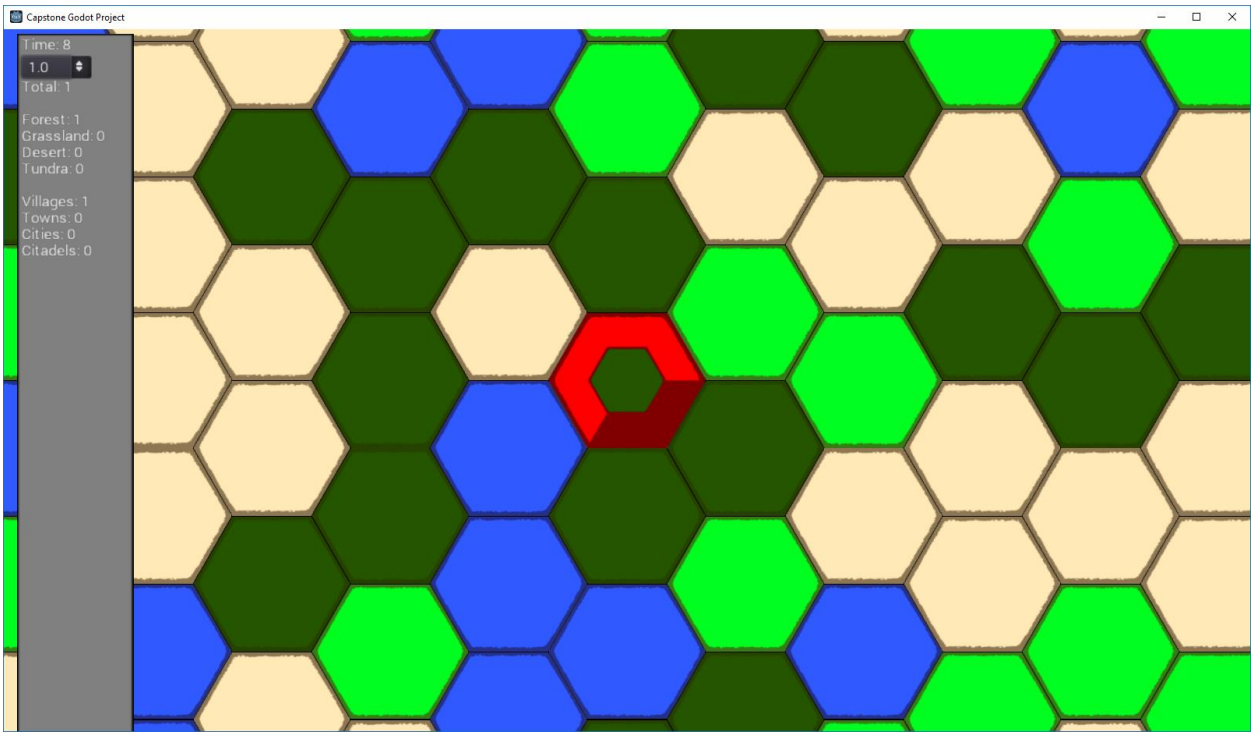
Using Cellular Automata to Populate a Simulated World

Nickel Revie

Dr. Reinhart



I. Abstract:

Simulations are an integral part of society. It allows people to imitate conditions to review specific events or predict future ones. This project aims to use cellular automata, a system where conditions of surrounding cells influence the state of one cell, to simulate population growth in an uninhabited area. Cellular Automata are used to study patterns and other processes and are even applicable in studying specific systems in the real world, such as the Belousov-Zhabotinsky chemical reaction [1] and patterns of some seashells [2]. Other applications include use in cryptography by using two dimensional cellular automata for random number generation [3]. Using the geography of cells, this simulation using cellular automata is somewhat predictive in its behavior to colonization of an area. Populations will colonize areas that are easier to live in unless no other conditions exist, thus we can observe how a population colonizes the world based on the geography layout of the world. This project aims to achieve a simulation where the population will begin to colonize areas of the world where humans normally inhabit, in addition to areas humans sometimes will inhabit merely because they are able to. In our simulated world, we generate a world of hexagonal tiles and randomly assign a geography to each space. Each geography type (forest, desert, grassland, etc) that has a food value and a difficulty of living value. One tile will start the simulation colonized and will spread over each time step towards less difficult tiles or tiles high in food availability. The populations on each tile will grow based on how many people are surrounding their home tile and how much extra food there is available. Thus population density is accounted for. In this way, we see that population growth can be represented in cellular automata.

