Computational Physics Final Project

Paul Miloro

May 2, 2019

Abstract

In this project I create a custom, from-scratch implementation of Conway's Game of Life (and related cellular automata) with Javascript and WebGL, exploiting GPU parallelization to increase performance and allow easy user interaction in a browser-based format. This successful and fairly simple implementation of GPU programming further increases my appreciation for the former's utility as a computational tool, and I hope to continue using them in the future.

1 Goals & Inspiration

I, like many others, have always found cellular automata interesting; emergent complexity is always fascinating to observe, and it can be fun to tinker with a self-contained ecosystem on one's computer. Likewise, as far as I am aware all of the freely available Game of Life simulators utilize CPU rather than GPU processing, and there is limited support for browser applications (e.g., most programs must be installed and configured before use, which requires some level of patience and technical skill not everyone possesses). Thus, I figured that this was a perfect opportunity to use the Javascript, HTML, and WebGL techniques we covered in class, to develop my own Game of Life simulator. My primary goals were as follows:

1.1 Portability & Efficiency

As the parallelization offered by GPUs is the main selling point in any WebGL implementation, I wanted the resulting application to achieve consistently high framerates, even for large and dense grids, as well as to be usable on a wide variety of browsers. As the current version of WebGL is supported essentially everywhere, and since I had previous experience with OpenGL, I judged this to be reasonably easy to achieve.

1.2 Interactivity

Another primary feature I felt was necessary was for the simulation to be easily interactive, e.g., the user should be able to tinker around with as many variables as I can reasonably expose, and especially to be able to create and inspect new grid patterns. For this, I judged some sort of Paint-like functionality would probably be necessary, with a camera and the ability to "draw" on-screen. I was fairly confident in my ability to implement at least the broad strokes of these ideas, although I was less certain about other features, such as saving, which later proved to be more difficult.

Likewise, a proper user interface (UI) would also facilitate interaction; I knew from our homework and in-class exercises that Javascript provided easy event handling, which was convenient, but I was (and probably remain) inexperienced with HTML and CSS for the design itself, which later resulted in me greatly underestimating the amount of time necessary to get a decent-looking web page operational.

1.3 Extensibility

Finally, and most generally, I wanted the program to be able to handle other cellular automata distinct from the Game of Life, or at least to be designed in such a way that this functionality could easily be implemented later. My original "stretch goal" was to move to continuous automata, like the currently existing SmoothLife, and take further advantage of GPU processing to accelerate the process, but this proved too time-consuming due to the fundamental differences between discretized code and any continuous implementation; I will probably return to it later.

2 How It Works

Like any user-facing program, the end result of my project is complicated with multiple interacting components; I will attempt to break down its operation below.

2.1 The Compute Shader & Timestep Loop

The heart of my project, the compute shader (main.vert and main.frag in the code) is responsible for actually performing the simulation. Its primary functionality is dependent on two framebuffers, which act as render targets for their associated textures. I had originally planned to use SSBOs for data storage and manipulation, only to later discover that WebGL does not currently support them, unlike its older cousin OpenGL with which I had previous experience. As a consequence I was forced to use textures as data storage objects, which is not ideal but did well enough for my purposes.

When a draw command is called with the compute shader bound, the compute shader renders a quad filling the whole viewport (and thus the whole texture of its render target), and writes to it whatever texture is currently bound, stepped forward one iteration of the simulation.Thus, by continuously switching back and forth between two framebuffers, one holding the current iteration and the other receiving the next iteration, a timestepping loop can be formed; this is accomplished by having the timeStep() function call itself after a user-specified delay, which determines the simulation tick rate.

