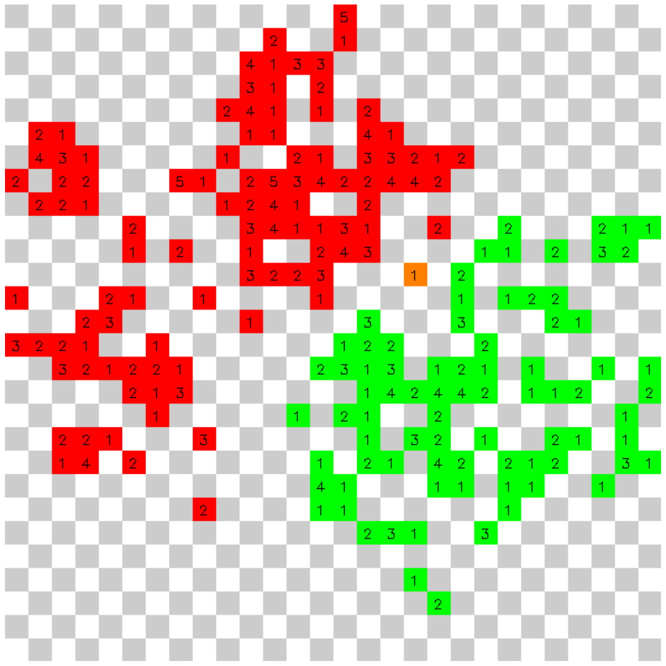
CMPS 112 Project Report
Ashley Rocha
Nicolas Hahn

Bacterial Combat Simulator



The goal of our project was to make a game entirely in Haskell. We thought this would be interesting because most games are created in imperative languages, and being functional, Haskell would provide a very different experience in programming a game. We also wanted to know if everything that can be done in an imperative language could also be done in a functional one. Since every programming language that is Turing complete should theoretically be able to do the same tasks, this had to be true, but the paradigms are so different that we needed to find out for ourselves. We also wanted to know if Haskell could even be a good language to program a game in, perhaps functional programming would lend itself to game making in some way we couldn't predict.

Mechanics

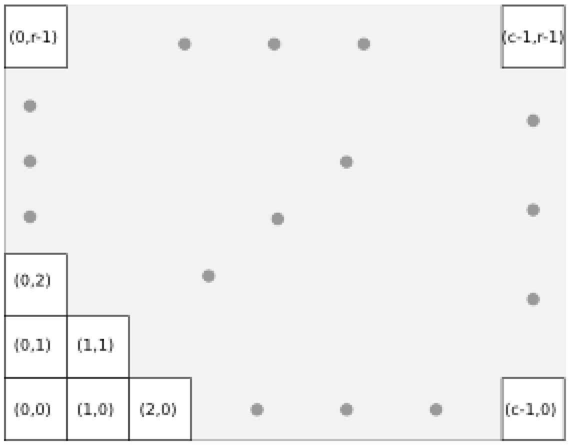
The idea is to have two colonies of bacteria that would grow, eventually meet and fight each other, and then one would win when the other colony is completely wiped out. The game takes place on a grid, and each colony starts off with one grid square on either side of the grid, each with a population of 1 (this signifies how many bacteria cells are living on this grid square). At each time step, every bacteria in the colony's population rises by 1. If it hits a certain number, then it has a chance to spawn a new cell on an adjacent grid square, which will also decrement its population. This is how the colony grows. If this causes the original square's population to hit 0, then the square reverts back to neutral territory. When squares from the two different colonies are adjacent to each other, their populations decrement each turn until one of the squares is gone. The remaining square's population resumes incrementing. Generally, a colony wins by pinning the other in a wall or corner, and occasionally when the entire colony is surrounded.

Currently, our project has two versions: a simulation, which does not take any user input and each colony grows randomly until one of them has been completely wiped out. There is also a game version that allows the user to control one colony by selecting a specific cursor grid square, and the controlled colony, instead of growing randomly, will choose to spawn in squares that are closer to the cursor. The enemy colony simply chooses to grow towards the average position in the user's colony (middle of the clump of cells). There is also a cap on how large a colony can be, otherwise it becomes too difficult to control. The simulation does not have this cap.

Libraries

Grid

Grid provides tools for working with tiles of various shapes. These tools were designed with game development in mind and provides a lot of flexibility. A "grid" can have borders or it can be unbounded, either running infinitely or looping in on itself. Shapes can have any number of sides to account for many different kinds of movement. We just used square tiles on a bounded grid for our game.



Since our needs were fairly simple, we only used Grid for a few basic functions. We used it to populate the game board itself so we could reference every cell's position. The "neighbours" function helped us determine whether a cell had an adjacent tile in which grow, or if it had to fight an opponent colony, or if it was stuck. We also used its "distance" function to find the shortest path between two points, allowing us to guide a colony's growth towards a certain point. This library proved to be a tremendous timesaver. While these functions are simple enough for us to create ourselves, this project was overwhelming enough that we were looking to save time wherever possible.

Gloss

There is not a single standard graphics library in Haskell. There are plenty of frameworks to choose from right on the Haskell website, but many of them are incomplete, have been abandoned, are poorly-documented, or just didn't suit our needs. When researching potential libraries to use, we found a few recommendations to just make the front-end in another language entirely because of how difficult it would be otherwise. While this is probably the approach we would take if we were to seriously develop a game in Haskell, the purpose of our project was to discover how to make a game entirely within this one language, so we had to find a suitable graphics library. That library turned out to be Gloss.

Gloss advertises itself as a user-friendly way to draw simple vector graphics and simulations. Its tagline is "Get something cool on the screen in under 10 minutes," and while we didn't find it to be quite that quick (it took longer just to get Cabal to cooperate), it was certainly simple enough for a couple of novices to learn work with. What Gloss doesn't advertise, but we ended up discovering, is that it also provides an interface for user input via keyboard or mouse. It has IO for a simulation and a game[1], making it extremely easy to design our process around making a simulation first and then converting to a game if we had time.