II. Problem Description:

Populations can be modeled through numerous ways, cellular automata could be used as a method to examine how populations move through an unpopulated world filled with sources of food and water. Cellular automata take into account the status of the cells around it in order to change and evolve over time. In this way we can simulate population change over time and how geography affects it.

III. Description of the method of solving the problem:

The simulation will contain a geographical map made up of hexagonal tiles. This map will be randomly generated and each space on its grid will be a sort of geographical feature, such as grass, forest, desert, water, etc. Each of these spaces will also have two values that is affected by its geographical feature: food, and difficulty. The population will expand based on the availability of food and water and will move in directions that are less difficult. Tile populations are split into four city types: Villages, Towns, Cities, and Citadels. This is representing the types of cities that would be made hundreds of years ago, when populations were more dependent on land.

IV. Methods and Results:

At the beginning of the project, a design document was created that would roughly provide a map to programming what was needed. Over time the document had things added and taken out, although a rough sketch of all the features were in from the start. A few features ended up not making the cut however, mostly due to time constraints.

In our program, the world is a randomly tile map generated with three different tile types: water, forest and grassland. Taking into account geographical locations, a percentage of forest and grassland tiles are then turned into deserts when they are near the equator and tundra when they are near the poles. This allows us to represent a more accurate geographical map. Tiles bordering water will provide more food base on how much water surrounds the tile.

The population starts on one tile in the world. In the simulation it is placed in the center of the world to understand how the population expands while having the most area. When the population is placed, that tile becomes occupied. Occupied tiles consume available food and will expand once every twelve time steps (referred to as turns) when it is a village. The more developed the tile is, the more people will live on it and the faster it will colonize nearby spaces, although they are less likely to have places to move to. Each turn, the tile will check its development status as well as the development status of other nearby tiles. The development status is related to population density. As the number of occupied

bordering tiles increased, the populations will attempt to make the central point a larger city, thus changing the city type of the tile.

Villages are the lowest city type, consuming no food on the tile. Towns are made when a village is surrounded by at least four other villages or larger city types. Cities are made when a town is surrounded by at least four other towns or higher. Cities become citadels when they are surrounded by four other cities or citadels. Towns will consume one food extra food, cities will consume two extra food, and citadels will consume three extra food. Thus only tiles that provide enough food will become larger cities.

