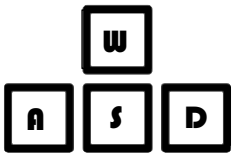
Garden of Eden
John Franklin

User Guide

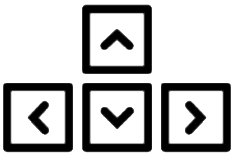
TO RUN PROGRAM:

1. Download the entire "Garden of Eden" folder
2. Open the file "Garden_of_Eden.html" with a browser

I implemented a three-dimensional space with a robust navigation system. I aimed to place some of my work from previous projects, as well as some new creations onto a planar grid. The jointed object I used was my plant structure from before, where I had branches grow and retract from a vase. I created a new tri-colored orb, a tetrahedron that reacts to lighting, and an axis drawer. Each object (including the planar grid) has an accompanying axis label to illustrate the x, y, and z directions for its matrix. To navigate the webpage:



Use the keys "WASD" to orient your viewport



Use the keyboard arrows to move within the 3-dimensional space

These controls emulate that of a video game. The "WASD" keys align similarly to the arrow keys in terms of usage. Feel free to expand/educe the window, as the aspect ratios stay in place.

Results

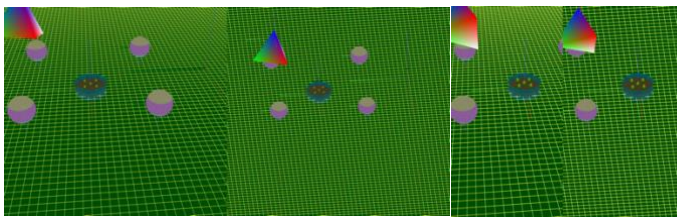


Figure 1. (left) The fullscreen aspect radio, (right) the same scene with a reduced browser window size. The image is not warped.

This project was certainly more difficult than the previous, but functioned as an important one. This incorporated lighting, viewing matrices, and further shape construction all into one scene. I really, *really* enjoyed doing this project: so much so, that I computed all of the math from scratch. I refrained form using cuon-matrix-mod for any vector math when computing the navigation angles. The windows now scale in size, where the canvas won't warp the shapes being animated. To navigate using the keys, I had to keep track of the eye, lookAt, and up points to be able to manipulate them later. Moving forward and backward was as simple as taking the vector between the eye and lookAt (or *radius*), and adding/subtracting a fraction of it to both perspective points. And tilting up was just a matter of adding to lookAt's z axis. Where it got tricky was turning horizontally, so that the lookAt point circles but the eye stays in the same location. What I ended up doing was calculating an initial Θ for the radius onto the xy-plane. Once a user pivots, the Θ is incremented

and the cosine/sine are taken. I then multiply that to the length of radius to reposition the lookAt in a new spot with the same distance. As for moving sideways, I calculated the cross product between the radius and the "up" vector. I then just reused my "forward()" function, except plugging in the cross product vector rather than the radius to move along.
The plant remains to be a complexly generated joint structure. It grows at a constant pace, and retracts back to a seed once it reaches full size. Each of the branches are composed of five cones stacked on top of one another, becoming increasingly smaller. Four of the branches have curvature, and each of the cones executes this curvature to create a "sagging" effect. The pot is comprised of two altered cones to give it a geometric shape.

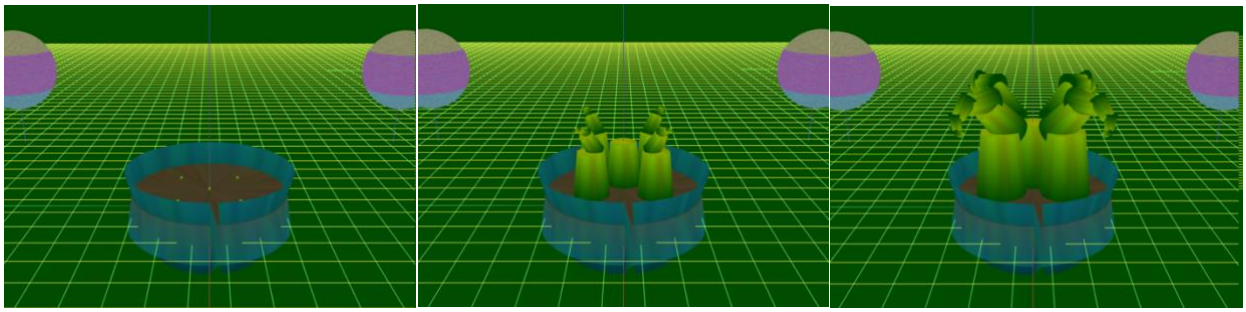


Figure 2. A growing plant from buds (left), to sprouts (center), to full-grown(right)

