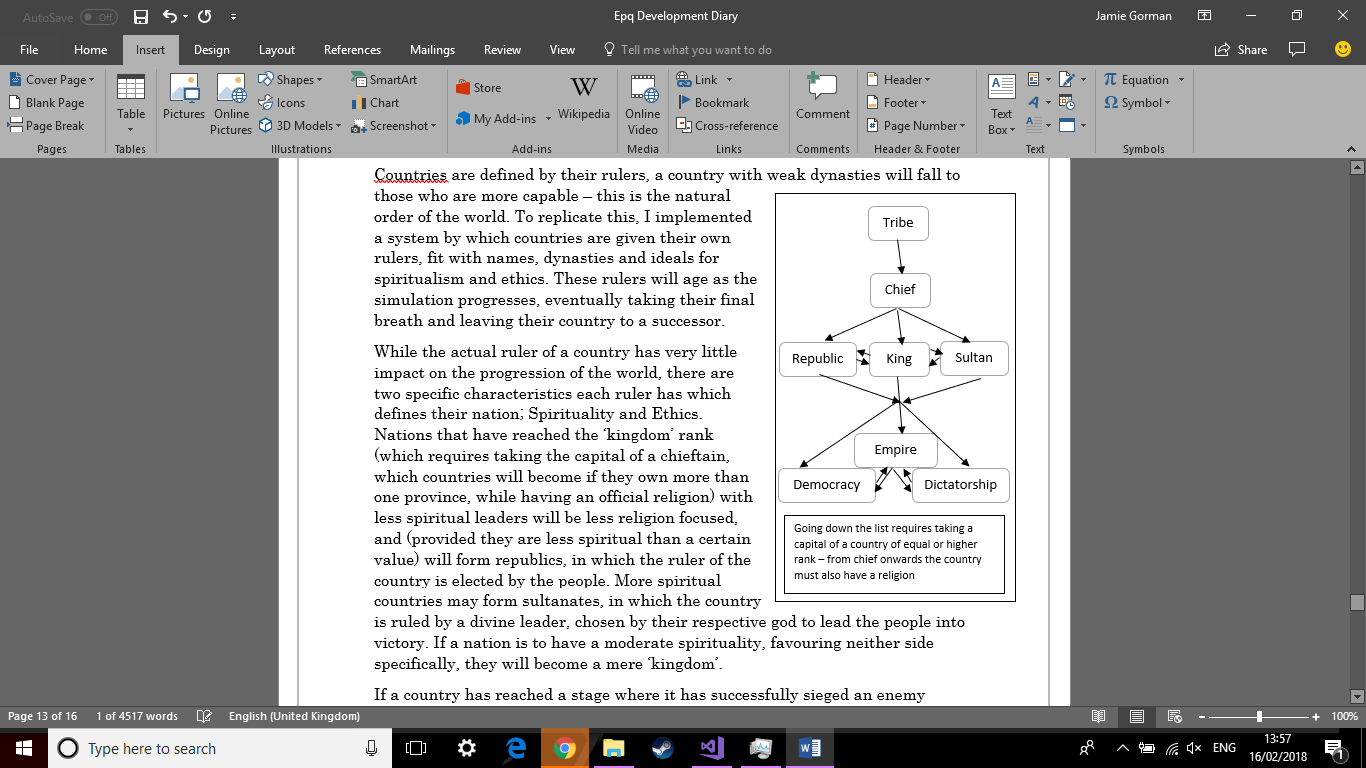
*Religion*

Different cultures develop different ideas for the creation of the world – and more importantly, their purpose as human beings. These differing opinions (which does not necessarily mean they are not factual) have incited thousands of wars throughout history, from drawn out conflicts that lasted through ages like the various crusades or simple disputes over minor territories like with the wars between orthodox and catholic Christians. Regardless of how you view religion and piety – it is an indisputable fact that it has shaped our modern world and its borders, not even to speak of how it has divided people. As it has had such an astonishing impact on the world, I attempted to include a religious divide between countries in my simulation.