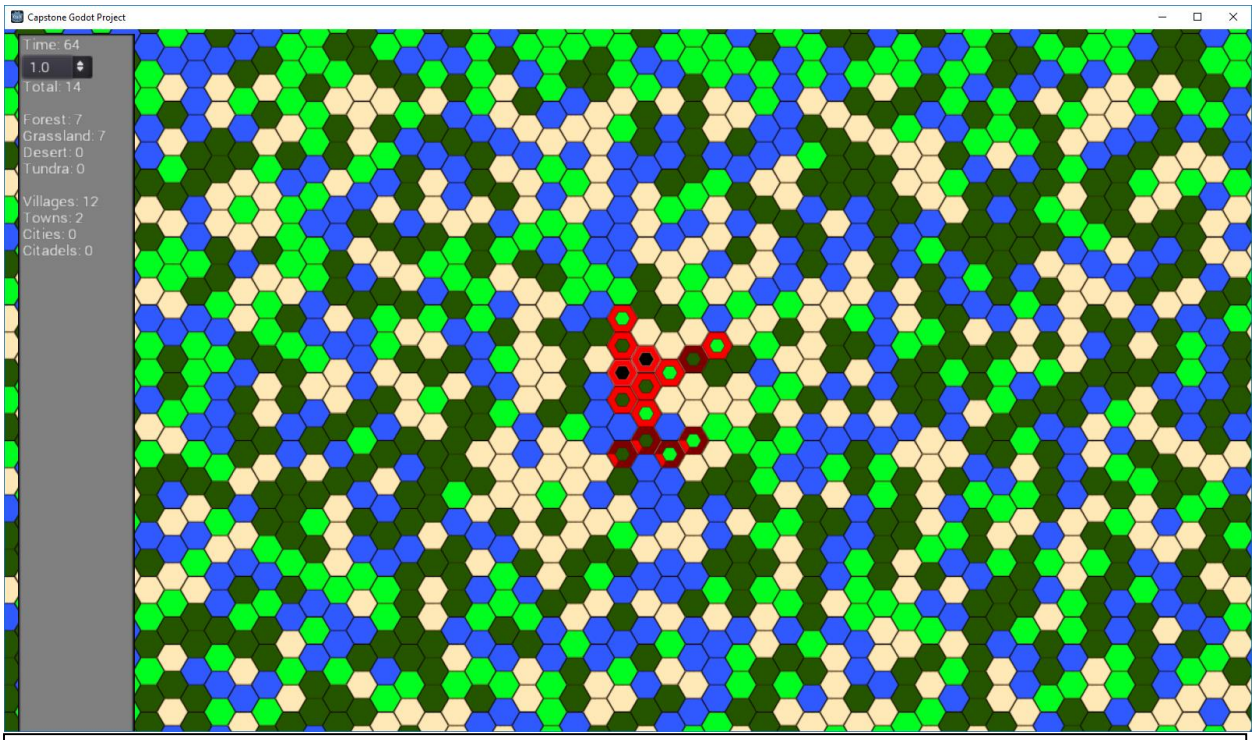


Figure 1. Beginning of the simulation. The center was colonized and the population has begun to spread, avoiding desert tiles. As they are not as easily colonized.

The node system in the Godot Engine is very nice to use and easily facilitates rapid prototyping and allows easy organization. There were a series of nodes made representing each specific part of the simulation. The main node was the world node, which stands as a base for the whole project. Connected to it is the tile manager, the camera and the user interface nodes. The camera is controlled by a script that takes in user input and will move about depending on the input.

The user interface is a series of child nodes that make up the table that represents the information that shows up on the screen. It also includes the time step options, to have the time steps be at one second, half a second or a tenth of a second intervals. The main UI node also contains the script, which accesses the child nodes and pulls information from the tile manager to display the information on the screen using the child nodes.

The tile manager is a node with an attached script that acts as the manager for the rest of the simulation. It generates the simulation and serves as the parent for all the other systems. It has a timer attached to it that will tick based on the set interval timing. The tile manager creates instance of the node for the tile map and has methods managing the distribution of information to the user interface and methods to control the timer. Also, when the timer is triggered, it will call the update method in the tile map node.

