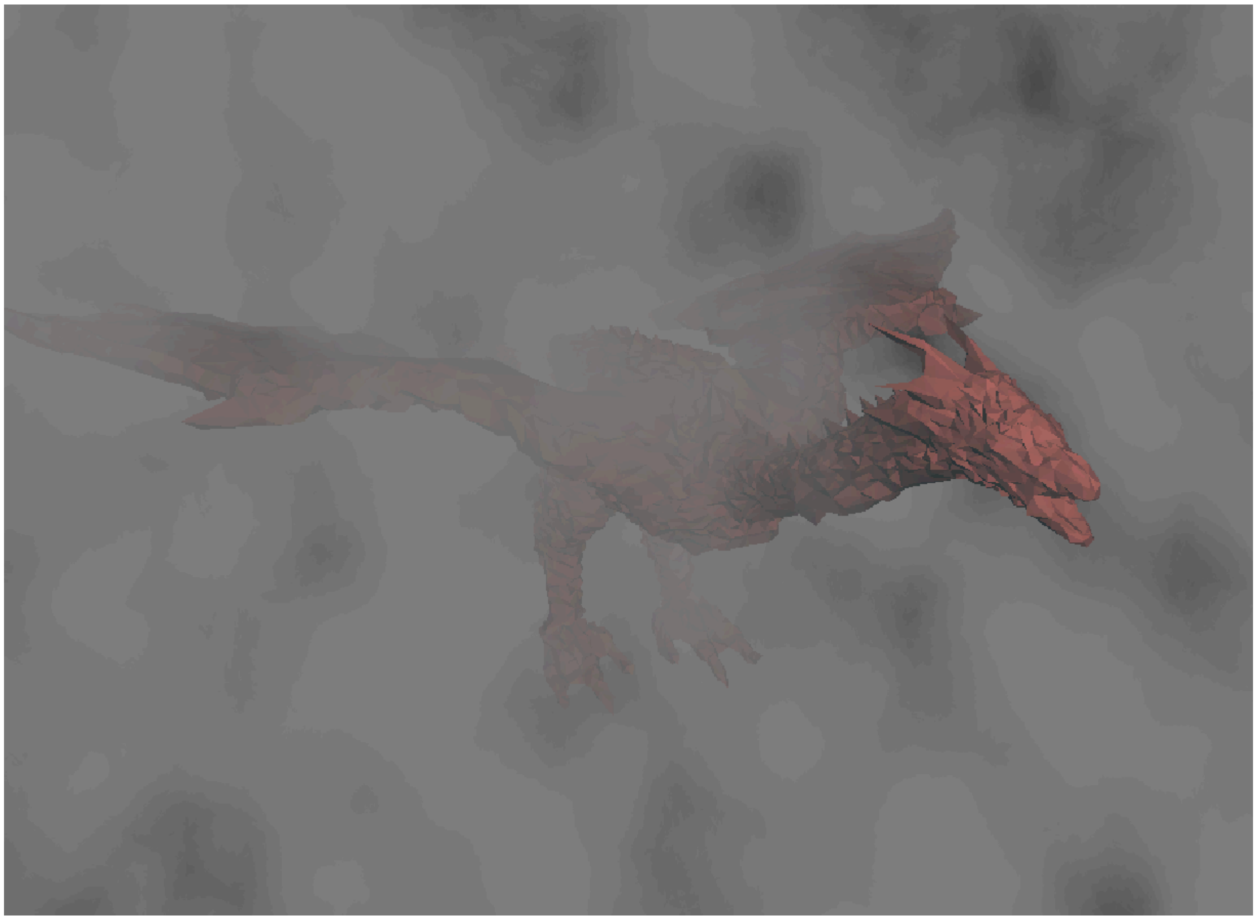
CS 457/557 -- Winter Quarter 2024

Project #6

The Dragon Menagerie Project

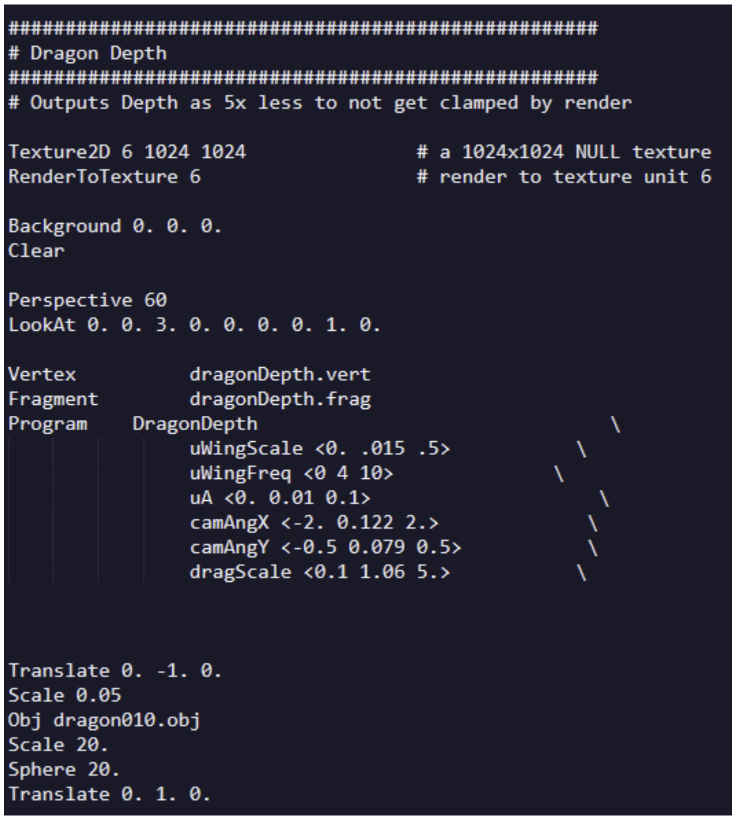
Dragon in the Fog

Donovan Burk - BurkDo@oregonstate.edu



GLIB Setup