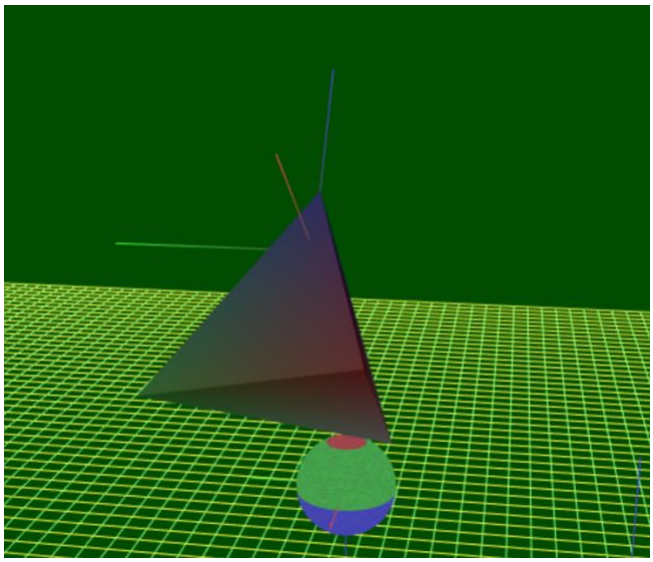


Figure 3. Frame of shaded tetrahedron with axis pointed out of it

The orbs are made up of 100 slices, each with 100 vertices to grant it a grainy texture. To color each orb, I explicitly gave a vertex a color assignment, with a specified amount of random variance. And each orb is divided into three layers of color structure. For example, the top layer is grounded at a beige color with a 25% range of random reduction to green to give the orb more yellowish spots. They are placed evenly in a square along the grid, and each have distinct color combinations. The layers heights are also randomly generated.

The tetrahedron is the only shaded object. I created a float called "bool" which controlled whether the pure color was passed on, or a dot product of the normal. When bool is greater than ½, the shader is applied. When it is not, the original color points are applied. The logic rests in the vertex shader, which was initially very hard to debug.

The axis markers were straightforward to implement. Using the “gl.LINES” parameter and manually mapped out line vertices, it was possible to write a “drawAxis()” function to handle placement for any items that needed marking. As mentioned beforem these are located at every major object in the scene. For viewing angles, the perspective view only requires a “perspective()” function, and the orthographic references an “ortho()” function.