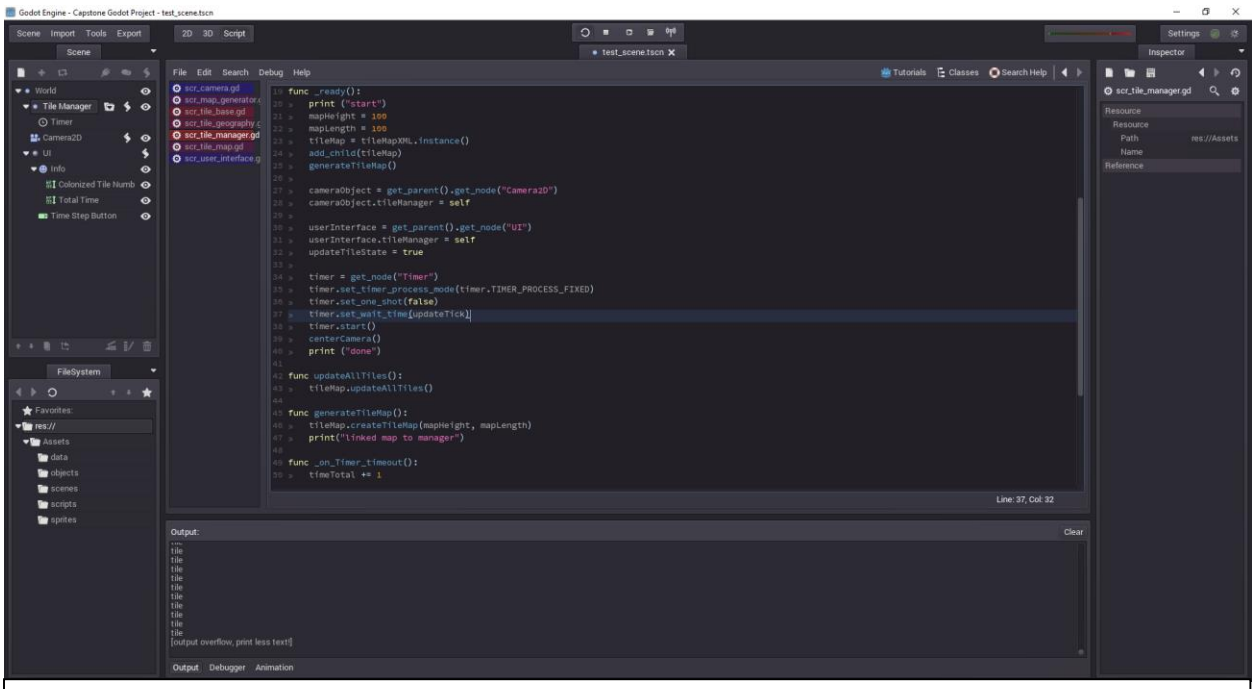


Figure 2. The code editor for the Godot engine as well as a look at some of the user interface of the game engine. Code shown is from the tile manager script.

The tile map node is the parent node for the entire tile map, and the script contains methods that create the two dimensional array of tiles and set their positions onto the map. First it will generate the

base tiles without any geographies mapped to any of them. It will then create the geography map, and uses the map generator node, which it instances in order to generate a random geography type. The geography generator takes the tile's position as an argument, so it will know when to spawn a tile as a location based tile (desert or tundra). After generating the geography map, the tile map will update the food count on all the tiles based on the number of bordering water tiles. In addition, to this, the starting tile in the tile map is placed in the center of the map, and every time the timer calls update, the information on the tiles is sent back to the user interface to display on the screen.