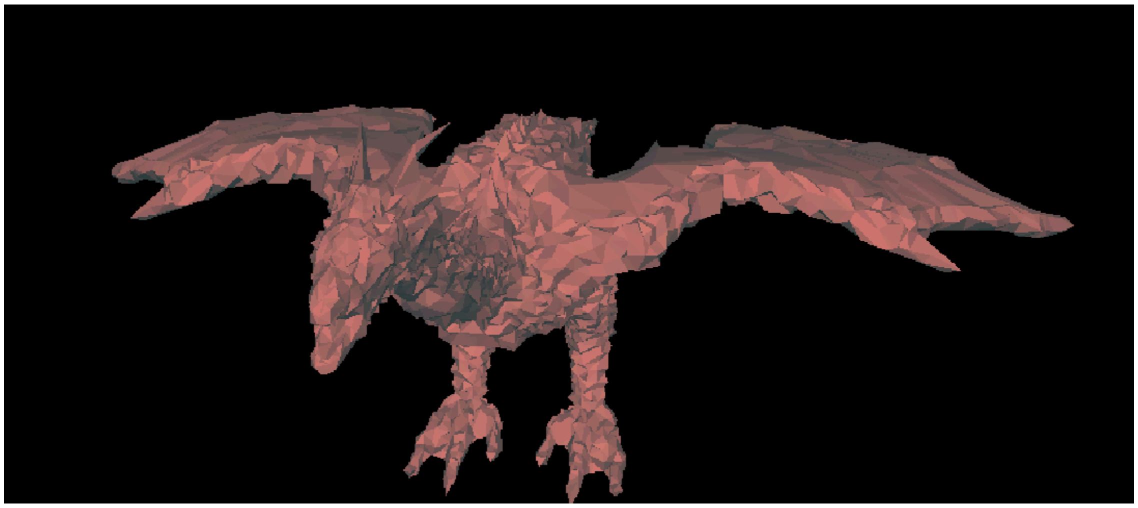
In order to set up the multipass rendering for buffer steps I have to set up the GLIB file to have programs render to textures and have later programs read their data as a texture file. Here is an example.