Religions will form in any province on land (using the names of ten real religions, which are randomly selected upon map generation) with atleast four other provinces bordering it, from there it will spread as much as possible (due to technical reasons, favouring to expand to the top left, a bug which I was unable to find a simple fix for) countries will also adopt a religion as their official religion, and will attempt to convert any provinces within their borders to their religion, furthering the spread of their beliefs.

Figure 5

Religions spreading around a map, featuring Docetist Christianity and Sikhism

*Rulers & Politics*

Countries are defined by their rulers, a country with weak dynasties will fall to those who are more capable – this is the natural order of the world. To replicate this, I implemented a system by which countries are given their own rulers, fit with names, dynasties and ideals for spiritualism and ethics. These rulers will age as the simulation progresses, eventually taking their final breath and leaving their country to a successor.

While the actual ruler of a country has very little impact on the progression of the world, there are two specific characteristics each ruler has which defines their nation; Spirituality and Ethics. Nations that have reached the ‘kingdom’ rank (which requires taking the capital of a chieftain, which countries will become if they own more than one province, while having an official religion) with less spiritual leaders will be less religion focused, and (provided they are less spiritual than a certain value) will form republics, in which the ruler of the country is elected by the people. More spiritual countries may form sultanates, in which the country is ruled by a divine leader, chosen by their respective god to lead the people into victory. If a nation is to have a moderate spirituality, favouring neither side specifically, they will become a mere ‘kingdom’.

If a country has reached a stage where it has successfully sieged an enemy kingdoms capital, it will become an empire, which (like its predecessor) has two alternative ideologies. Nations with lower ethics (meaning left leaning, not that the country is less ethical) may form democracies, which allow the voting in of new rulers every four years, allowing for quick changes of policy depending on the needs of the people. Countries with exceedingly high ethics (right wing) will form dictatorships, which serve to meet the needs of the ruler, rather than the people it consists of. Countries with a more moderate ethics will form simple ‘empires’.

When a ruler dies, changes may be made to their nation, successors have a chance of slightly altering the ethics and spirituality of a country to fit their ideas, which can lead to the reformation of the country into a new type of government. In rare cases, a country without elections (not republic or democratic) may have their ruler die without an heir, which will cause the successor to have completely random spirituality and ethics, as well as a unique dynasty.

*War & harmonic series*

For empires to expand, conflicts must arise between them. In human history, the reasons for these conflicts could be practically anything, ranging from simple family disputes, or arguing over vague border agreements dating back to feudal times (Such as the French-German dispute over Alsace Lorraine, which can be dated back to the split of the Carolingian Empire, which divided the empire of Charlemagne into three sections, the west, now known as France, the east, now known as Germany, and the middle, which the two countries have disputed over up to the modern day). Because of this, it can often be hard to replicate the complex reasons why a country will declare war on its enemies, though in almost all cases, it comes down to the value of the area itself (which happens to be a driving factor in the middle-eastern conflicts, due to the abundance of oil in that area). To replicate the value of provinces, I devised a formula to calculate which provinces the countries of the world would value the most;

Where P refers to the Provinces Dataset and K refers to the Kingdoms dataset

Using this formula, all provinces in the world would be given a value roughly between 0 and 100, this value would be updated each month, and would be imperative to the AIs decision for war, as the AI will take into account the estimated value of a province in order to prioritise the most valuable land they can take.

Once the values are calculated, each country eligible for war (which changes per month) will weigh up the cost and benefits of declaring a war on the target. If the benefits outweigh the costs by 100 arbitrary units, a war will be declared between the two nations.

After this, the actual ‘war phase’ will begin, meaning each month a number will be generated between one and the sum of each countries manpower, if the number is less than the manpower of the aggressor, it will become the aggressors turn. Likewise, if the number is greater than the manpower of the aggressor, it will become the defenders turn. Then, the country whose turn it currently is will be given the opportunity to exchange some of their manpower for one of the enemy countries provinces.

This acquisition of land will also generate ‘score’ for the kingdom that gains land, which will be carried with the country throughout the war.

