4D Cantor Fractals

Andrew Kazenas

December 9, 2015

1 Abstract

4-space is an abstract theoretical concept that can be difficult to understand without helpful visual aids. Fractals are an example of this. Combine the two, and the result can be quite complicated and confusing. My project is Javascript code that will allow its users to study the Menger Hypersponge and Sierpinski 5-cell by rotating these objects in various directions. This allows for better visualization and can be used to create some interesting fractal art.

2 Background

Fractals are bizarre mathematical objects that exist between the standard three dimensions that humans are used to. Unlike simple lines and shapes that clearly have integer dimensions, fractals have so called fractional dimension which causes properties like a closed planar figure having an area of 4 inches and a perimeter of infinite length. The most simple and common fractals are created by starting with an ordinary shape like a cube or a triangle, and applying some sort of process to it which results in multiple smaller versions of the original object. You can then apply the same process to the smaller versions of the original. The limit of repeating this infinitely many times will often be a fractal. For example, to get the Sierpinski Carpet, start with a square and divide it into 9 equal squares. Remove the middle square, and then divide the remaining 8 squares into 9 squares each. Remove the middle square from each of these groups, and continue infinitely. There are other kinds of fractals, but those are much more complicated and will not be the focus of this project.

Many fractals have analogs in higher dimensions. For example, the *Cantor Set*, which exists in 1-space, can be generalized to either the *2D Cantor Dust* or the *Sierpinski Carpet* in 2-space, and then to the *3D Cantor Dust* or the *Menger Sponge* in 3-space. Using similar methods, it is possible to generalize fractals into 4-space. This is a coordinate system obtained by taking standard 3-dimensional space and adding an additional spatial axis. This way, you can have fractals with dimension above 3 but not quite 4, some of which will have infinite volume but zero 4th dimensional "girth."

It is common for fractals to be represented graphically by showing each iteration of the fractal up until the differences become too small to notice without zooming in. This is pretty simple when dealing with fractals in 1 through 3 space, but displaying a 4-dimensional object in a 3-dimensional reality can get tricky. To solve this problem, we use the same method used to represent 3-dimensional objects on flat surfaces like pieces of paper or computer screens: shadows. Whenever you see a diagram of a cube or a pyramid, all you are seeing is a projection of the 3-dimensional object onto a 2-dimensional plane. Computers can make these 2-dimensional projections appear 3-dimensional by rotating the direction in which the projecting is being done in a way that makes it look like the object is spinning in more than just the two dimensions on the screen. You can use similar tricks to project 4-dimensional objects into 3-dimensional space, and then onto a 2-dimensional screen. It's a lot of projection, but color-coding parts of the object can help you keep track of what's moving where and picture what the object is supposed to look like.

For my project, I will use Javascript to create a webapp that uses colored shadows to display the nth iteration of the Menger Hypersponge or the Sierpinski 5-cell for intellectual and artistic purposes.

3 Completed Goals

1. Become proficient in Javascript.
2. Create a web app that is able to display a shadow of the first three iterations of the Menger Hypersponge and Sierpinski 5-cell.
3. Allow the user to rotate the object in all 6 directions.

4. Create a version of the Sierpinski 5-cell that is compatible with Firefox and mobile browsers.

4 How the Code Works

The first section of the code sets constants, and initializes variables that I will use later. The most confusing part is that the program is both keeping track of the directions of the axes in four-space (the otus variables) and keeping track of what positions these directions map to on the canvas (the coord variables). Keeping track of which is which can be confusing, especially because the names of the axes we're keeping track of are obviously the same as the names of the coordinate axes in 4-space we are using to track their position. As a general rule, if it comes in only "x" or "y" varieties, it's a canvas-based variable, and if it comes in "x," "y," "z," and "w" varieties, it refers to one of the axes that rotates around. The 4-space coordinates that we're using to measure position are only represented as indexes of an array.

Next, you find the "rotate" function. This function is called to rotate the axes, thus changing our view of the object. All methods of manipulation call this function to get the job done. The first thing the function does is determine which two coordinates are being affected. I numbered the 6 directions of rotation 0 through 5, and the activeDirections array is created with a "1" in the index of any coordinate which is being affected and a "0" in the coordinates which are being left alone. The program then runs through all 4 axes and translates their two affected coordinates from Cartesian to polar. It then increases the theta coordinate by however much the rotation called for before translating the polar coordinates back into Cartesian. Once this is done, the axes have successfully been rotated, so the "render" function is called so that the display accurately reflects the changed internal values.

"updateAxisLocations" is a function that translates the axes' 4-space coordinates into locations on the canvas, then sets the canvas location variables to these new values. It's called only at the beginning of the render function, and is separated so the render function can be clean and easy to edit. The render function is next, and it does exactly what you'd expect. It finds the canvas on the page, clears it, finds the middle of the page, puts the pen there, and ex-

ecutes the instructions to draw the object. "updateCoords" is a function that takes the canvas locations of the axes and scales them up or down so that the same drawing instructions can be called for different levels of the fractal and produce the correct shape at whatever size and layer is currently appropriate based on what the user has manipulated on the page.

This brings us to "recursiveDraw," which is the part of the program that actually draws the fractal. It takes a size variable and a layer variable. If the layer variable is set to 0, the function will draw a 5-cell of whatever size it gets fed at the current location of the pen. If the layer is bigger than 0, it will jump around to the origin points of each of the six smaller 5-cells that result from iterating the fractal, and at each point it will call the function itself with half of the size that was input and one less layer. This recursion allows the fractal to be drawn with an arbitrary number of layers.

The final three functions all have to do with user input. The "acceptKeyInput" is simple. It calls the rotate function with the direction that whatever key is being pressed maps to and the rotation amount equal to whatever has been entered into the speed box. This function will be called whenever the EventListener on the page picks up a keydown event. The "iterate" and "deiterate" functions, which are called by the similarly named buttons on the page, will simply move the number of iterations variable up and down. There are checks in place to prevent the number of iterations from going below zero and warn the user if they are about to push the number of iterations high enough that rendering the fractal would be especially taxing on their web browser.