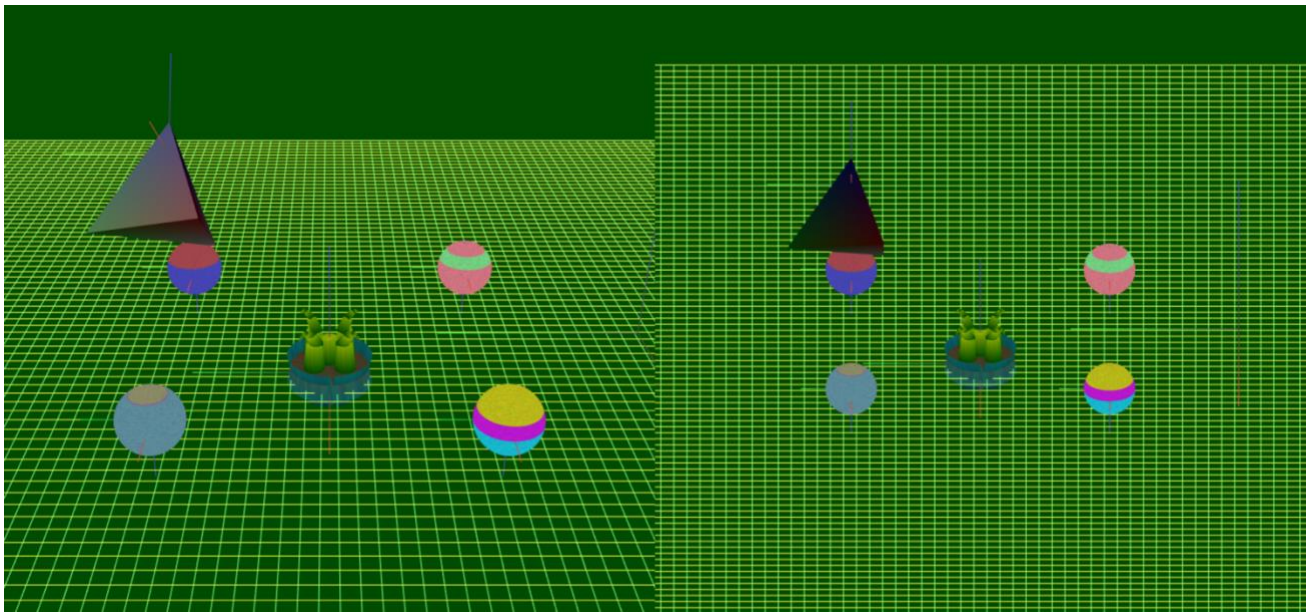
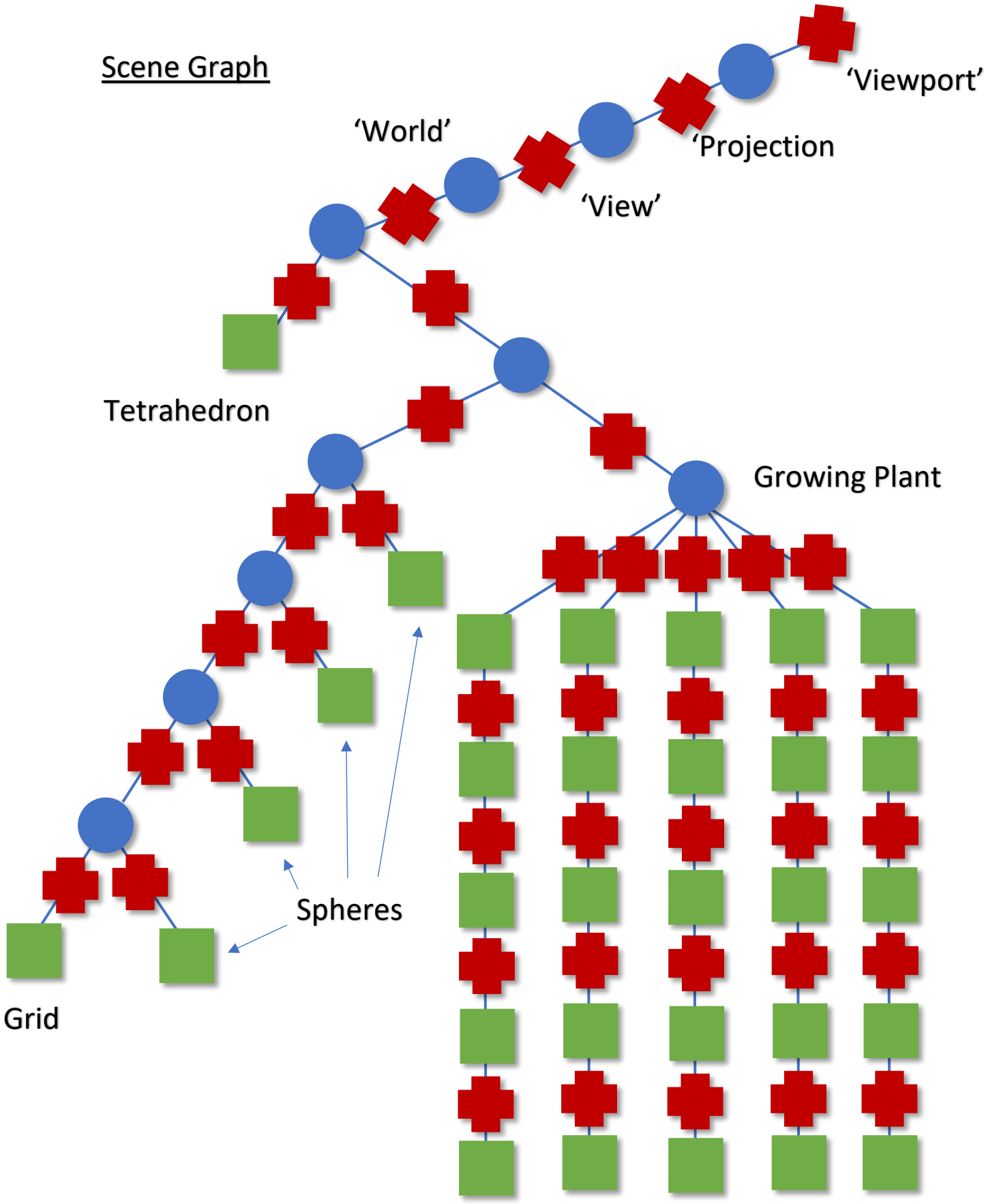


Figure 4. (left side) Perspective plane vs (right) orthographic plane. Camera angle is similar for both.


Scene Graph
'Viewport'
'Projection
'World'
'View'
Tetrahedron
Growing Plant
Spheres
Grid