The actual mechanism of the shader is not overly complicated either: At each pixel on the screen, the shader will grab the current coloration of the pixels one grid-step away from the current one, as well as its own value, and compare that to the current coloration of living cells. A running tally collects the number of live cells surrounding the current pixel, which is then compared against the values in the selection rule uniform to determine what the simulation should do next. Each index of the selection rule array corresponds to the behavior of a cell with that number of neighbors, e.g., the value in index 2 holds information about the intended behavior for a cell surrounded by 2 neighbors. The value at that index represents the intended behavior: 0 means that a living cell should survive but a dead cell should not be born, 1 means a living cells dies but a new cell is born, 2 means a living cell survives and a new cell is born, and -1

means that a living cell dies and a dead cell remains dead. The current pixel is then filled with the color corresponding to that option and the shader moves on.

2.2 The Camera Shader & Render Loop

The camera shader (and, to a lesser extent, the gen, refresh, and random shaders) is responsible for much of the interactive functionality of the project. It is essentially similar to 2D camera shaders, with the exception that it does not really have to perform any processing by itself, as it is merely copying the texture data from the current backing framebuffer to an HTML canavs. The only outstanding feature is its use of texelFetch() rather than texture() when reading data, as we want to avoid unnecessary texture filtering.

Likewise, the other three shaders are fairly simple as well: The gen shader creates a block of colored pixels exactly one grid unit wide at the location specified by startCoord (for grid painting), the refresh shader converts all colored pixels to a new color (or equivalently deletes them if that color is the background color), and the random shader creates a random grid of states, using a seed value created by Javascript's Math.random() and the lattice points of the current grid.

All of these inputs are triggered as necessary, either during or immediately preceding rendering based on user input. Like the timestep loop, the render() function also calls itself repeatedly, but unlike timeStep() it uses requestAnimationFrame() to remain in lockstep with the normal browser framerate and avoid screen tearing. It should be notable that these are two separate threads that run independently of each other, allowing user input to be processed effectively even when the simulation is paused, or allowing the simulation to run at speeds greater than that of the render loop.

2.3 Events & HTML

As is typical in Javascript, a substantial fraction of the code is devoted to event handling, monitoring the large number of potential hotkeys (as detailed in the "Controls" section of the webpage) for potential keypresses and updating accordingly; likewise, an equal number exist for the various buttons and sliders in the control panel, either devoted to book-keeping (like keeping the sliders and text boxes synchronized) or to more complex behavior one probably wouldn't want to trigger accidentally (like the Randomize button). Of particular note are the import and save functionalities, which transfer data to or from the WebGL textures from or to an image, respectively; these took a particularly long time to implement due to the level of abstraction in use with WebGL, as well as Javascript's seeming reluctance to save files in any meaningful way.

Also resulting in delays was the length HTML/CSS necessary to properly configure the control panel. As mentioned previously, my unfamiliarity with CSS resulted in some delays as I grew more accustomed to web development, exacerbating an already tedious process. I do feel the final result is worthwhile, however, and neatly encapsulates all of the functionality I wanted to add originally.

3 Results & Conclusion

As this project represented a proof-of-concept rather than an actual research question, I will use this section to reflect on the final state of the project. Overall, I am quite pleased with the program; almost all of the functionality I wanted implemented was

done so successfully, although of course the devil is in the details, as much of it is not so polished as I would like (see: Saving, trying to draw wide swathes of pixels at once, that weird thing the grid does when zooming in, etc.), but this is ultimately a small part of the final product. I do wish I was able to get to implementing SmoothLife, but as I stated earlier this would have required writing an entirely new set of shaders and thus ultimately proved impractical to complete in the time I had (blame the endless nested grid containers in the CSS portion). Being a web-based project, it lends itself quite well to external hosting, which I plan to do sometime soon, and presumably from github.io; I will likely continue development if possible.

In terms of broader implications, I was quite impressed with the ease of use and efficiency of the shader pipeline once the initial hassle of compiling it was out of the way, and will probably try to exploit GPUs more in future projects. All in all, I believe this serves as an excellent proof-of-concept for the utility of GPUs in simulation, as well as an interesting and easily portable tool for exploring the fascinating behavior of cellular automata.