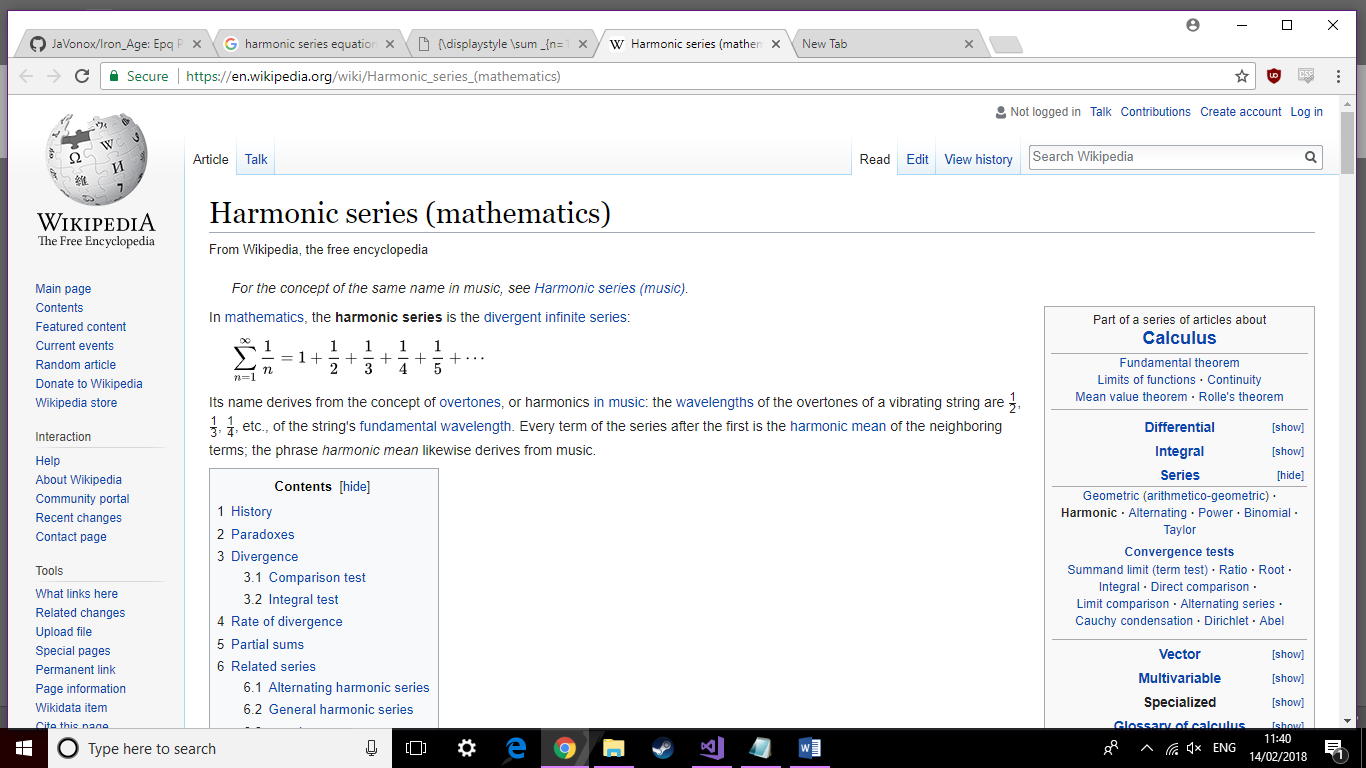
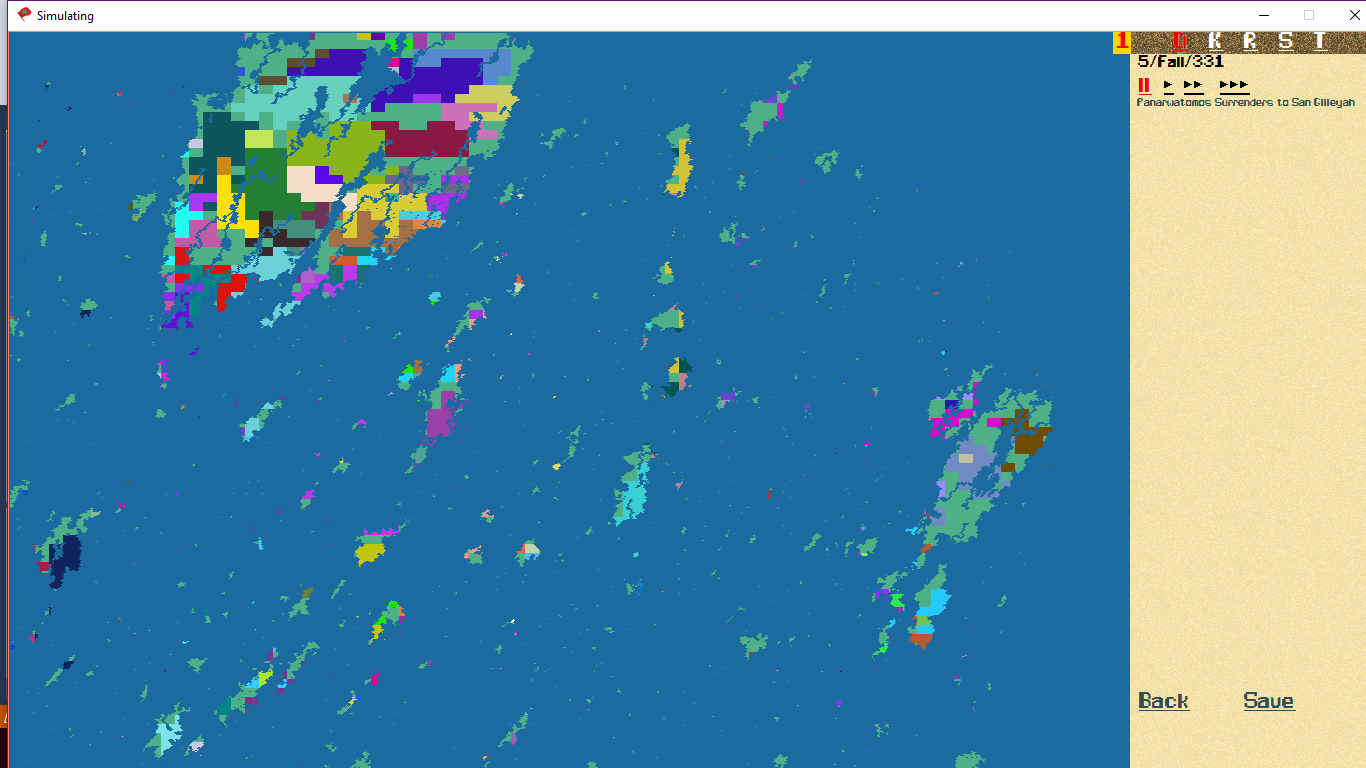
This Formula is a variation of a mathematical formula known as the ‘harmonic series’, in which a value is increased by exponentially small numbers, causing it to have an infinite value. For my implementation, I had the denominator get increasingly smaller, meaning the added value would become exponentially bigger. From this, I used an algorithm to check if the total ever exceeded 60, and if so, the war would end – with both sides taking whichever lands they gained during the war. Additionally, taking a capital city resets the scores, allowing a successful war to gain far more land that would usually be expected.

