User Guide

Game of Life Introduction

The game of life is a game of cellular automata. The game revolves around a grid of squares, or cells, which are either alive or dead. Cells either become alive, remain alive, or die based on how many of their neighbors (8 adjacent squares) are alive. The game of life represents populations in this way, as living cells die when there are too few or too many neighbors, and it takes optimal conditions for a cell to be brought to life. Every generation, or discrete time step, the following rules are applied to all cells in order to run the game:

- A living cell remains alive if it has exactly two or three living neighbors
- A dead cell is revived if it has exactly three living neighbors
- All other cells die

Wikipedia has a great article for further reading (http://en.wikipedia.org/wiki/Conway's_Game_of_Life)

Compiling and running

the main cpp file (main.cpp) can be compiled using the included makefile, or by running the following command in the shell:

g++ main.cpp -lglut -lGlu - lGL

The resulting executable can then be run, and should open one window titled “Game of Life”.

Drawing life on the screen

When the program is first run, the program window should initially show a 16x16 grid of white squares. These represent the cells in the game of life. The x and y dimensions can be increased or decreased by pressing ‘x’, ‘X’, ‘y’, and ‘Y’. The cells can be drawn on or erased by clicking and dragging across the grid. The color to be drawn can be changed by right clicking and navigating the menu of colors.

Running life

Pressing the spacebar will play or pause the game of life. The game of life runs at a fixed speed, which can be increased or decreased by pressing ‘s’ or ‘S’ respectively. Alternatively, pressing ‘n’ will step through one generation of the game of life. When the game of life is running, cells will die and be revived based on how many living neighbors they have. How edges are handled by the game (whether they wrap or are considered walls) can be changed in the right click menu.

Viewing

The game of life can be viewed in multiple ways. Pressing ‘0’ will bring the user to the drawing plane. Pressing ‘1’ displays the game of life as a texture on a plane. Pressing ‘2’ displays the game of life as a texture on a toroid. Pressing ‘3’ displays the game of life as a texture on a sphere. Pressing ‘t’ toggles between using closest pixel or average pixel color in texture displays. Pressing ‘p’, ‘P’, ‘q’, and ‘Q’, will increase and decrease the number of 3D faces of the sphere or toroid in the x-y and y-z planes. In addition, the sphere and toroid can be rotated using ‘,’, ‘.’, and ‘/’, with the shift key reversing the rotation direction.

Command list

Left-Click/Drag	- draw and erase cells of life
Right-Click	- open up a menu to change life color and wrapping rules
Spacebar	- plays or pauses the game of life
‘n’	- step through one generation of game of life
‘s’	- speeds up game of life
‘S’	- slows down game of life
‘x’	- increase number of cells in x direction
‘X’	- decrease number of cells in x direction
‘y’	- increase number of cells in y direction
‘Y’	- decrease number of cells in y direction
‘0’	- view the drawing plane
‘1’	- view the textured plane
‘2’	- view the textured toroid
‘3’	- view the textured sphere
‘p’	- increase the number of 3D faces in x-y plane
‘P’	- decrease the number of 3D faces in x-y plane
‘q’	- increase the number of 3D faces in y-z plane

‘Q’	- decrease the number of 3D faces in y-z plane
‘t’	- toggle between linear averaging and closest pixel in texture maps
‘,’	- rotate 3D models in x direction, use shift to reverse
‘.’	- rotate 3D models in y direction, use shift to reverse
‘/’	- rotate 3D models in z direction, use shift to reverse

Design Decisions

3D projections

When starting out we first had to decide how we wanted to get the game from a 2D board to the surface of 3D object. The first thought was to construct the surfaces so that vertices defined quads and simply make each face a cell mapping cell color to vertex colors. This proved much much more complex than we would like, so we instead opted to expand the board into a texture at draw time and then simply map that texture onto each surface. This allowed us to much more easily add new surfaces as all that was required was to define a parameterization of the surface.

Sphere Neighbors and Wrapping

The other big decision was how we wanted to handle adjacency on the sphere. Specifically we weren’t sure how the game should behave near the poles since it doesn’t easily map to quadrilaterals. A few different approaches were considered. The first idea was to treat the sphere like a blown out cube, letting 4/6 faces act like a donut, with one face at the top and bottom that was adjacent to the top row (or bottom row) of the 4 normal sides. Or we could just say they all met at a point and so every square on the top row was adjacent to every other square on the top row (or likewise for the bottom). What we ended up settling on was to instead keep a constant number of neighbors by saying that cells at the pole were adjacent only to the square opposite them and the squares to the left and right of those opposite squares.

Implementation

The game runs in one of 4 modes. The first mode draws a grid intended for picking an initial state and simply draws a grid of rectangles that we color as the user interacts. The second mode draws a single rectangle and is

a texture. The texture is constructed from the basic board in a subroutine and then mapped to the four corners of the square.

The other two modes work in the same fundamental way. First they fill out the texture, then they use a parameterization of the intended surface to fill an array of vertices and an array of texture coordinates which we tell OpenGL about. The scene is then drawn from these arrays as a series of triangle strips. The only difference between these modes is the parameterization used to generate vertices. It’s worth noting that the parameterizations give vertices centered at the origin which we then translate out to the center of the view box so that rotations can be done more easily.