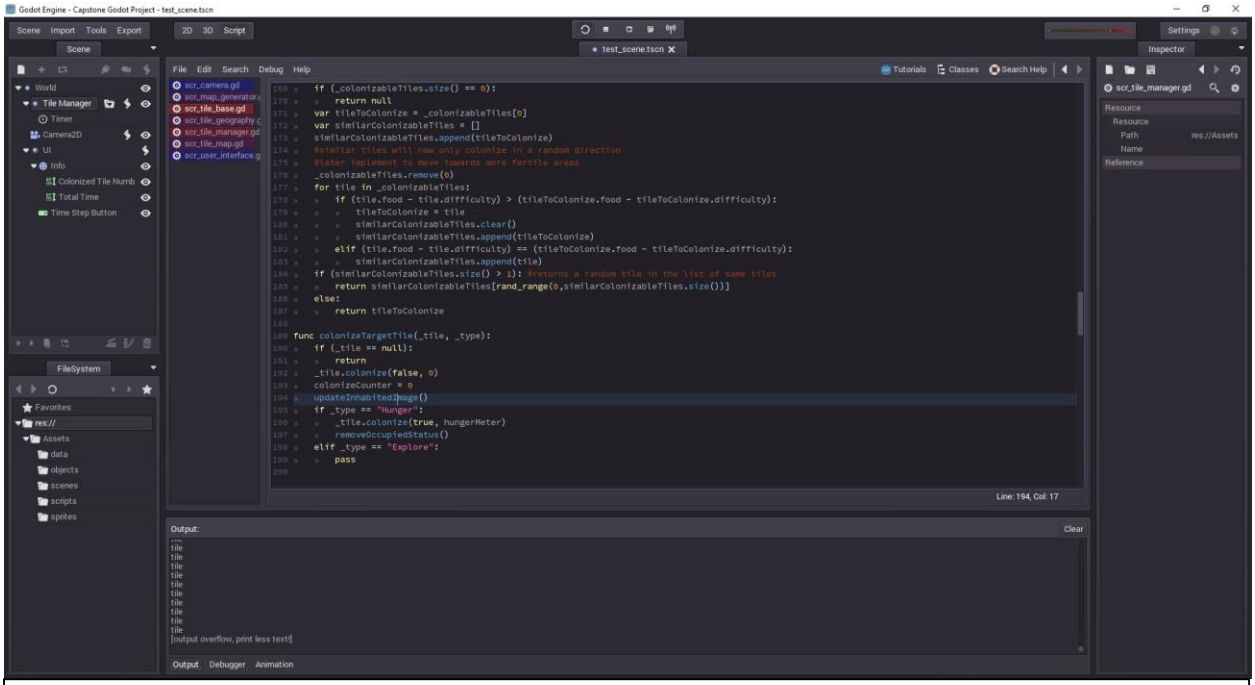
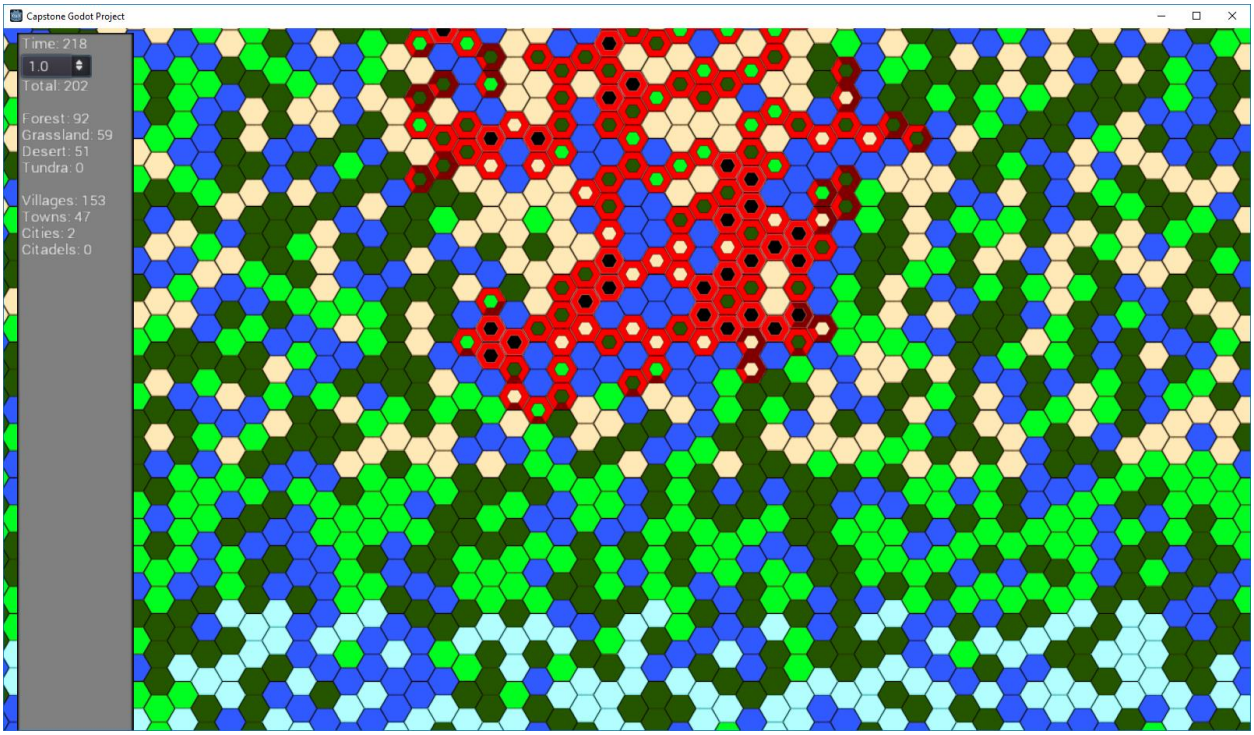


Figure 3. Code in the tile base script. The code shown shows how the tile to colonize is chosen, as well as the method for colonizing a space.

Then finally there is the base tile node. The node contains all the methods to make the tile perform all its duties. There is a large array of stats that are recorded. It contains a main update method that will be regularly called. In that method, if the tile is occupied, it will update the status of the tile surrounding it, and then it will check the food status. If the tile is hungry, it will perform specific functions for hungry tiles, such as the population moving to a new area. Otherwise, it will update the colonization counter and will initiate the colonization of a nearby tile if the counter is full. When

