HyperSnake

Vassil Mladenov

December 11, 2013

Abstract

The purpose of this project is to present the concept of basic four-dimensional geometry in an interesting and inventive way by building a four-dimensional rendition of the classic Snake game using Python and OpenGL, henceforth referred to as PyOpenGL. For the unfamiliar, Snake is a survival-type game in which a snake runs across the screen in one of the cardinal directions specified by the user, attempting to eat as many apples as possible without intersecting its tail. There is only one apple on screen at a time and each apple eaten increases the snake's length by 1 unit. The following is an effort to document the project and to hopefully serve, in conjunction with the project source, as an API to help explore far more complicated topics of four-dimensional geometry.

1 Mathematics

1.1 Hypercubes

Just as a cube is a three-dimensional analogue of a square, a *hypercube* is an n-dimensional generalization of a cube. The hypercubes dealt with in HyperSnake are four-dimensional hypercubes, also known as *tesseracts*. Throughout this document, I will use both of the words hypercube and tesseract - let it be noted here that all of the hypercubes referenced are of dimension 4.

A tesseract has 16 vertices and 32 edges. This gives us an interesting property - namely, that the graph formed by the vertices and edges of a tesseract contains at least one *Euler cycle*[1]. An Euler cycle in a graph begins at a vertex and traverses every edge in the graph exactly once before returning to the original vertex. This means that one could take a piece of wire and bend it in four dimensions to construct a so-called "wireframe" tesseract. The same cannot be said for a cube. There is no way to bend a wire in three dimensions to construct a cube in such a way that each edge is only traversed once. The importance of this property of tesseracts will be discussed in 2.2.

As we are bound by our three dimensional world, there is no way to directly represent a hypercube as a rigid structure. Consider a wireframe cube centered at the origin. Shining a flashlight at the cube from somewhere along the z-axis would create what is called a *stereographic projection* onto a screen somewhere below the cube. Rotating this wireframe cube in three dimensions would cause its projection (its shadow) to undergo all sorts of transformations that never occur in two dimensions. When does a regular two-dimensional figure have an edge that's inside of itself, or how does a square become a hexagon? If we look directly at one of a cube's corners, our eyes see just that - a hexagon with three edges converging at the middle at 120° angles, but we have learned to perceive it as a three-dimensional object. Analogously, what we see in HyperSnake is not a set of wireframe tesseracts in and of themselves but rather their **three-dimensional shadows**. This is why some of the edges appear to be inside the hypercubes, and this is why these hypercubes appear to change shape when they are merely rotating in four dimensions.

This is an understandably difficult concept to grasp. The entire purpose of this project is to make this concept more accessible. The next session will discuss the mathematics behind the actual game, regardless of dimensionality.

1.2 Game Engine

A *game engine* is, in simple terms, the framework upon which a game is built. The complexity and versatility of a particular engine determine the types of games that can be built atop the engine. Most popular game engines, like the Unreal Engine, can handle in-game physics, sound, artificial intelligence, and other more complex properties. For HyperSnake, we can get away with a far simpler engine.

In very basic terms, the snake itself only has two properties:

- its current direction, expressed as a unit vector $\mathbf{d}$ of the form $(x, y, z, w) \mid x, y, z, w \in \{-1, 0, 1\}$ and only one component is non-zero
- an array $\mathbf{S}$ of the locations occupied by its body, with each location expressed as a point (x_i, y_i, z_i, w_i) with coordinates bounded by the game space $\mathbf{\Lambda}$, which in this case is a tesseract centered at the origin.

Each element of $\mathbf{S}$ defines the orthographic center of that particular element of the snake. It is important to note that we are not restricted to four variables. An element of a snake could have 1000 coordinates, and it's directional vector would tell us in which of those thousand coordinates the snake is currently moving. It's possible to play snake as an entirely text based game by printing the position of the apple, the positions of all the snake's elements, and the snake's current direction, all at each advancement of the snake. It's all purely mathematical.

To keep things simple, I made all of the dimensions of $\mathbf{\Lambda}$ an equal length $l = 7.5$. That is to say, the vertices of $\mathbf{\Lambda}$ have initial values that can only be 3.75 or -3.75. The apple appears at a random location $\mathbf{a}$ inside $\mathbf{\Lambda}$ where $\mathbf{a} \notin \mathbf{S}$ and moves to a new random location subject to the same constraints each time it is eaten. The snake moves in increments of 0.5 units at each moment. Given that $\mathbf{S}_0$ is the position of the head of the snake, the game computes $\mathbf{S}_0 + 0.5 * \mathbf{d}$ at each turn and then acts based upon whether in front of it lies free space, an apple, or part of its body. Actually, the above computation is not exactly accurate. The following is the actual computation for the head's new position: $e = -3.5 + ((e + 0.5 + 3.5) \bmod 7.5)$, where e is the element of $\mathbf{S}_0$ that is in the direction specified by $\mathbf{d}$. The bounds on e are consequently -3.5 and 3.5, and the snake can wrap around to the opposite face of $\mathbf{\Lambda}$ when it collides with a face of $\mathbf{\Lambda}$. This is because the apple and each element of the snake are hypercubes of side length 0.5, which means that the bounds on the values of the coordinates of their vertices are -3.75 and 3.75. Thus, they fall precisely within the bounds of $\mathbf{\Lambda}$. It could be possible to eliminate much of these decimals by multiplying all of the constants by 4 or something else, but my values work for the game. All of the above math assumes no rotation. Rotation is acually re-computed at each advancement of the snake. Given the relatively low quantities of vertices by computer standards, this does not cause noticable performance hiccups.