Figure 6 –

A continent after 300 years of war (before the patch that changed the colouring for chieftains)

Figure 7

https://en.wikipedia.org/wiki/Harmonic\_series\_(mathematics)

Evaluation

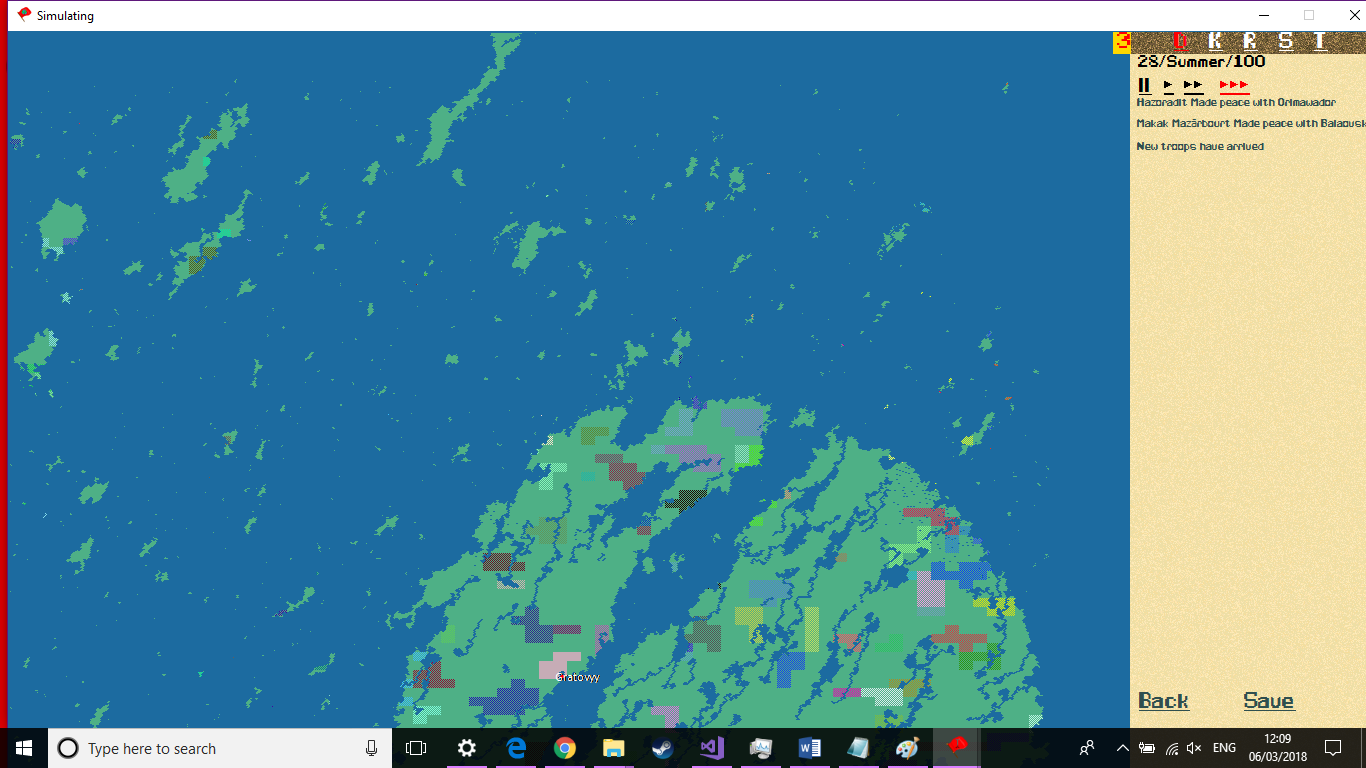
Development ended on the project early in March, as all the intended features had been completed, and there were no outstanding bugs that could be found. This left time to reflect on my completed project, and overall, I found myself feeling happy with the program, but unsatisfied with its lack of quality on release. Many modern development cycles can take upwards of a year to complete the most basic prototype of a system, and that would be with the assistance of a team of developers, designers and artists. As I had significantly reduced time and resources, the project was unable to reach the quality found in professionally made programs, which had been a goal of mine. Additionally, while there were no unfixed bugs identified during testing, there was a significant amount of flaws with the program, namely the time and processing power it takes to run the program, this was due to a difficult issue concerning the time taken to place pixels on the screen, which, despite my best attempts, I could not solve, late into development I chose to replace the twelve months per year system with a seasonal system, which allowed me to reduce the overall runtime significantly.

Figure 8

A late development image of the map, notice the shading on chieftains and name tags for kingdoms