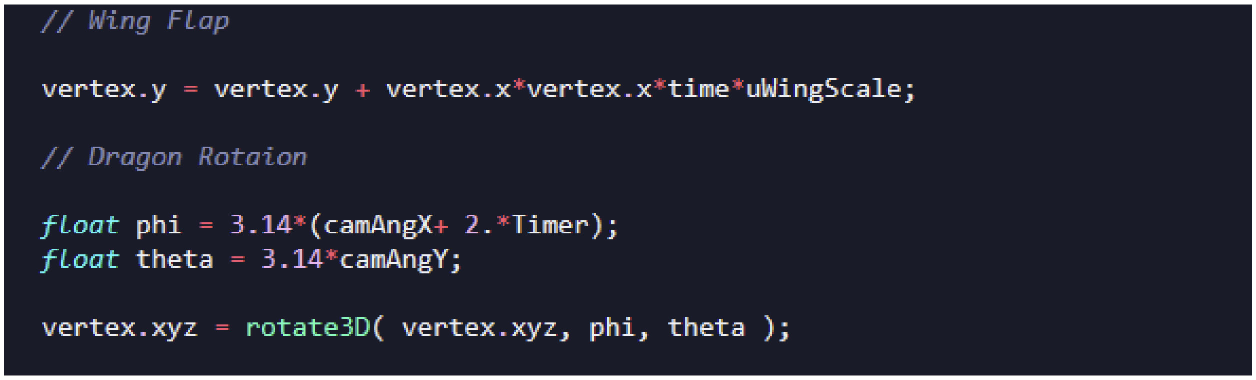
```
####################################################
# Dragon Depth
####################################################
# Outputs Depth as 5x less to not get clamped by render

Texture2D 6 1024 1024               # a 1024x1024 NULL texture
RenderToTexture 6                   # render to texture unit 6

Background 0. 0. 0.
Clear

Perspective 60
LookAt 0. 0. 3. 0. 0. 0. 0. 1. 0.

Vertex          dragonDepth.vert
Fragment        dragonDepth.frag
Program    DragonDepth                                 \
                uWingScale <0. .015 .5>            \
                uWingFreq <0 4 10>               \
                uA <0. 0.01 0.1>                     \
                camAngX <-2. 0.122 2.>             \
                camAngY <-0.5 0.079 0.5>           \
                dragScale <0.1 1.06 5.>            \



Translate 0. -1. 0.
Scale 0.05
Obj dragon010.obj
Scale 20.
Sphere 20.
Translate 0. 1. 0.
```

