CS195V Project 1 - Life

February 2, 2012

1 Overview

If you took CS31, you might remember the "Life" project, which was an implementation of John Conway's cellular automaton. Now, we will implement a similar program using OpenGL and GLSL. Since each cell's next state can be determined in parallel, the algorithm for Life transfers well to the GPU. Using a large framebuffer as the simulation board, we will be able to simulate a huge number of cells at high speeds.

This project is due at 11:59PM on the day of class two weeks after the assignment date (February 14).

To get the support code, you will be using our very advanced version control system GOCP-R also known as "good ol' cp -r". The support code is in the /course/cs195v/gpusupport directory.

Before you run the code, you will have to run cmake once to create the proper files for compilation. Just run "cmake ." in the gpusupport directory after you have copied it over. You will also want to modify your Qt project settings to point to the correct executable so that you can easily run it from QtCreator. The executable should be located at gpusupport/projects/life/life.

2 Requirements

You will need to implement a working simulation of Conway's game of life. The main purpose of this assignment is to be a refresher for OpenGL and GLSL, and to familiarize yourself with the new language features. Thus, a good portion of the assignment will focus on setting up the infrastructure to run your simulation, i.e. Vertex Buffer Objects (VBOs), Framebuffers, and Shaders.

The computation part of this assignment should not be too difficult, so we expect a good deal of work on the visualization slide. Try to experiment with different shaders and post processing effects, and be creative with your 3D visualization. You want to be able to present this in demo reel!

Note that you probably could do all of this project in WebGL if you wished. We will not stop you, but keep in mind that future projects will make use of new OpenGL features not yet supported by WebGL. That said, for this project it should be fine.

You should avoid using deprecated OpenGL features whenever possible. This includes glBegin/glEnd primitive specification and the GL matrix stack. If you are not sure if a feature is deprecated, ask the TAs.

3 Technique

To hold our simulation states, we will require some sort of array, in our case, a framebuffer. Each pixel in the framebuffer can represent a single cell. With two framebuffers, we can hold the current simulation step and calculate the next one. To execute the simulation algorithm on each cell, we can draw a full screen quad and use a specialized fragment shader.

4 Instructions

You will need to make modifications to the primitive class to set up VBOs and VAOs for your choice of geometric primitives. For the Life simulation itself, you should modify the lifeengine class.

5 Tips

To create a geometric primitive...

- Set up the vertex array
  - glGenVertexArrays
  - glBindVertexArray
- Set up the vertex buffers
  - glGenBuffers
  - glBindBuffer
  - glBufferData
- Set vertex attributes
  - glEnableVertexAttribArray
  - glVertexAttribPointer

You will also need to set up your shader to bind certain vertex attributes with the shader's input parameters using glBindAttribLocation. *You need to bind all of the attributes before you link the shader program.*

To run your simulation, you will need to draw a full screen quad (both for simulation and for actual display). To create a quad, you will need a GLQuad

Object which is the size of your window. To make it fullscreen, call the vsmlOrtho() method on to set up an orthographic projection, then draw the quad.

In your simulation shader, you will have to index into the neighbors of a particular cell, given only the current cell's texture coordinate. In this case, you may find the textureOffset() method of GLSL to be useful.

For your 3D visualization, you may want to draw many of one type of primitive to represent your cells in 3D space. In this case, you may find the draw(int instances) method in your primitive class to be useful. glDrawInstanced will draw a specified number of instances of your primtive, and you will have access to an instance ID in your shader, so, for example, if you had 10 primitives and wanted to draw them in a line, you could use instanced drawing and translate them in the vertex shader using the instance ID to determine the amount.

6 Support Code

The support code provides a (relatively) easy way to spawn cells. Use the enter key to start and stop the simulation, left and right arrows to cycle through different patterns, left click to apply the pattern and right click to erase. Use the backspace key to switch between 2D and 3D visualization. We have provided a simple shader program that colors everything white and a simple quad primitive class. Use these as an example to build your own shaders and primitives.