The library comes with several examples and offers some documentation on the GitHub page, but ultimately it is not too friendly for a complete beginner to figure out. This was unfortunate for us as it was the very first thing we had to get down. Fortunately there is a very nice tutorial someone made on how to make a basic Asteroids clone with Gloss[2], and we used this as the first stepping stone to building our game.

Timeline

In our proposal we stated that "the biggest risk is being able to get graphics working." It turned out that almost every step of our timeline was a big challenge to figure out. Despite this, we did not stray too far from our proposed timeline. Here is how our development ultimately played out:

- spent a while trying to install libraries, then had to learn them
- started with example code (asteroid) worked first on printing the game board
- then making things move over time (simulation)
- building a growth algorithm
- added another player
- modifying growth algorithm
- added fighting algorithm
- tweaking them until we get something we like
- building a version that takes keyboard input (game)

Overall, 2 people each averaged 6 hours a week of work over 7 weeks. This approximates to about 84 hours of work from the initial steps to finishing the program.

[1] https://github.com/benl23x5/gloss/tree/master/gloss/Graphics/Gloss/Interface/IO

[2] http://functional-programming.it.jyu.fi/chapters/ProjectAsteroids.md

What did we learn?

From this project, we gained a lot of experience with Haskell, and are much more aware of its positive and negative qualities in comparison to imperative languages. What we liked about Haskell was that it was a very concise language, and less code means less opportunities to create bugs. This also means it's harder to write bad or confusing code. If you're doing something wrong, it becomes clear more quickly than in imperative languages. It has no side effects, so for any one thing we're doing, we don't need to worry about what else in the program it affects. Also, most often if the code compiles, then it works as you expect it to. Rarely did we have behavior we didn’t expect if our code ran. The amount of time we spent with a functional language allowed us to internalize functional programming techniques, and some of them we can also use in writing imperative code as well. For example, having a lot of small functions is better than one huge one, and if a language has functions like map, filter, or reduce, these are generally shorter and clearer than writing for loops.

There were some things about Haskell we didn't like. Sometimes, side effects are desirable, especially in games where global variables are useful. It makes something like showing the number of cells for each team during the game difficult. In an imperative language, we would just need an int that we could increment and decrement as needed. But in Haskell, to do this we would need to pass the entire list of cells to the function that outputs the score, which is unnecessary computation. Additionally, because of Haskell's stateless nature, dealing with random numbers is not as trivial as in imperative languages. In Python, for example, every time we call the random number function, it will give us a new pseudorandom number generated based on some time-sensitive factors (system time, fan speed, etc). But because everything is stateless and happens 'at once' in Haskell, this will always give the same number. Therefore, to generate new random numbers, each time we use a random number, we must feed it back into a generator to get a new random number, and pass it along with the results of the function we used it in.

We also learned how to use Cabal, Haskell’s package manager, and Github for version control and sharing code. Gloss was also our first introduction to graphical programming, and I believe what we learned from this project will transfer over to other graphical applications, even in other languages.

What we would have done differently?

We jumped into this knowing nothing about Haskell beyond the few chapters we had read in “Learn You a Haskell.” Since we were learning about lists, we decided to use that data structure for everything. A colony does not have a definitive end or beginning though, so it couldn’t really be “ordered” as it grows. This meant we’re iterating through the entire list for each colony several times per turn, many times only to alter data in the middle of the list, which is a costly approach. Even though it’s just displaying some colored squares, this game hangs if the colonies are too big. In retrospect the colonies should have been held by a more efficient structure, perhaps vectors.

It might have been better to have simply recreated an existing game rather than trying to design a new one. We spent some time trying to figure out how this could be a game, which was time wasted in the context of this assignment.

We could have taken advantage of functors/applicatives/monads if we had known about them sooner. Looking back, our Haskell code feels somewhat basic compared to what was taught in class. This is not necessarily something we could have done differently in the context of the assignment but something to explore if we were to seriously expand upon this game in the future.

Still to do

Although we do have an interactive version, I would hesitate to call it a game. The general strategy is too simple and the AI is easy to beat. There are a few modifications that could make it more interesting, however.

An idea we had would be to spawn 'food pellets' randomly throughout the grid at certain time intervals. When you spawn a bacteria cell on top of one, it would temporarily allow your colony to either grow more cells per turn, increase population quicker, or increase your colony's cap. We could also have poison pellets which would do the opposite.

The AI is currently not very smart, growing directly towards the opposing colony makes it easy to get surrounded. We could program the AI to try wrapping its cells around the player's colony, or if we implement food pellets, have it switch to growing towards the food pellet before the player can reach it. Another option is to have a two-player mode.

Obstacles on the map could make it more interesting, if we had walls, islands, moving objects, or perhaps a non rectangular grid could make things more strategic.

Using the keyboard to move the cursor is not ideal. To select a specific point in the grid, a mouse would be more intuitive and faster, especially if you want to change the direction your colony is growing in very quickly.

Conclusion

It is absolutely possible to make a game in Haskell, although it certainly was a challenge. While we didn't make this game in any other programming language, we're confident it would have taken less time to complete and that we would have accomplished more in the given timeframe. However, this could also be due to our lack of experience with Haskell, given that functional programming goes against most of what we've been taught up until now. It's difficult to determine whether or not it's worthwhile to make a game in Haskell because we still have so much to learn. We believe we can do everything imperative languages can in Haskell, but with vastly different techniques.

https://github.com/nicolashahn/haskellGame

https://www.youtube.com/watch?v=mLZGIqlDHc8