Dragon Color

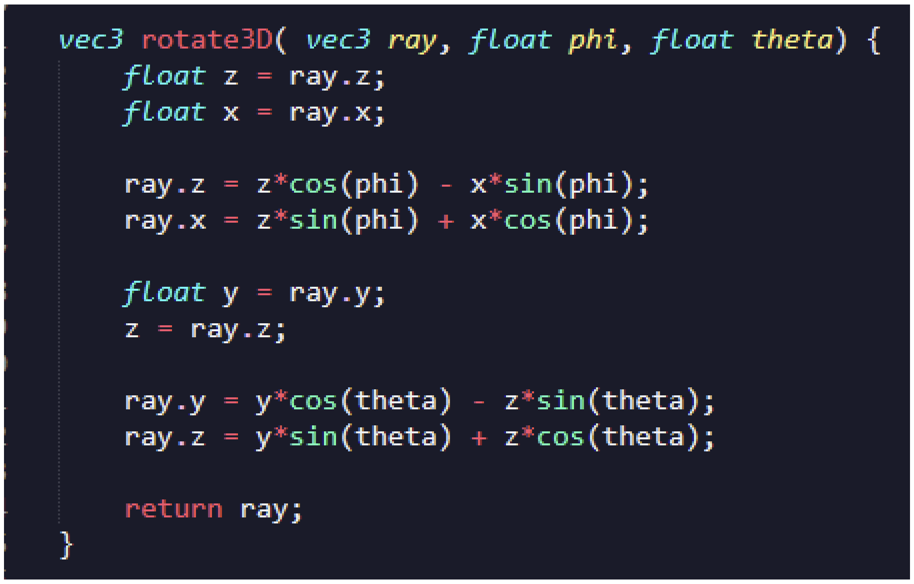
One of the first passes we do is the color and lighting pass for the Dragon we are rendering inside the fog. This will just be a basic lighting shader that approximates, ambient, specular, and diffuse lighting. Here we also spin the model and make the wings flap in the vertex shader here because multi-pass rendering is not too nice with multiple cameras.



Rotation code:



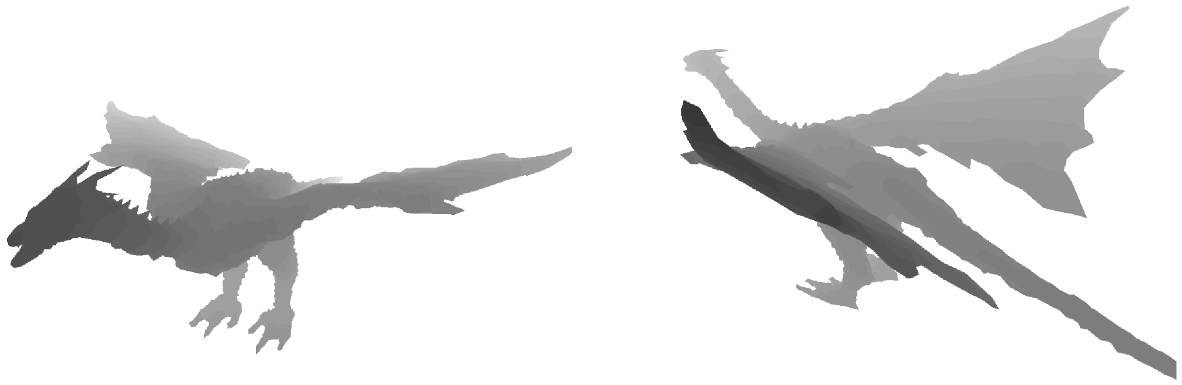
```
// Wing Flap

vertex.y = vertex.y + vertex.x*vertex.x*time*uWingScale;

// Dragon Rotaion

float phi = 3.14*(camAngX+ 2.*Timer);
float theta = 3.14*camAngY;

vertex.xyz = rotate3D( vertex.xyz, phi, theta );
```



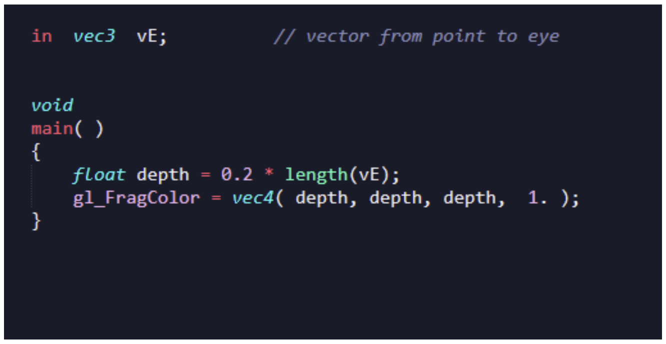
```
vec3 rotate3D( vec3 ray, float phi, float theta) {
    float z = ray.z;
    float x = ray.x;

    ray.z = z*cos(phi) - x*sin(phi);
    ray.x = z*sin(phi) + x*cos(phi);

    float y = ray.y;
    z = ray.z;

    ray.y = y*cos(theta) - z*sin(theta);
    ray.z = y*sin(theta) + z*cos(theta);

    return ray;
}
```