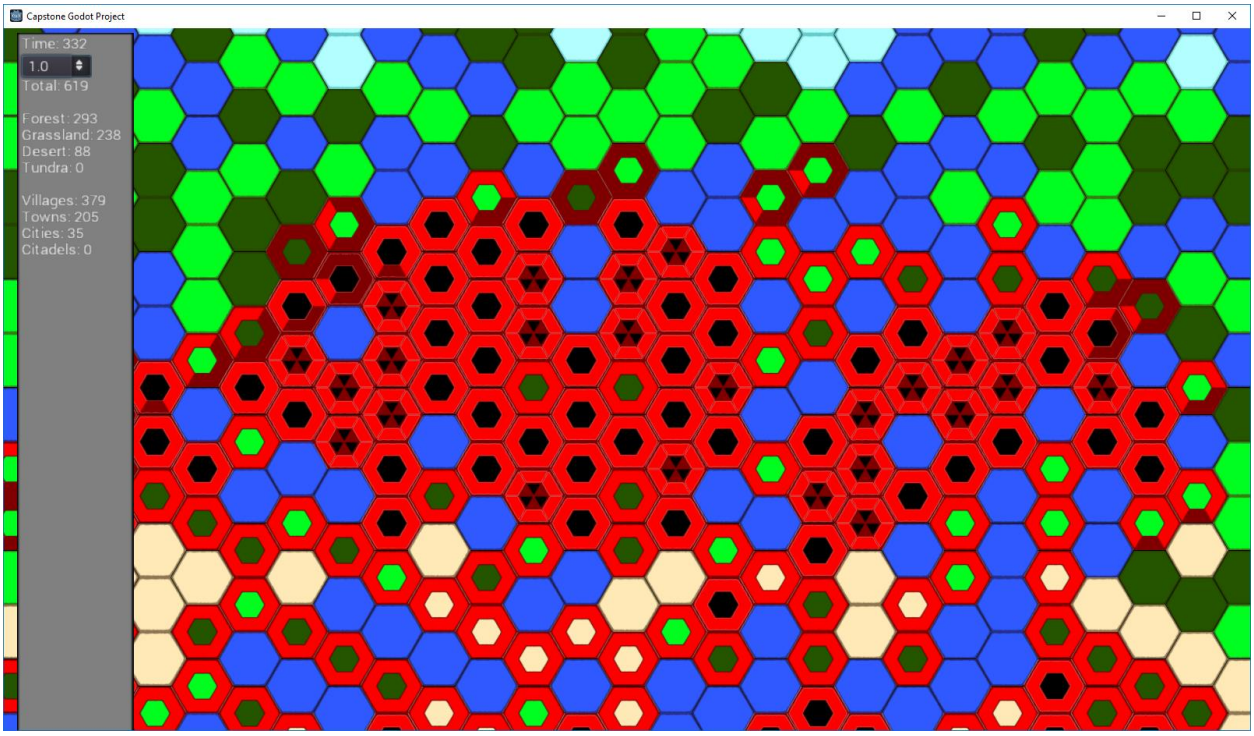
colonization is initiated, the tile will look at nearby tiles and check their habitability. It will ignore any already colonized tiles, and then create a list of tiles if their food value is greater than their difficulty. It will then choose the most habitable tile (the one with the biggest positive difference between the food and the difficulty). If there are multiple tiles with the same habitability, it will randomly choose one of the available tiles and colonize it.



When a tile is colonized, an inhabited sprite is instanced over the sprite, but under the city indicator, thus allowing the user to know that the area is occupied. This node instanced by the code has multiple sprites that will change based on the colonization status of the tile. Since each tile is a hexagon, the colonization counter is represented as a slowly filling bar around the hexagon. Each side of the hexagon represents two points on the colonization counter where the counter has a maximum of twelve before the population will expand to another available space (if any). The colonization method also takes in several arguments pertaining to the way it can be colonized. Currently, the only two types are regular colonization, and if a tile is hungry, it will move one space in an attempt to find a more habitable area. A

third type, exploration was to be implemented, but was not able to be implemented in the time allotted. This method was originally several methods, but it was refactored to reduce the code complexity.

Then there are the methods pertaining to the population density system. When the method is called, each update step, the tile will look at what the other surrounding tiles are in the area. It will then update the development of each city based on how many of a same or higher value are surrounding it. The specifics of this are referenced above. When a tile increases in level, it will also decrease the food value of the tile. Depending on the available food, a tile may not be able to move up in development since, the land cannot support enough people.



The population expands in a fairly predictable pattern. The algorithm for the population expansion prioritizes the more habitable areas first, but they will eventually colonize every habitable area. This means that all forest and grassland tiles become colonized and a large number of tundra tiles are colonized as well. Deserts cannot be colonized unless there is enough water bordering them, so the deserts are generally not as colonized.