1.3 4D rotation

To simplify the concept of rotation, let's consider three dimensions. Every rotation is defined as rotating an axis "into" another axis, with the other axis or axes remaining fixed. So, when we perform an XY rotation by rotating the x-axis into the y-axis by a positive angle θ, what we get is a counterclockwise rotation about the z-axis. A YX rotation would then be defined as a clockwise rotation about the z-axis. Thus, there are six types of rotations in 4D: XY, YZ, XZ, XW, YW, and ZW, and their reverses. With each rotation, the remaining two axes stay fixed. Observe that the orientations of such remaining axes do not change when performing a 4D rotation in HyperSnake.

Now, XY, YZ, and XZ do not touch W. They are simple rotations in 3-space and can be handled by changing our viewpoint. For the remaining three, we need rotation matrices[6]. The way these rotations are computed is by multiplying a rotation matrix $\mathbf{R}$ by a column vector $\mathbf{v}$, which corresponds to one of the vertices contained in $\mathbf{a}$, $\mathbf{S}$, or $\mathbf{\Lambda}$. These rotation matrices are

$$\mathbf{R}_{XW} = \begin{pmatrix} cos(\theta) & 0 & 0 & -sin(\theta) \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ sin(\theta) & 0 & 0 & cos(\theta) \end{pmatrix}, \mathbf{R}_{YW} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(\theta) & 0 & -sin(\theta) \\ 0 & 0 & 1 & 0 \\ 0 & sin(\theta) & 0 & cos(\theta) \end{pmatrix}, \text{and } \mathbf{R}_{ZW} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & cos(\theta) & -sin(\theta) \\ 0 & 0 & sin(\theta) & cos(\theta) \end{pmatrix},$$

with θ representing the angle of counterclockwise rotation. In order to maintain the rotation across advances, these angles are tracked with global variables and the rotations are applied as sequential matrix multiplication in the order $\mathbf{R}_{XW}\mathbf{R}_{YW}\mathbf{R}_{ZW}\mathbf{v}$.

1.4 Projection of 4D objects onto 3D space

Each 4D point is projected down to 3-space before it is displayed according to the stereographic projection formula given to me by the legendary George Francis. The formula projects (x, y, z, w) to $\left(\frac{x}{f-w}, \frac{y}{f-w}, \frac{z}{f-w}\right)$, where f is merely the position of the projection "flashlight" discussed in 1.1 along the w-axis.

2 HyperSnake

The following sections discuss the main functions of HyperSnake's source, which is located at `class198f13/mladeno2/pyopengl/finalProject/finalProject.py`. I chose not to use object-oriented design for this particular project because it would create an unnecessary bureaucracy of code. There is one snake, one apple, and one game space, all of which are handled by class variables $\mathbf{S}$, $\mathbf{a}$, and $\mathbf{\Lambda}$. The entire game operates by using the data supplied by these variables.

2.1 Core Geometry Functions

There are three functions that define the geometry of HyperSnake: `generateTesseractVerts`, `rotatePoint4D`, and `projectPoint4Dto3D`.

- `generateTesseractVerts` takes 2 parameters, a point and a side length. The point is to be the orthographic center of a tesseract with the specified side length. The function returns a list of vertices that are obtained by systematically adding or subtracting half of the side length from each of the coordinates of the point. Since each point has four coordinates, this gives us $4^2 = 16$ sets of coordinates (points), which is exactly the amount we need to define the vertices of a tesseract. The function then creates a new list to which it appends each of those coordinates at least once to define an Euler circuit around the tesseract, which means that the list returns 33 vertices in total, with the first and last vertex being the same.
- `rotatePoint4D` takes a point, an angle, and a plane of rotation and performs the matrix multiplication discussed in 1.3. The rotation matrix and the point are treated as a list of lists and a list, respectively[2]. The multiplication is performed by a nested for-loop that emulates the standard matrix multiplication formula $(AB)_{ij} = \sum_{k=1}^{m} A_{ik}B_{kj}$.
- `projectPoint4Dto3D` brings the game down to our three dimensions by taking a single point as a parameter, performing the operation in 1.4, and returning the 3D point.

2.2 Core Game Functions

Every refresh of the game is governed by the `advance` function. It performs all of the functions discussed in 1.2 along with updating the class variables. For efficiency's sake, the game does not recompute the position of each center in **S**, but rather, it appends a new center in the direction of motion and then removes the center at the end of the list if the snake has not just eaten an apple. The `update` functions merely encapsulate the rotation and generation listed in the previous sections.