Scene Depth

In order to know how far into the fog to traverse before we expect to hit a solid object to render. The vertex shader for this is the same as the coloring vertex shader because we want to render the same scene, we just need to send different information to the fragment shader. Speaking of which, the fragment shader is very simple as all it does is make the distance from fragment to camera the color from black at zero distance and 1 at “5” away in order to get some distance data out and not get data cut off from color clamping. There is actually a large sphere around the scene to act as the background and get a large depth value for the background.



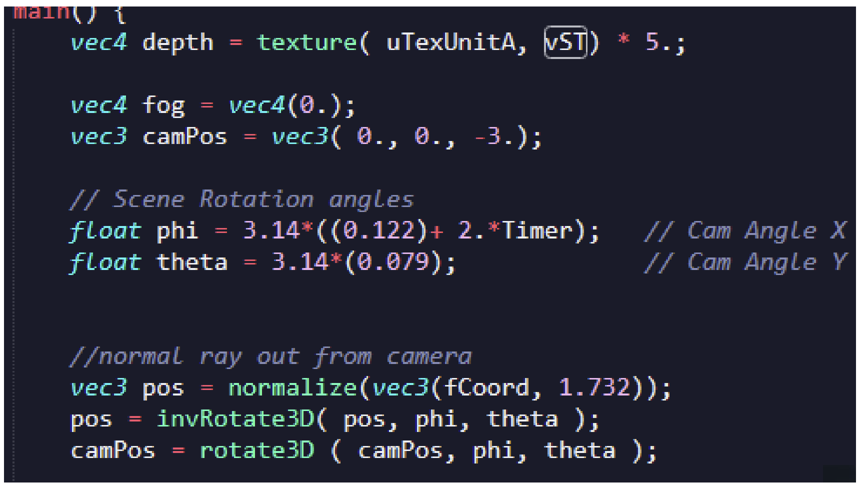
Two different angles showing distance works with rotation



```
in  vec3  vE;          // vector from point to eye


void
main( )
{
    float depth = 0.2 * length(vE);
    gl_FragColor = vec4( depth, depth, depth,  1. );
}
```

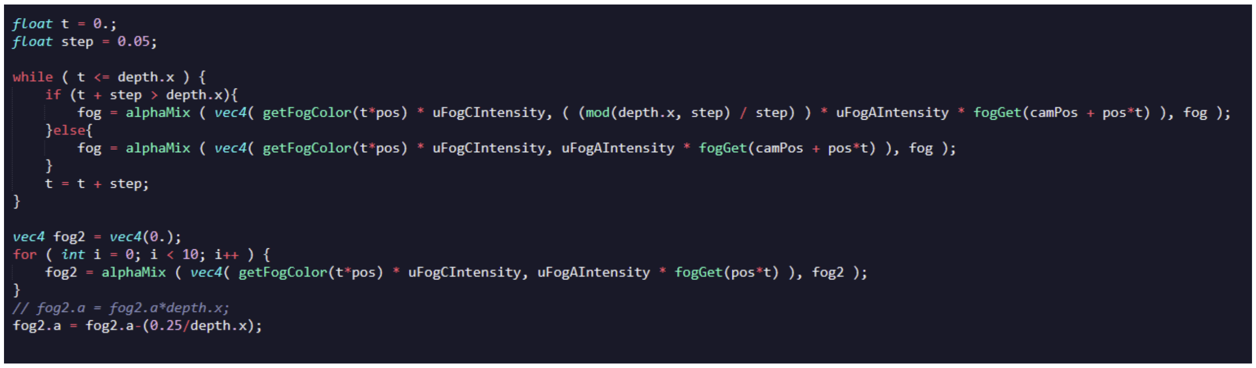
Entirety of the fragment shader.