V. Conclusion:

This simulation is pretty effective at representing population growth, although there are no constraints in the current system that limit population growth based on more complex geographical factors or even political factors. At some point, the population ceases to be one cohesive unit and divisions appear. Dealing with the more geographical changes, there are other factors that can influence populations. Rivers, seas and oceans can limit people and there are more varied environments around on Earth. And certain areas could contain deadly predators, a canyon can be carved into the landscape to prevent colonization, and also, the map is randomly generated and not procedurally generated, so the map created looks less like something that would happen in real life, and more like a conglomeration of different geographies mashed together.

Fixing the geographical part would definitely help in looking at colonization. Moving more into an actual simulation as opposed to keeping it strictly cellular automata would likely also make it more accurate, although this model could still be similarly accurate even if it stayed in the realm of cellular automata.

Overall, I felt like this project was a great test of my skills and abilities. I used my software engineering skills to create design documents and prepare what I was going to make in advance. I used multiple tutorials found online and my own skills to figure out how to do many things. Several times I went through and refactored multiple parts of the code, making it less complex and making it easier to scale. Assuming this project expanded and added many more features, the process of refactoring my code would have made it much easier as adding new features would not have required rewriting large portions of my code. Over all I used knowledge that I had learned in class and through other methods in order to make my project a culmination of my abilities.

VI. Bibliography:

[1] Turner, A.; (2009) A simple model of the Belousov-Zhabotinsky reaction from first principles. Bartlett School of Graduate Studies, UCL: London, UK

[2] Coombs, Stephen (February 15, 2009), *The Geometry and Pigmentation of Seashells*, pp. 3–4, retrieved September 2,2012.
[3] Tomassini, M.; Sipper, M.; Perrenoud, M. (2000). "On the generation of high-quality random numbers by two-dimensional cellular automata". *IEEE Transactions on Computers* **49** (10): 1146–1151. doi:10.1109/12.888056.
[4] godotengine.org and docs.godotengine.org
[5] https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life#Rules
[6] Various Youtube tutorials on using Godot Engine

VII. Appendix (Original Proposal):

Using Cellular Automata to Populate a Simulated Planet

Description of the general area of study:
A cellular automaton is a collection of cells displayed on a grid of some shape that adheres to a set of rules and evolves over a number of time steps based on states of neighboring cells. Cellular Automata are used to study patterns and other processes and are even applicable in studying specific systems in the real world, such as the Belousov-Zhabotinsky chemical reaction [1] and patterns of some seashells [2]. Other applications include use in cryptography by using two dimensional cellular automata for random number generation [3].

Description of the specific problem to be addressed:
Populations can be modeled through numerous ways, cellular automata could be used as a method to examine how populations move through an unpopulated world filled with sources of food and water. Cellular automata take into account the status of the cells around it in order to change and evolve over time. In this way we can simulate a sort of population change over time and how geography affects it.

Description of the method of solving the problem:
The simulation will contain a geographical map projected onto a 3-dimensional sphere. This map will be randomly generated and each space on its grid will be a sort of geographical feature, such as grass, forest, desert, ocean, etc. Each of these spaces will also have 3 values that is affected by its geographical feature: food, water and difficulty. The population will expand based on the availability of food and water and will move in directions that are less difficult. Over time we will see how the population will expand based on the availability of resources and ability to navigate their terrain. In addition, projecting the map onto a sphere will allow it to look more like a planet.

Tasks
1. Create a grid that can be projected onto a sphere.
2. Program basic cellular automata and put it onto the grid.
3. Program in traits that cells will use to multiply and populate the planet.
4. Add in geographical features and resources to the map that the cells will use to populate.
5. Create a class to randomly generate the grid with the geographical features and resources.
6. Create a sphere using graphics.
7. Convert the cellular automata to display on the sphere.
8. Simulate
9. Report and Presentation

Weekly Schedule
Week 1: Task 1 Completed

Week 2: Task 2 Completed
Week 3-4: Task 3 Completed
Week 5: Task 4 Completed
Week 6: Task 5 Completed
Week 7-8: Tasks 6 and 7 Completed
Week 9-10: Simulation and Bugfixes

Materials Needed:
Only software is needed. In this case, Java will be used.