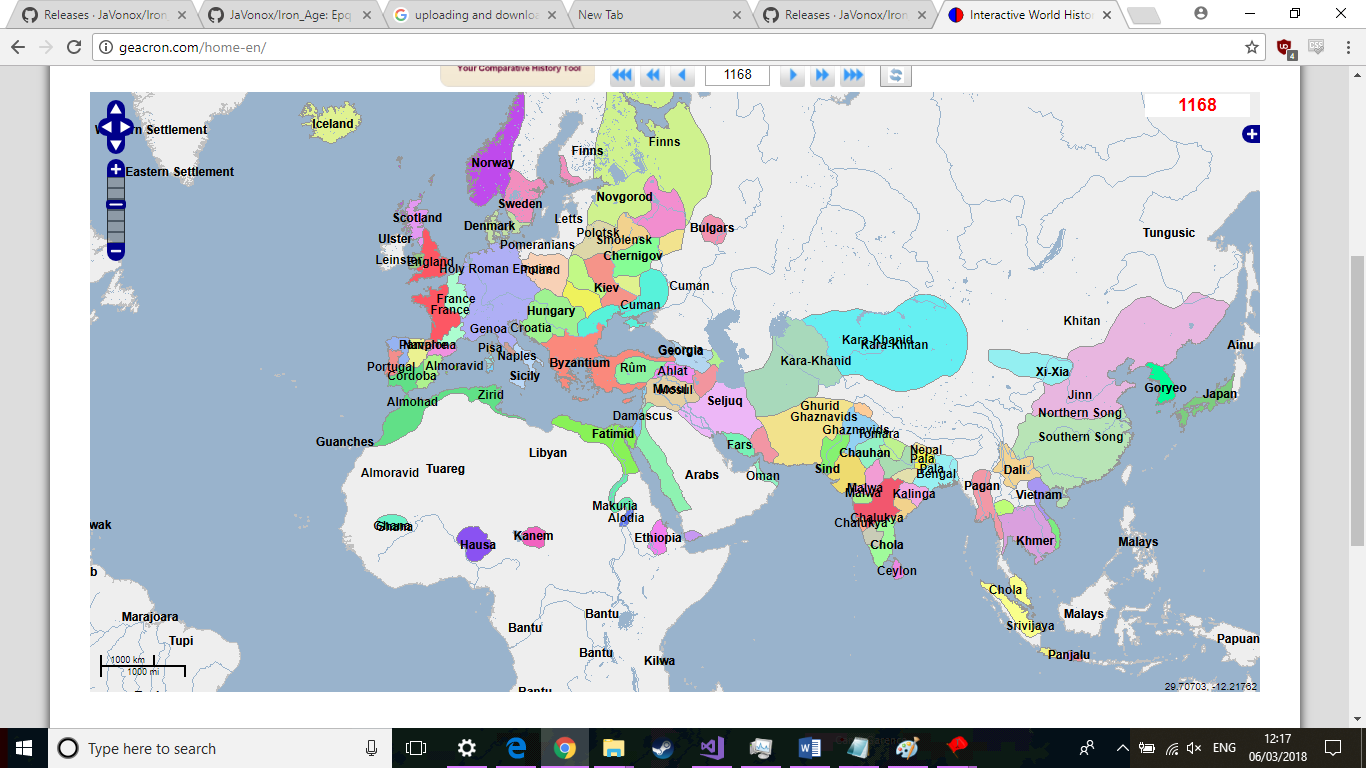
However, it cannot be said that I was completely disappointed in the finished product; The program was able to simulate a worlds history in a accurate way, with nations spreading their borders in ways that simulate intelligence, and would mirror the kind of border expansion that can be seen in history. Additionally, developing the program and fixing its issues taught me significant amounts about programming (specifically the language c#), mathematics (Harmonic series and efficiency calculations) and human history (I used a website called Geacron[6], which provided a map of human civilization throughout history, which I frequently used as a comparison for countries that had developed in my simulation).

Figure 9

A Geacron map of the world in 1168, which closely resembles the type of map generated by my program

Overall, I was pleased with the finished product, but felt there could have been many improvements, had there been more time and resources available during development (as well as if I had started the project with the amount of skill I had during the end of the project, as I had learned a significant amount of things that could have improved development). Thank you for reading this document detailing my progress during development.

Sources

[1] <https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life#Rules>

[2] Images are from:

<https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life>

<http://www.paleotechnologist.net/wp-content/uploads/2010/10/glider.png>

[3] Evolution Simulator:

<https://www.youtube.com/watch?v=GOFws_hhZs8&list=PLrUdxfaFpuuK0rj55Rhc187Tn9vvxck7t>

[4] Evolution Simulator online:

<https://www.openprocessing.org/sketch/205807>

[5] Markovs chain:

<https://en.wikipedia.org/wiki/Markov_chain>

[6] Geacron:

<http://geacron.com/home-en/>

The Full program can be downloaded from:

<https://github.com/JaVonox/Iron_Age/releases/> - To run the program access the exe file which is within the FILE > EXECUTABLE directory. Instructions are included in the README