Fog Ray Marching
The main fog rendering technique for the fog in this program is ray marching. In this case, we are sending rays from the camera for each pixel and at set intervals sampling the noise texture at that 3D point. Along the way we collect the samples into one fog vec4 with a maxing function that takes alpha values into account when determining how much show color shines through. Sampling continues until we reach the depth defined by the depth texture. There was also an experimental technique where we always took 10 samples then reduced fog by how close the depth texture said the pixel was.



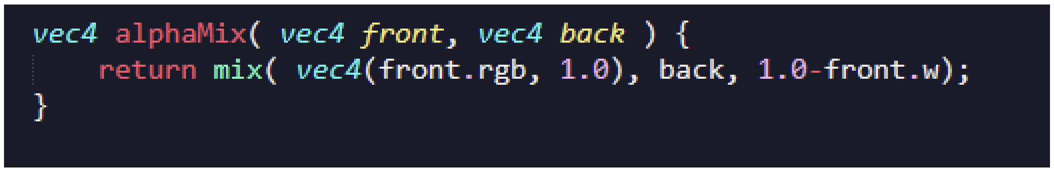
```
main() {
    vec4 depth = texture( uTexUnitA, vST) * 5.;

    vec4 fog = vec4(0.);
    vec3 camPos = vec3( 0., 0., -3.);

    // Scene Rotation angles
    float phi = 3.14*((0.122)+ 2.*Timer);    // Cam Angle X
    float theta = 3.14*(0.079);              // Cam Angle Y


    //normal ray out from camera
    vec3 pos = normalize(vec3(fCoord, 1.732));
    pos = invRotate3D( pos, phi, theta );
    camPos = rotate3D ( camPos, phi, theta );
```

Depth texture sampling and camera ray setup, Rotating camera to have fog rotate with dragon. Inv rotate does the rotation in the opposite order of normal rotate.



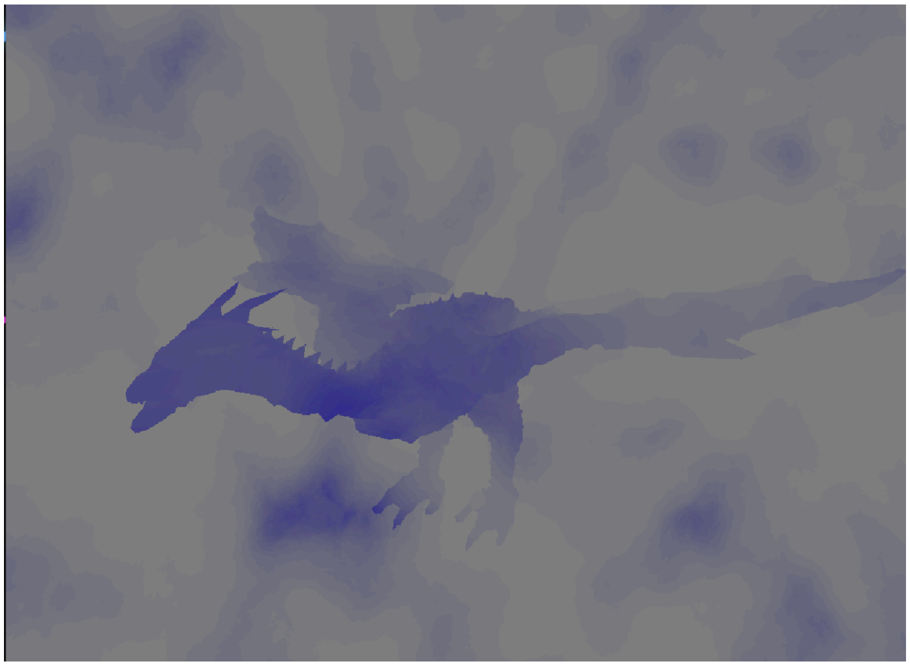
```
float t = 0.;
float step = 0.05;

while ( t <= depth.x ) {
    if (t + step > depth.x){
        fog = alphaMix ( vec4( getFogColor(t*pos) * uFogCIntensity, ( (mod(depth.x, step) / step) ) * uFogAIntensity * fogGet(camPos + pos*t) ), fog );
    }else{
        fog = alphaMix ( vec4( getFogColor(t*pos) * uFogCIntensity, uFogAIntensity * fogGet(camPos + pos*t) ), fog );
    }
    t = t + step;
}

vec4 fog2 = vec4(0.);
for ( int i = 0; i < 10; i++ ) {
    fog2 = alphaMix ( vec4( getFogColor(t*pos) * uFogCIntensity, uFogAIntensity * fogGet(pos*t) ), fog2 );
}
// fog2.a = fog2.a*depth.x;
fog2.a = fog2.a-(0.25/depth.x);
```