Finally, the `draw` functions tell the OpenGL window to draw the tesseracts. There is no function built into OpenGL or GLUT (OpenGL Utility Toolkit) to draw a wireframe tesseract like there is for a wireframe cube. It appears that I would have to draw 32 separate lines per hypercube to draw the wireframes. This is where a fantastic draw mode named GL_LINE_STRIP[4] comes in. The function draws an interrupted line with as many interruptions as specified by `glVertex3f` inputs. In other words, it gives you a wire of infinite length that is bent according to the points fed to the draw mode. A quantity n of points input by `glVertex3f` will result in $n-1$ connected line segments drawn. Now, all we need to draw a tesseract is to have a set of vertices arranged in such a way that connecting them would define an Euler cycle around the tesseract's edges. Now we see the value of `generateTesseractVerts` - the function gives us exactly the aforementioned set of vertices we need.

The point of the above paragraph is to illustrate how little my project relies on the built-in functions of PyOpenGL. If I were making a 3D snake game, the graphics implementation would be trivial, as GLUT already has well defined functions for all sorts of 3D graphics. For HyperSnake, I define all of the points myself. I tell them how to rotate, and I tell them how to move through the game space. OpenGL just connects the dots and colors them for me. I interface with only the lowest of the low-level OpenGL functions, and that is what made creating HyperSnake so enthralling. Almost everything is coded from scratch.

2.3 OpenGL-required Functions

Python, like Java, looks to run the `main` funtion in its code. Thus, in our code, we define the functions required by PyOpenGL any way we like, and in `main`, we tell GLUT that those are the functions that we want to use. For example, `glutKeyboardFunc(keyboard)` tells OpenGL that it is to handle keystrokes as defined earlier in the `keyboard` function. Similarly, `glutDisplayFunc(display)` tells OpenGL to display that which it is explictly instructed by the `display` function. The last function called in `main`, `glutMainLoop()` is where the Python script "lets go" of the program. From then on, the OpenGL window behaves autonomously according to the functions it has been told to access. In other words, if there is a function that is not called by `display` or `keyboard` or any of the other functions that we have told OpenGL to pay attention to, once the Python script hits the GLUT main loop, that function will never be called.

3 Miscellaneous PyOpenGL and LaTeX Discussion

The following is a compilation of knowledge that I have acquired in the process of completing this project that did not seem to fit with any of the other topics.

3.1 PyOpenGL

- `glColor3f` can be used anywhere, and will remain in effect until it is overridden by the next call to `glColor3f`, even if that is at the next iteration of the GLUT main loop.
- Any of the draw modes defined in the OpenGL documentation need to be wrapped by `glBegin(mode)` and `glEnd()`. Between those functions is where the `glVertex3f` calls are made.
- `glColor3f(1.0,0.0,1.0)` (full red, full blue, no green) yields a color that is nowhere near magenta. `glColor3f(1.0,0.0,0.5)` works better.

3.2 LaTeX

- Taking 10 minutes to set up the cross-platform Sublime Text editor with Package Control and LaTeXTools will improve your experience with LaTeX immeasurably. Single command building, common command shortcuts and completion, removal of auxillary files, and automatic BibTeX parsing are worth the minimal effort required to set up. For more information, please visit the LaTeXTools GitHub[5].
- Add \usepackage{url} in order to cite webpages with BibTeX.
- The LaTeX/Mathematics Wikibook[3] is an invaluable resource.
- Mac users, don't underestimate the power of BibDesk that comes bundled with MacTeX.
- Using \sum for summation will put the limits of summation next to the Σ. If you would like them to be at the top and bottom, use \sum\limits
- The `pmatrix` environment gives you a simple matrix with parentheses.
- If you need more than one character in a subscript or superscript, wrap them in braces like in the following code: R_{XY}

References

[1] How can we tell if a graph has an euler path or circuit? URL: `http://www.ctl.ua.edu/math103/euler/howcanwe.htm` [cited 11/27/2013].

[2] Two dimensional array in python - stack overflow, November 2011. URL: `http://stackoverflow.com/questions/8183146/two-dimensional-array-in-pythonhttp://stackoverflow.com/questions/8183146/two-dimensional-array-in-python` [cited 11/04/13].

[3] Latex/mathematics, August 2013. URL: `http://en.wikibooks.org/wiki/LaTeX/Mathematics` [cited 12/10/13].

[4] Silicon Graphics. glbegin, 2006. URL: `http://www.opengl.org/sdk/docs/man2/xhtml/glBegin.xml` [cited 11/15/2013].

[5] msiniscalchi. Latextools. URL: `https://github.com/SublimeText/LaTeXTools` [cited 12/08/13].

[6] Gilbert Strang. *Linear Algebra and its Applications*. Cengage Learning, 4th edition, 2005.