The main technique has a clause for the last fog sample to scale based on distance left inorder to avoid an obvious cutoff between a number of fog samples and one more sample. **Getfog** is a function that takes a position and samples the noise texture while also doing some adjusting in order to tweak the fog amount and density and minimum contribution amount. **GetFogColor** currently just returns gray. These two functions were made in order to prepare for possible scene configuration for more effects that would be outside the scope of the **Dragon Managirie**.



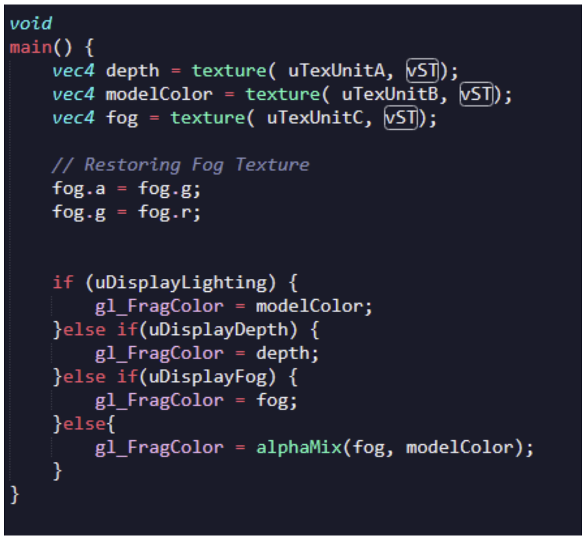
```
vec4 alphaMix( vec4 front, vec4 back ) {
    return mix( vec4(front.rgb, 1.0), back, 1.0-front.w);
}
```

Alpha considering mix function


This is a human readable version of the fog texture against a pure blue background. The actual texture stores color in the Red and Blue channels while using the green channel as an intermediate for the alpha channel because these programs render to a solid image with no alpha data. This picture is the result of restoring the alpha from green and copying the red value over to the green channel and displaying that as an image.

Master Layer

The final layer is just an image compositing shader that takes all the previous stages and displays them either one at a time or the final result, which does include a little calculation between the fog texture and the color texture with the alpha considerate mix function.



```
void
main() {
    vec4 depth = texture( uTexUnitA, vST);
    vec4 modelColor = texture( uTexUnitB, vST);
    vec4 fog = texture( uTexUnitC, vST);

    // Restoring Fog Texture
    fog.a = fog.g;
    fog.g = fog.r;


    if (uDisplayLighting) {
        gl_FragColor = modelColor;
    }else if(uDisplayDepth) {
        gl_FragColor = depth;
    }else if(uDisplayFog) {
        gl_FragColor = fog;
    }else{
        gl_FragColor = alphaMix(fog, modelColor);
    }
}
```

Youtube Video:
https://youtu.be/VG1mq9mGQKY